International Journal of Cooperative Information Systems,

A METHODOLOGY FOR QUERY REFORMULATION IN CIS USING SEMANTIC KNOWLEDGE*

DANIELA FLORESCU
INRIA Rocquencourt, 78153 Le Chesnay Cedex, France
Daniela.Florescu@inria.fr

and

LOUIQA RASCHID
University of Maryland, College Park, MD 20742
louiqa@umiacs.umd.edu

and

PATRICK VALDURIEZ
INRIA Rocquencourt, 78153 Le Chesnay Cedex, France
Patrick.Valduriez@inria.fr

Received (October 1, 1995)
Revised (May 1, 1996)
Communicated by (Huhns, M. and Singh, M.P.)

ABSTRACT

We consider Cooperative Information Systems (CIS) that are multidatabase systems (MDBMS), with a common object-oriented model, based on the ODMG standard, together with local databases that may be relational, object-oriented, or dedicated data servers. The MDBMS interface (or mediator interface) that describes this CIS could be different from the union of the local interfaces, that describe each local database. In particular, the mediator interface may be defined by semantic knowledge, that includes views over particular local databases, integrity constraints, and knowledge about data replication in local databases. We present a methodology for query reformulation which is based on the uniform representation of all semantic knowledge in the form of integrity assertions and mapping rules. A reformulation algorithm exploits this semantic knowledge, and performs semantic rewriting based on pattern-matching, to obtain a query on the union of the local interfaces. A decomposition algorithm then produces a composite query, and local sub-queries, one for each local interface. The reformulation is general enough to re-use the results of previously computed queries in the CIS. We have implemented this reformulation technique in our Flora compiler prototype which we used for validation and experimentation with O2 databases.

Keywords: mediators, query rewriting, integrity constraints, object query languages.

*This research has been partially supported by the Advanced Research Project Agency under grant ARPA/ONR 92-J1929, the National Science Foundation under grant CDA9422138, and by the Commission of European Communities under Esprit project IDEA.

1. Introduction

Recent advances in distributed systems and computer networks have led to a demand for high-level integration of heterogeneous information sources such as databases and file systems. In order to have significant practical impact on future information systems, these Cooperative Information Systems (CIS) must be both flexible and efficient. Heterogeneity in the context of database servers typically stems from multiple data models (*eg.* relational, object-oriented), different DBMS, and dedicated data servers such as file systems, WAIS servers, or image servers. CIS should contribute the necessary technology for interoperability of distributed, heterogeneous and autonomous data sources [23,30]. A CIS must provide transparent access to the participating data sources, which we call *local databases*. In our context, since we focus our attention on database servers, we use the term CIS-MDBMS to refer to the environment and architecture of these information systems.

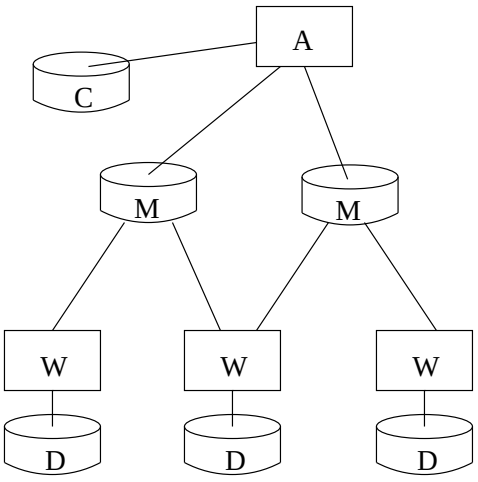

Figure 1: An architecture for networked heterogeneous servers A: application, M: mediator, C : catalog, W : wrapper, D : database

A general architectural approach that has been advocated in many recent research projects dealing with networked heterogeneous information servers, e.g., the TSIMMIS project at Stanford [25,26], and the Garlic project at IBM [7], is to identify the functions of *mediators* and *wrappers* or *translators* (see Figure 1). Typically, wrappers or translators convert data from multiple sources into a common model, and mediators integrate data provided by multiple translators, in the common model. We follow a similar architectural approach for our CIS-MDBMS environment. To achieve transparency of distribution and heterogeneity, at the level of a mediator, the CIS-MDBMS is based on a common data model and language. The ODMG standard [8], which extends the OMG object-oriented data model [22], pro-

vides a good basis for a common integration framework. We assume the existence of wrappers to handle query execution in the local database systems and provide data in the common model.

Consider a CIS-MDBMS which supports the ODMG data model and the Object Query Language (OQL) [8]. Local databases may be relational or object-oriented databases, or more specialized data sources (*eg.* multi-media servers). A database schema expressed in the ODMG model is called an *interface*. The CIS-MDBMS provides a single *MDBMS interface*, which is also called the *mediator interface*, and a set of *local interfaces*, one for each local database. Since the CIS-MDBMS must maintain the autonomy of local databases, which have been developed independently, each local interface will provide an exact correspondence to the entities of each local database. However, we do not want to limit the MDBMS interface (the mediator interface) and restrict it to the union of the local interfaces. For example, the mediator interface may include views over entities in one or more local interfaces, so that the mediator may combine data from multiple servers. Similarly, each local interface may only describe a view over the mediator interface.

In this context, our methodology for CIS-MDBMS query processing proceeds along the following steps: reformulating the input OQL query against the mediator interface into equivalent OQL queries on the union of the local interfaces; and decomposing the reformulated queries into local sub-queries (to be evaluated by corresponding wrappers, in each of the local databases), and a composite query to be evaluated in the CIS-MDBMS. We then select a composite query, for which we produce an efficient execution plan. The final step corresponds to optimization, and must use heuristics or a heterogeneous cost model, as in [11]. All of these steps are considered mediator functions, in the general architecture that has been proposed. Each of the local sub-queries, on the local interfaces, will be handled by the corresponding wrapper.

In this paper, we address the steps of query reformulation and decomposition. They are important for several reasons. First, the mediator interface may be different from the union of the multiple local interfaces, and reformulation allows us to obtain queries that we may execute on one or more of the local databases. Second, query reformulation can use semantic knowledge to develop alternate queries, and thus, explore good optimization opportunities. For instance, the fact that data can be replicated in several local databases, and that the results of previous query execution may be stored in the CIS-MDBMS for later re-use, can yield alternative ways of computing answers to the same query. Reformulation in this context is more than simply translating the query from the mediator interface to the local interfaces, since several alternatives may be explored, based on the available semantic knowledge.

To correctly reformulate a query, the CIS-MDBMS must know the mapping from the mediator interface to the local interfaces. We assume that some mapping information is obtained using schema integration techniques, *eg.*, [1,14,20]. An object in the mediator interface may not directly correspond to a particular object in the

local databases. It is also possible that only selected values from the local interfaces are of interest in the mediator interface; for example, the CIS-MDBMS may select only employee instances that occur in *all* the local interfaces, or the CIS-MDBMS may have a criterion to pick a value from a particular interface. There may also be knowledge on redundancy in the interface descriptions, which may result in data replication in the local databases, and therefore possibilities for alternate reformulations. Finally, there may be integrity constraints in the local or mediator interfaces which allow for query simplification. All this information is *semantic knowledge* which we exploit for CIS-MDBMS query reformulation, and query reformulation is made complex by the variety of the semantic knowledge that must be used to assure the correctness of reformulation, (*i.e.*, the reformulated query produces the expected answer).

Most work in query reformulation assumes a restricted language based on SQL or Datalog [18]. However, many application domains, *eg.*, scientific databases [6], typically use complex data structures, which cannot be easily described or queried using Datalog-like languages. The generality of the query language OQL, in our CIS-MDBMS environment, allows us to deal with such complex data structures and queries. OQL supports nesting of structures, functions and queries, aggregation, arithmetic operators, type constructors, *etc.*

Our methodology for CIS-MDBMS query reformulation relies on the uniform expression of semantic knowledge, which are either integrity assertions or mapping rules. They are rewrite rules, and are expressed in a canonical form of the OQL expression, based on a strongly typed algebra. The reformulation algorithm is a pattern-matching of well typed expressions, and uses both syntactic rewriting (to express a query in canonical form), and semantic rewriting (using the semantic knowledge in the form of rewrite rules). We note that our reformulation is more difficult, compared to reformulation with a Datalog-like language [18]. For a query against the mediator interface, we obtain alternate equivalent OQL queries in the union of the local interfaces. The algorithm also has the flexibility to re-use the results of previously computed queries that may be stored in the CIS-MDBMS.

The paper is organized as follows. Section 2 introduces the CIS-MDBMS query processing environment, and defines the CIS-MDBMS architecture, model and language. This section also defines the problem of CIS-MDBMS query reformulation, and includes an example describing some alternative reformulations. Section 3 defines the necessary semantic knowledge, in terms of integrity assertions and mapping rules, that describes the CIS-MDBMS environment. Section 4 presents a methodology for CIS-MDBMS query reformulation. We describe a reformulation algorithm, which uses syntactic rewriting and semantic rewriting, based on pattern-matching. Section 5 describes query decomposition for the CIS-MDBMS environment. Section 6 shows our experiments for query reformulation, within the *Flora* compiler/optimizer for the ODMG data model and query language. We compare our research with related research in this area, and then conclude. Earlier

results from this research have appeared in [13].

2. The CIS-MDBMS Environment

Our CIS-MDBMS environment supports a mediator based on a common data model and common query language. It provides transparent access to multiple, heterogeneous databases, through translators or wrappers. We present our assumptions wrt the CIS-MDBMS architecture, and then define the query reformulation problem for this CIS-MDBMS environment.

2.1. The Common Data Model for the CIS-MDBMS

The model and language used to describe each local database and the CIS-MDBMS is based on the ODMG standard [8]. We introduce the main elements of the object data model and query language (with minor changes).

The object data model is based on a type system. Types can be atomic or structured. The set of atomic types is the union of the set of predefined types, such as integer, boolean, string, and the particular set of object types of the application. Type constructors are the set, bag, list and tuple. Type expressions are constructed from atomic types, through the recursive application of type constructors.

Object types are described in the data model through an object interface. An object interface specifies the properties (attributes and relationships) and operations or methods that are characteristic of the instances of this object type. A relationship is a reference-valued attribute of the object type. An object interface, as defined in the ODMG model, allows the declaration of a key constraint, and the declaration of inverse links between object types. However, we support more general integrity constraints on object types. These integrity constraints play an important role in query reformulation.

The object types are organized along a subtype hierarchy. All the attributes, relationships and methods defined on a supertype are inherited by the subtype. Furthermore, the instances of a subtype satisfy all integrity constraints defined on its supertype. Object type extensions[a]can be explicitly named in the object type interface, in which case they are automatically maintained.

The set of operators includes built-in operators, user-defined functions and user-defined methods. The built-in operators are comparison and arithmetic operators, aggregation operators (*eg.*, count, min, max, sum, avg), set operators(*eg.*, union, except, intersect, flatten, element), list operators (*eg.*, append, first, last, nth), set membership operator (in). Special built-in operators are value constructors (*eg.*, set, bag, list and tuple constructors), field selection, quantifiers and select.

An object database is accessed through the set of *named variables* which define the entry points of the database. Named variables are used as handles for data of any type (integer, objects, set of any type, *etc.*). Named variables are persistent, (their name and value is maintained in the catalog), and can be referred to by any

[a]An extension is the set of all instances of a given object type and its subtypes.

query. Particular named variables are associated with extensions of object types that are automatically maintained. A *database interface*[b] consists of a set of object type interfaces, and a set of named variables (with their types). In the CIS-MDBMS environment, we assume that there is a MDBMS interface (mediator interface) and a local interface for each local database.

2.2. The Common Query Language for the CIS-MDBMS

The common query language used for expressing queries is OQL, a nonprocedural, functional language. OQL *queries* corresponding to an interface are well-typed expressions constructed in this interface. Given an interface, OQL *expressions* are syntactically constructed by a recursive application of user-defined and built-in functions, starting with constants and variables. Each OQL expression has an associated type. During reformulation, we assume that OQL expressions are well-typed, since type checking precedes query reformulation.

Definition 1 *An OQL expression over a set of variables X is recursively defined as follows:*

expr:	*const*	(constants)
	| *var*	(variables from X)
	| *lambda_var*	(inside a select or quantifier)
	| *f*([*expr* [, *expr*]*])	(function application) [c]
	| *expr.method_name*()	(method call)
	| *expr.field_name*	(field selection) [d]
	| [*field_name* = *expr* [, *field_name* = *expr*]*]	(tuple constructor)
	| set([*expr* [, *expr*]*])	(set constructor)
	| bag([*expr* [, *expr*]*])	(bag constructor)
	| list([*expr* [, *expr*]*])	(list constructor)
	| select [distinct] *expr*	
	from *lambda_var* in *expr* [, *lambda_var* in *expr*]*	
	[where *expr*]	(selection)
	| exists *lambda_var* in *expr* : *expr*	(existential quantifier)
	| for all *lambda_var* in *expr* : *expr*	(universal quantifier)

Definition 2 *An OQL query against a database interface is a well-typed OQL expression over the set of named variables of this interface. The answer of an OQL query in a given state of a database is the result of the evaluation of the corresponding expression.*

An OQL *query* is more general than a select-expression in SQL. However, the OQL *select-expression* is a built-in n-ary operator of particular importance, and we expect most of the queries to be expressed in this form. The expressions corresponding to each input collection, the predicate, and the projection of a select-from-where expression, may all be general OQL expressions. As a consequence, OQL allows

[b] In the rest of this paper we will refer to a database schema as an interface.
[c] We allow the application of any built-in or user-defined function, of any arity, in prefix or infix notation.
[d] In order to ease the syntax, we unify the notation for accessing a field of an object (usually noted by an arrow) and accessing a field of a tuple.

navigation (following object identifiers), nested selects, dependent joins, quantified predicates and user-defined functions or methods to appear in all clauses of the select operator.

A variable defined in the from clause of a select-expression or in a quantified expression (forall or exists) is called a lambda variable. The collection-valued expression associated with a lambda variable is called its domain. Compared to a variable in a general programming language, a lambda variable has particular semantics. The value of a lambda variable is restricted to range over the value of the associated set expression and its lifetime is that of the expression (select or quantifier) which defines it. The specific constructors of OQL for expressions, mainly the select and the quantifiers, strongly impacts our query reformulation which is based on pattern-matching.

2.3. Example Interfaces

Local Interface 1	
class	Monitor
(	extension monitors
	key code)
{	attribute integer code;
	attribute integer dimension;
	attribute string manufacturer;
	attribute set(string) compatible;}

Local Interface 2	
class	Disk
(	extension disks
	key code)
{	attribute integer code;
	attribute integer capacity;
	attribute string manufacturer;
	attribute string disk-group;
	attribute set(string) compatible;}

MDBMS Interface or Mediator Interface	
class	Product
(	extension products
	key code)
{	attribute integer code;
	attribute string descriptor;
	attribute string manufacturer;
	attribute string type;
	relationship set(Product) compatible: inverse compatible; }

Fig. 2. Mediator and local interfaces for a simple CIS-MDBMS

Figure 2 is a simple example of a CIS-MDBMS. The MDBMS interface or mediator interface describes products from two local databases. Local interfaces 1 and 2 describe classes `Monitor` and `Disk`, respectively, in each of the local databases. Each `Monitor` (`Disk`) has a `code` (unique key) and manufacturer `attribute`. A monitor has an attribute `dimension` whereas a disk has an attribute `capacity`. These interfaces also describe compatibility of disks and monitors with each other. Each `Disk` is described by an attribute `disk-group` which is used to specify its compatibility with monitors. Attribute `compatible` of class `Monitor` is set-valued and its values range over the *domain of the attribute disk-group* of class `Disk`. Attribute

`compatible` of class `Disk` is also set-valued, but its values range over the *domain of values of attribute manufacturer* of class `Monitor`. The latter is semantic information which can be expressed as integrity constraints. Thus, although compatibility is replicated in the two interfaces, the structures themselves are dissimilar.

Class `Product` in the mediator interface integrates the common attributes of `Monitor` and `Disk` which are `code`, `manufacturer` and `compatible`. A new attribute `type` indicates whether a product is a monitor or a disk, and attribute `descriptor` corresponds to attribute either the `dimension` or the `capacity`, respectively. Further, in the mediator interface, the compatibility information is not based on the disk's disk-group or monitor's manufacturer data. Instead, an attribute `compatible` represents the codes of the compatible products. Thus the mediator interface is a view over the union of the local interfaces. In addition, attribute `compatible` is specified to be an inverse relationship in the mediator interface. Each interface has particular named variables `products`, `monitors`, and `disks`, corresponding to the extensions of the classes `Product`, `Monitor`, and `Disk`, respectively. Each interface is simple, but we are able to include a variety of semantic knowledge, and this is described in the next section.

Consider the following query, expressed against the mediator interface, which will be used to describe reformulation and decomposition in this paper:

Example 1 Select the code and description of all HP disks which are compatible with some 19 inch monitor. The input OQL query is the following:

```
Q1:     select [code:=x.code, descriptor:=x.descriptor]
        from x in products
        where x.type="disks"and x.manufacturer like "HP" and
                              (exists y in x.compatible : y.type="monitor"
                               and y.descriptor like "19 inch")
```

In this OQL query, *products* is a named variable corresponding to the extent of class Product, in the mediator interface, x is a lambda variable ranging over the set-valuated expression *products* and y is a lambda variable ranging over the set-valuated expression *x.compatible*. The variable *products* acts as a global variable, the lifetime of the lambda variable x is the select expression, and the lifetime of the lambda variable y is the expression following the semicolon. The query of example 1 is constructed in the mediator interface. We use the operator *like* to indicate a substring match.

2.4. Architectural Assumptions for the CIS-MDBMS

We follow the architectural approach of a mediator, based on a common data model and query language. Each local database is accessed through a *wrapper* which provides an interface describing the local data in the common data model. The wrapper is responsible for translating queries, expressed in the CIS-MDBMS common query language, into the native query language, (*eg.*, SQL for a relational database). The wrapper is also responsible for reformatting the answers, and resolving semantic conflict of data values between the local databases and the CIS-

MDBMS.

We make several assumptions about the CIS-MDBMS environment. First, the local databases may store data about the same entities from the real world. But, since each of the local databases is developed independently, data corresponding to the same entity may be structured differently in two different databases. The mediator interface may represent an integrated view over these dissimilar local databases, or it may include a view defined over the union of the local interfaces, *eg.*, Product is defined over both Disk and Monitor.

Second, the data in the local databases can contain information about the same objects in the real world (data redundancy). Due to autonomy of the local databases, although data can overlap, this data may not be identical in the different databases, since the two local interfaces may be quite different. In our schema, the data on compatible products is redundant in both local databases, and for simplicity it is stored in a similar structure.

Third, there is no data in the CIS-MDBMS level. As previously noted, the global object types defined in the mediator interface are virtual, and they do not have instances. However, we assume that the results of previously computed queries may be stored and later reused during query processing. The problem of maintaining consistency of these stored queries, while data may be updated in the local databases, is not addressed in this paper. We note that there is extensive previous research on maintaining views, in a DBMS, and this is a similar problem. Further, there are many obvious benefits to partially instantiate and maintain data for complex objects, in the mediator interface, when one considers the possible costs of accessing data from remote servers.

Fourth, for the sake of autonomy, we assume that there is no sharing of object identifiers, either between the mediator interface and the local databases, nor between multiple local databases. An object cannot directly reference another object in a different database, and all such references must be value-based (*eg.*, based on key information).

Fifth, we do not allow methods, *eg.*, written in a general programming language such as C++, in the mediator interface. To support methods in the mediator interface, we would need to either utilize a procedure to translate from a combination of a general programming language and data structures, in the mediator interface, to the local interface. Alternately, we would first instantiate all objects in the mediator interface and then compute the method.

Under these architectural assumptions, query processing in the CIS-MDBMS is as follows. A query is expressed in the CIS-MDBMS common query language, against the mediator interface. The CIS-MDBMS query processor first represents this query in a canonical form, (to be discussed later). Then, it reformulates the canonical input query into a set of queries, (in the canonical form), against the union of the local interfaces. Second, each of these reformulated queries is decomposed into a set of local subqueries, each corresponding to one local database, and a composite query which re-groups the local answers, in the CIS-MDBMS. The local subqueries

are evaluated by the wrapper for the corresponding local database.

Each of the previous steps can lead to several alternatives. The reformulation process can produce several reformulated queries, each of these representing a possible way to compute the expected answer. There are several alternatives to compute the answer, because data may be replicated in the local databases, and because previously computed answers may be re-used. For each reformulated query, several decompositions are possible. A heterogeneous distributed cost model is needed to decide the best reformulated query, and the best decomposition for it. In this paper we only discuss obtaining the choices for the reformulated query, and obtaining the composite query and subqueries, (*i.e.*, creating the search space). We do not address the problem of selecting the best evaluation using a heterogeneous cost model.

A simple plan for the evaluation of a composite query is as follows: each local subquery is independently processed by the corresponding wrapper, which translates it into the local query language. The local subqueries may be executed in parallel in the local databases. After executing all the local subqueries, the composite query is computed at the CIS-MDBMS, to combine the local results. Alternately, a more sophisticated evaluation strategy for the composite query and the local subqueries can be determined, using semi-join type optimization techniques, etc. However, query reformulation and query decomposition are independent of the actual evaluation plan.

The task of query reformulation in the CIS-MDBMS architecture can be stated as follows: given a well-typed input query against the mediator interface, and some semantic knowledge describing the state of the CIS-MDBMS environment, the reformulation task produces a set of well-typed queries against the union of the local interfaces, which are equivalent to the input query.

In a single database, *query equivalence* is defined as the property that two queries evaluate to the same value (produce the same results), in a particular state of this database. In a CIS-MDBMS environment, query equivalence is more complicated, since the input and the output queries are expressed against different interfaces, and are evaluated on different databases. Further, the mediator interface does not contain any data. Thus, we cannot determine the equivalence of the input and the reformulated queries by simply executing them. Our solution is to use semantic knowledge, describing the data in the CIS-MDBMS environment, during the reformulation process. Semantic knowledge is expressed using rewrite rules stored in the multidatabase catalog, and are assumed to be valid, for example, they *define* the mapping from the mediator interface to the local interfaces. We use a pattern-match based rewrite algorithm, using these valid rewrite rules, to produce the set of reformulated queries, which are then equivalent to the input query.

2.5. Examples of CIS-MDBMS Query Reformulation and Decomposition

We now present reformulated queries for the input CIS-MDBMS query of Example 1. When a reformulated query is to be evaluated against more than one local database, then we represent the reformulated query as a nested query. The

inner select-from-where OQL expressions, (or select-expressions, for short), are the subqueries, each constructed against a local interface. The outer select-expression is constructed in the mediator interface, and is the composite query, and represents the regrouping of the final results at the CIS-MDBMS level. We do not present the actual queries in the local query language, *eg.*, SQL queries, that are evaluated by the wrapper.

Both databases must be accessed to identify HP disks and 19 inch monitors. However, there is a constraint in the local interfaces, about replication of the compatible products in the two databases, as well as other relevant integrity constraints. Thus, there are several ways in which we can evaluate this query.

One possibility is to select the disk-group of products (disks) that are compatible with 19 inch monitors, from one database, and select products that are HP disks from the other database. Then, based on the value of the disk-group attribute of the HP disks, we can determine if these are indeed compatible products, and thus, restrict the answer to compatible HP disks. The first nested subquery retrieves the set of disk-group values of products that are compatible with some 19 inch monitor, the set F1, from local database #1. The second nested subquery retrieves the code F2, descriptor (locally capacity), and the value of the disk-group attribute F4, of all HP disks, from local database #2. The composite query, (evaluated at the CIS-MDBMS), restricts the final answer to the code and descriptor of all the HP disks (retrieved in the second nested subquery), whose disk-group (value of F2) occurs in the result of the first nested subquery (in the set F1).

```
A1:     select distinct [code:=y.F2, descriptor:=y.F3]
        from x in (select distinct [F1:=z.compatible]          /* local database #1 */
                   from z in monitors
                   where z.dimension like "19 inch"),
             y in (select distinct [F2:=a.code, F3:=a.capacity, F4:=a.disk-group]
                   from a in disks                           /* local database #2 */
                   where a.manufacturer like "HP")
        where y.F4 in x.F1
```

Due to replication of compatible information about the products, a second possibility is to select the manufacturer of all 19 inch monitors, from one database, and retrieve HP disks and the set of manufacturers of their compatible products from the other. Then, we restrict our answer to those HP disks, where the set of compatible (manufacturer) values include the manufacturer of a 19 inch monitor. The first nested subquery retrieves the manufacturer, (F1), of all 19 inch monitors, from local database #1. The second nested subquery retrieves the the code, descriptor (locally capacity), and the set of values of attribute compatible (manufacturer values), the set F4, for each HP disk, from local database #2. The CIS-MDBMS composite query restricts the answer to those HP disks in the second subquery, whose set of compatible manufacturers (set F4) includes a manufacturer that occurs in the result of the first nested subquery (a value of F1).

```
A2:     select distinct [code:=y.F2, descriptor:=y.F3]
        from x in (select distinct [F1:=a.manufacturer]        /* local database #1 */
                   from a in monitors
```

```
            where a.dimension like "19 inch"),
        y in (select [F2:=c.code, F3:=c.capacity, F4:=c.compatible]
            from c in disks                          /* local database #2 */
            where c.manufacturer like "HP")
    where x.F1 in y.F4
```

Further, suppose that the result for the following query Q2, that selects the code of all products that are compatible with some 19 inch monitors, has already been computed, and the result is stored at the CIS-MDBMS level. It is possible to compute Q1 using Q2.

```
Q2:   select x.code
      from x in Product
      where exists y in x.compatible:
            (y.type="monitor" and y.descriptor like "19 inch")
```

Query Q2 has already computed the code of all products that are compatible with some 19 inch monitors, and this is useful for Q1. However, Q2 does not include the descriptor of such compatible products, to determine if they are HP disks, and this information is in database #2. The first nested subquery retrieves the codes already stored in Q2. A second nested subquery, executed on local database #2, retrieves the code, (set F1) and descriptor, of all HP disks. The composite CIS-MDBMS query restricts the final answer to the code and descriptor of all the disks in the second nested subquery, whose code (value of F1) occurs in the result of the first nested subquery.

```
A3:   select distinct [code:=y.F1, descriptor:=y.F2]
      from x in Q2,                          /* previous query */
          y in (select distinct [F1:=d.code, F2:=d.capacity
              from d in disks                /* local database #2 */
              where d.manufacturer like "HP")
      where x=y.F1
```

To summarize, A1 obtains the disk-group values of products compatible with 19 inch monitors from database #1, and determines if they match the disk-group value of HP disks; A2 obtains manufacturers of products compatible with HP disks from database #2, and determines if they match manufacturers of 19 inch monitors; A3 gets codes of products compatible with 19 inch monitors from query Q2, and then determines if they match the code of HP disks.

3. Semantic Knowledge in the CIS-MDBMS

The semantic knowledge in the CIS-MDBMS catalog describes the properties of data stored in the local databases and defines the "data"[e] in the mediator interface. This knowledge includes the mappings between the mediator interface and the local interfaces, the integrity constraints satisfied in the mediator interface, and information about data replication in local databases. Information about the previously computed queries, whose answers are stored at the CIS-MDBMS level, and are similar to views, are also stored temporarily in the CIS-MDBMS catalog.

[e] Recall that the CIS-MDBMS does not contain actual instances for object types in the mediator interface.

Research in schema integration techniques, as described in [2,14,20], all provide different sources for semantic knowledge. In our architecture, this knowledge is uniformly expressed using the OQL query language, which is expressive enough to represent the variety of knowledge involved. Using a single representation also allows us to provide a uniform basis for query reformulation using this semantic knowledge.

3.1. Integrity Constraints

There are several categories of integrity constraints, including integrity constraints (which are satisfied) in each of the local databases, integrity constraints across several local databases which describe data replication, and integrity constraints (satisfied by the view) in the mediator interface. All provide alternatives for query reformulation. Thus, in the previous example, knowledge about data replication of compatible products, in the local databases, was used to choose different ways to compute a query in these databases. We represent these integrity constraints in a declarative manner, as a set of *assertions*, defined as follows:

Definition 3 *An assertion is a first order formula of the form:*

[forall [< *variable_declaration* >]*] $E_1 \sim E_2$

where a variable declaration is either:

<*var_name*> of type <*type_expression*> (1)

or

<*var_name*> of type <*type_expression*> in <*set_expression*> (2)

E_1 and E_2 are well-formed OQL expressions and are allowed to use the variables from X, as well as the named variables in the global interface and the local interface. An assertion is denoted (X, E_1, E_2), where X is the set of quantified variables.

We allow variable quantification over a type, as in (1), or over a particular collection, as in (2), *eg.*, x in monitors, where monitors is the extent of user-defined type Monitor. In (2), the variables are restricted, and the associated set-expression is the restriction domain. The type of a restricted variable can be induced from the type of the associated set-expression, and so the type is omitted.

An assertion is *valid* in a particular state of the CIS-MDBMS environment, if for each correct instantiation of the variables from X, the two expressions evaluate to the same value. An *instantiation* for the variables X is a mapping from each variable x in X to a value in the domain [f]of the type of x. An instantiation is *correct* if the value associated with every restricted variable belongs to the corresponding restriction domain.

We now give examples of particular assertions. When both expressions E_1 and E_2 of an assertion are constructed in the same interface, the assertion expresses an integrity constraint verified in the corresponding database. If E_1 is constructed in one local interface and E_2 is constructed in another, then the assertion describes data replication in the two local databases.

[f]The domain of a type is the set of possible values.

Example 2 *Integrity constraints in a local interface.*

- The attribute code is a key for the object type Monitor in the interface #1:

  for all x: Monitor and y: Monitor x.code=y.code ∼ x=y

Example 3 *Integrity constraints in the mediator interface.*

- The relationship compatible defined the object type Product is its inverse relationship:

  for all x:Product and y:Product x in y.compatible ∼ y in x.compatible

- A product is either a disk or a monitor:

  products ∼ union((select x from x in products where x.type="disk"),
  (select y from y in products where y.type="monitor"))

- A monitor is compatible only with a disk, and vice versa:

  for all x:Product and y:Product
  x.type="monitor" and y in x.compatible
  ∼ y.type="disk" and x in y.compatible

Example 4 *Integrity Constraint across several local interfaces.* Here both E_1 and E_2 may be constructed in several local interfaces. Data replication in the local databases is a particular case of such an integrity constraint. This is represented as the equivalence of two queries, evaluated in two different local databases, and which give the same answer. So, the same information is stored in both local databases, but it may be structured in different ways in each database.

- The compatibility information for disks and monitors is stored in both local databases:

  for all x: Monitor and y: Disk
  y.disk-group in x.compatible ∼ x.manufacturer in y.compatible

Since OQL is an expressive query language, we are able to express and utilize a rich set of integrity constraints, wrt the complex data structures of the common object model, in comparison to knowledge used in [3,7,18].

During reformulation, the assertions are used as rewrite rules. Given a set W of assertions, describing the CIS-MDBMS environment, then, the set R(W) of rewrite rules induced from W is as follows:

$$R(W) = \{(X, E_1 \Rightarrow E_2) \mid (X, E_1, E_2) \in W\} \cup \{(X, E_2 \Rightarrow E_1) \mid (X, E_1, E_2) \in W\} \quad (1)$$

The correctness of the rewriting, or that the input query is equivalent to the output query, while using a particular rewrite rule, is ensured by the validity of the corresponding assertion.

We address verifying the validity of assertions as follows: when the two queries of an assertion are constructed in the same local interface, or constructed in more than one local interface, (representing an integrity constraint across several local databases), then these equivalences could be verified by evaluating particular queries

in the local databases. We assume that each local database is responsible for maintaining local integrity constraints. Further, if a rewrite rule expresses an integrity constraint in the mediator interface, then it cannot be verified directly, since the mediator interface does not contain data. However, it is possible to verify a particular integrity constraint in the mediator interface, if we assume that all other assertions are valid in the mediator interface. We would reformulate the queries in the assertion that is to be verified, and obtain corresponding queries to be verified in the local interface(s).

3.2. Mapping Between the Mediator Interface and the Local Interfaces

The mapping from the mediator interface to the local interface is semantic knowledge that is provided by the user. We use *rewrite rules* to specify this mapping. These rules define the CIS-MDBMS environment and are assumed to be valid. They will not be verified.

Definition 4 *A mapping rule is a rewrite rule of the form:*

$$Q_1 \Rightarrow Q_2$$

where Q_1 is a well-formed OQL query in the mediator interface, and Q_2 is a well-formed OQL query in the union of the local interfaces and the mediator interface.

The query Q_1, against the mediator interface, can be computed by evaluating the query Q_2, in the local and mediator interfaces.

Example 5 Information on products of type disk are obtained from database #2.

```
select [code:=x.code,                ⇒    select[code:=y.code,
        descriptor:=x.descriptor,                descriptor:=y.capacity,
        manufacturer:=x.manufacturer,            manufacturer:=y.manufacturer,
        compatible:=x.compatible]                compatible:=
                                                     (select z
                                                      from z in products
                                                      where exists t in monitors :
                                                          (t.code=z.code and
                                                              t.manufacturer in y.compatible))
from x in products                       from y in disks
where x.type="disk"
```

Our approach to specifying these mappings is similar to the concept of virtual classes, proposed in [1]. We extend their approach to represent object references, or a relationship in the ODMG data model. The general form of a mapping rule is that Q_1 and Q_2 are queries. Thus, Q_1 may be a view definition in the mediator interface. Furthermore, Q_2 may involve the mediator interface in addition to the local interfaces. If the query Q_1 projects only values, then query Q_2 can be constructed in the local interfaces only.

However, Q_1 may project a relationship which refers to another object in the mediator interface. For example, the set x.compatible represents the set of products in the mediator interface compatible with product x. Since object identifiers are not shared between the CIS-MDBMS and the local databases, we cannot explicitly map an object in the mediator interface with some corresponding entity in the local

databases. To map this relationship between objects in the mediator interface, the mapping may use a key constraint in some local database to obtain the values for the fields of an object in the mediator interface. In the example, product z is linked to a monitor t through a key constraint. Thus, in this case, the query Q_2 is defined in both the mediator interface and the local interface. Using the common object model and OQL, we are able to express complex mapping rules about complex data structures, *eg.*, a product in the mediator interface must have at least 3 compatible products, *etc.* In this paper, we do not use examples of such complex constraints.

3.3. Comparison with Other Approaches

In a later section, we compare our query reformulation methodology with other approaches. Here, we compare the semantic knowledge of our CIS-MDBMS, based on the ODMG/OQL common model and language, with knowledge that is captured in other mediator architectures.

The research on schema integration is exemplified by systems such as Pegasus[2], UniSQL[14] and SIMS[3]. In Pegasus[2] and UniSQL[14], conflicts among entities of multiple local databases were resolved within the entities of a unified global schema, based on the object data model. Semantic knowledge, in the form of mapping rules, expressed an entity in the global schema as a view over the entities in the local databases. In Pegasus[2], special reconciliation functions were able to mediate discrepancies among data in the multiple local databases. Similarly, in SIMS[3], unify the multiple local databases. However, each entity in the local database had to correspond to an entity in the LOOM knowledge base, and there was no capability of defining a view over the local databases.

In the Information Manifold project, [15,18], a *world-view* conceptually unifies information from *site relations* in multiple local databases. A concept in the world-view can be defined as a view (conjunctive Datalog query) over the union of the site relations, and vice versa. Further, semantic knowledge includes *completeness information*, *i.e.*, a site relation may have complete information about a world-view concept. Finally, they also represent query forms that describe queries that may be executed at local databases.

In our CIS-MDBMS mediator architecture, all semantic knowledge is uniformly represented as rewrite rules $Q_1 \Rightarrow Q_2$, where both Q_1 and Q_2 are OQL queries, possibly in the union of the local interfaces and the mediator interfaces. We are able to express integrity constraints in the local interfaces and the mediator interfaces, as well as data replication in the local databases. This knowledge could not be expressed by the previous approaches, which only support view definition. Thus, we are able to explore several alternatives for reformulation that are not considered by other approaches. For example, in the mediator interface, a product has a relationship *compatible* which is a reference to a set of products, also in the mediator interface. None of the other approaches would support a mapping rule which described products as a view over products (in the mediator interface), and disks and monitors (in the two local interfaces).

A drawback of many mediator architectures is that, as new data sources and wrappers are added, we may need to modify the mediator interface, and we always need to modify the mapping rules or semantic knowledge. An architectural alternative is to add higher order capability to the wrappers so that they are able to handle dynamic sources. This is a subject for future research.

4. Methodology for CIS-MDBMS Query Reformulation

This section presents a methodology for CIS-MDBMS query reformulation. We first present an overview of query reformulation in general, emphasizing the difficulty presented by the expressiveness and complexity of OQL queries. Then, we present the CIS-MDBMS query reformulation algorithm, which uses syntactic rewriting, and semantic rewriting based on pattern-matching.

4.1. Overview

The objective of query reformulation is to transform an input query into equivalent queries, each corresponding to an alternative way of computing the result. We previously defined the *equivalence* of two queries, in a particular database state, as the condition that they evaluate to the same result. But this general condition is difficult to prove, without actually executing the queries. Given some semantic knowledge, we say that two queries are *semantically equivalent* if they evaluate to the same result, in each particular database state which is described by the semantic knowledge.

We say that two expressions are *logically equivalent*, if they evaluate to the same result, in *all* states of the database. However, an OQL expression can be written in several syntactically dissimilar ways. During query reformulation using pattern matching, it is important to identify these logically equivalent expressions. Otherwise, a pattern matching procedure may determine that two expressions Q_1 and Q_2 do not match, although there may be an expression Q_3, which is logically equivalent to Q_2, and which matches Q_1. The generality of OQL queries increases the number of dissimilar ways in which an expression can be written. For example, a select-expression can also be written as a nested select-expression, a dependent join can also be written as a logically equivalent independent join, a select-expression using navigation can be rewritten using an explicit join, *etc.* Such problems do not usually arise with Datalog-like queries. We presume that determining the logical equivalence of OQL expressions is undecidable. Our solution is to require all select-expressions to be placed in a canonical form, to reduce the possibility of not identifying a logical equivalence.

Another problem occurs while matching OQL select-expressions. Even if two select-expressions, Q_1 and Q_2, in canonical form, are not found to be a match, it may be possible to rewrite Q_1 as a nested query Q_3, which contains Q_2 as a subexpression. Thus, the rewriting algorithm must be able to detect that a select-expression is a subexpression of another select-expression. Consequently, rewriting

of OQL queries is not straightforward.

In this section, we present our methodology for CIS-MDBMS query reformulation. For an input query expressed in the mediator interface, and some semantic knowledge defining the CIS-MDBMS, in the form of rewrite rules, we produce a set of semantically equivalent queries, in the canonical form, expressed in the union of the local interfaces. The query reformulation algorithm consists of a *syntactic rewriting* to place the input query in the canonical form, followed by a *semantic rewriting* using the rewrite rules. A key aspect of semantic rewriting is the pattern matching algorithm for OQL expressions. The algorithm is general enough to solve the problem of answering queries using a materialized view, and this increases the alternatives explored during query reformulation.

The declarative nature of the rewrite rules eases the specification of semantic knowledge in the CIS-MDBMS environment. Since each rule describes an independent transformation, it would be simple to associate a cost benefit with a particular transformation, and use this as a first step in selecting the best reformulation. But there is a potential trade-off between ease of expressibility of the rule language and efficiency of the rewriting. The complexity of the rewrite algorithm, given an input query and a rewrite rule is, in the worst case, exponential in the size (number of collections in the from clause) of the input query. Given the fact that the size of OQL queries is typically limited to a few collections, the algorithm is quite efficient in practice.

4.2. Syntactic Query Rewriting

Syntactic query rewriting produces a canonical form representation, for an OQL select-subexpression in the input query, or a select-subexpression of an expression in the rewrite rules. The canonical form representation is based on a strongly typed object algebra.

Specifying such a canonical form is an extremely hard problem. In order to solve all syntactical dissimilarities, one must integrate **all** the algebraic properties of the built-in operators and user-defined methods, (*eg.*, the commutativity of the addition operator, the associativity of the intersection operator, the distributivity of the select over the union, the neutral element of the empty set for the union operator, *etc.*). Our solution is to find a *relaxed canonical form*. This relaxed canonical form may not detect *all* syntactical dissimilarities, but will reduce the possibility of not identifying a logical equivalence. The effectiveness of this relaxed canonical form would depend on the completeness of the compiler, with regard to all the possible algebraic properties of the operators and methods. We define a relaxed canonical form for an OQL select-expression satisfying the following properties:

- the subexpression corresponding to the predicate in the where clause is in conjunctive normal form;
- existential quantifiers in the predicate of the where clause are eliminated, whenever possible;

- there are no nested select-expressions in the from clause;
- particular cases of nested select-expressions occurring in the where clause are eliminated; examples are testing membership in the result of a nested select-expression, or testing that the result of a nested select-expression is empty;
- any dependencies that may exist between different collections in the from clause are eliminated, whenever possible;
- navigation within complex objects (the so-called functional joins) is transformed into explicit joins whenever possible; The criterion for elimination is that the corresponding class extent is in the interface.
- if it is possible to deduce, based on the key, that the result cannot contain duplicates, then the **distinct** clause is explicitly introduced in the select-expression. The existence of duplicates is important when matching a query with a view. Duplicates are also important when eliminating existential quantifiers, or when manipulating expressions involving object references.

The compiler used during reformulation includes a built-in rewrite rule for each of the previous transformations, and an OQL select-expression is converted to its canonical form by applying these syntactic rewrite rules, in any order, until saturation. Appendix A gives examples of the most important syntactic rewrite rules.

A set of syntactic rules may not be complete, *i.e.*, they may not capture all the algebraic properties of the OQL operators. However, the above properties enable us to identify almost all of the commonly used syntactic variations of OQL, so that two logically equivalent queries will be identical. The set of the syntactic rewrite rules used by the compiler is not fixed, and it can be easily extended by including new rewrite rules.

4.3. Semantic Query Rewriting

Given an OQL query in its canonical form, semantic query rewriting produces semantically equivalent queries by applying the rewrite rules. It uses a pattern matching rewriting algorithm which, for a given OQL expression and a rewrite rule, produces a set of semantically equivalent OQL expressions, (in canonical form).

A *substitution* for the set of variables X=$\{v_1, \cdots, v_n\}$ is a finite ordered set θ of the form $\{v_1/e_1, \cdots, v_n/e_n\}$, where each e_i is an OQL expression distinct from v_i, but with the same type as v_i. The substitution θ is *correct* if for each restricted variable v_i the corresponding expression e_1 is a lambda variable verifying the property that the domain of e_1 is exactly the domain restriction of v_i. Let $\theta = \{v_1/e_1, \cdots, v_n/e_n\}$ be a substitution, and E be an OQL expression. Consider the expressions $E_0, E_1, \cdots, E_n$, where E_0=E and E_i is obtained from E_{i-1} by replacing each occurrence of variable v_i in E_{i-1} by e_i. E_n is called *the instance of E by the substitution* θ, and it is denoted by Eθ.

An OQL expression E_1 *matches* another expression E_2, if there exists a substitution, θ, such that $E_1\theta$=E_2. Substitution is the basis for derivation of OQL expressions which we define as follows:

Definition 5 *Given two OQL expressions E and E′ , and a rewrite rule r=(X, L ⇒ R), E′ is derived from E, by applying rule r, if there exists a variable y_0, distinct from X, and an OQL expression E_0, such that there is at most one occurrence of y_0 in E_0, and a correct substitution θ for the variables in X, such that $((E_0\theta_1)\ \theta)$ is logically equivalent to E, and $((E_0\theta_2)\ \theta)$ is logically equivalent to E′, where $\theta_1 = \{y_0/L\}$ and $\theta_2 = \{y_0/R\}$.*

Given an input OQL expression E, and a rewrite rule r=(X, L ⇒ R), the computation of the set of OQL expressions which may be derived from E by applying the rule r works as follows: We first identify a subexpression E_1 of the expression E which corresponds to an occurrence of the left-hand side, L, r, and substitute the subexpression E_1 with the right-hand side, R, of r. Further, we must guarantee that the application of r is sound, (*i.e.*, the resulting expression does not contain any variable from X of r). Depending on whether L of r is a select-expression or not, we have two different cases of the algorithm.

The case where L *is not* a select-expression is simple and we describe it as follows: We use a procedure **Match()** which, given two expressions e_1 and e_2 which are not select-expressions and have the same type, either fails, or succeeds and returns a substitution θ, such that $e_1\theta$=e_2. If Match succeeds for input expressions L and E′, with substitution θ, then subexpression E′ in E is replaced with Rθ. The resulting new instance of E, in its canonical form, is added to the result set $\mathcal{R}$. Match uses a classic pattern matching technique. The fact that OQL expressions are typed can be exploited to increase the efficiency of pattern matching. All the subexpressions of E having the same type as L are selected first, in a single traversal of E.

```
procedure Rewrite(E, r=(X, L, R))      /*the case when L is a select-expression*/
{R={}
 foreach select-subexpression Q of E
  {if FindSubquery(L,Q) succeeds with substitution θ and expression Q′ and
   if Rθ does not contain any variable from X    /* there is a sound application of r */
   then replace Lθ in Q′ by Rθ, replace Q in E by Q′,
        convert the result to the canonical form and add it to R}
 return R
}
```

Figure 3: Rewriting algorithm.

The case where L (of r) *is* a select-expression is more involved and requires significant extension of the pattern matching algorithm. The rewriting algorithm for this case is depicted in Figure 3. Consider a select-subexpression Q (of E) and L the left-hand side of the rule r. Even if Q and L are not matching expressions, it is possible to rewrite Q as a nested expression, logically equivalent to Q, which contains L as a subexpression. The algorithm uses the procedure **FindSubquery**.

Given two select-expressions L and Q, **FindSubquery** either fails, or if it succeeds, it returns a query Q′, logically equivalent to Q, and a substitution θ; further, Q′ contains Lθ as a subexpression. If FindSubquery succeeds for inputs L and Q, and returns Q′ and a substitution θ, then Lθ is replaced in Q′ with Rθ and Q in

E is replaced with the new instance of Q′, where Q′ is logically equivalent to Q. Finally, this new instance, in its canonical form, is added to the result set $\mathcal{R}$.

Procedure **FindSubquery** is used extensively with the mapping rules, which are usually expressed as select-expressions. It is also used to reformulate a query using the result of a previously computed query or a *materialized view*. Consider the following: Given a query Q, and a stored view V, defined by the query Q', we want to determine if it is possible to rewrite the query Q as a nested query Q'', equivalent to Q, which contains the view, Q', as a subexpression. If successful, the subquery Q' of Q'' can be replaced by the view V, and the query Q'', equivalent to query Q, can be computed using this view V. In the case of a CIS-MDBMS environment, a previously computed query can be considered as a stored view V. Thus, the algorithm **FindSubquery** is able to solve the problem of answering queries using materialized views. As mentioned earlier, we do not address maintaining materialized views, but note that there are significant performance benefits, and so it is an important consideration during query reformulation.

We now describe Procedure **FindSubquery**. See Appendix B for the complete algorithm. We note that the details of this algorithm are not needed to understand the remainder of this paper. Suppose that the input select expressions, Q′ and Q, (which correspond to L and Q in Figure 3), have the following form:

Q′= select $proj_1$
from x_{11} in C_{11}, $\cdots$, x_{1n} in C_{1n}
where p_{11} and p_{12} and $\cdots$ and p_{1q}

Q= select $proj_2$
from x_{21} in C_{21}, $\cdots$, x_{2m} in C_{2m}
where p_{21} and p_{22} and $\cdots$ and p_{2r}

FindSubquery works as follows: First, each query C_{1i} corresponding to an input collection in Q′ is matched with some collection C_{2j_i} in Q. If the match succeeds, the resulting substitution is added to the global substitution, together with the binding x_{1i}/x_{2j_i} of the corresponding variables. If for some collection of Q′, none of the collections of Q are found to match, then the algorithm fails. However, it is possible that some of the collections of Q do not match any of the collections of Q′. These collections, together with the corresponding variables, cannot be eliminated and must appear in the from clause of the final query Q″. Next, each conjunction p_{1i} in the predicate of Q′ is matched with some conjunction p_{2j_i} in Q. If for some conjunction in the predicate of Q′, none of the conjunctions of Q match, then the algorithm fails. However, it is possible that some of the conjunctions of Q do not match with any of the conjunctions of Q′. These conjunctions cannot be eliminated and must appear in the where clause of the final query Q″.

The resulting query Q″ is produced from Q as follows. A new collection, corresponding to query Q′, is added to the from clause of Q″. Some collections in the from clause of Q need not appear in Q″, since the corresponding conditions and projections are already included in Q′, but some other collections cannot be eliminated and must appear in Q″. There are two criteria for a collection to appear in Q″: either the corresponding variable appears in some unmatched predicates or in the projection, or the corresponding variable appears in the expression of another

collection which must appear in Q''.[g] Thus, some collections may appear in both Q' and also appear in the from clause of Q''. In this case, these collections appear twice in the from clause of Q'', with two different variables ranging over them. Thus, an additional predicate must be added to the where clause of Q'', in order to link these variables. This link assures the equivalence of the two queries Q and Q'', if and only if the projection of Q' is a superset of a key, *i.e.*, if the $proj_1$ uniquely identifies the tuple $x_{11}, \cdots, x_{1n}$.

4.4. The CIS-MDBMS Query Reformulation Algorithm

Given an input query in the mediator interface, and a set of rewrite rules defining the CIS-MDBMS, the CIS-MDBMS reformulation algorithm produces a set of equivalent queries in the union of the local interfaces. The algorithm is shown in Figure 4.

```
procedure Reformulate(Q, R)
{ S={Q}        /*Q is in the canonical form*/
  δ={Q}
  repeat
      new = ∅
      for each q ∈ δ and r ∈R
            {new = new∪Rewrite(q,r)
            δ = new − S
            S = S ∪ δ}
  until δ = ∅
  eliminate all queries from S with occurrences of the named variables of the
            mediator interface
  return S
}
```

Figure 4: Reformulation algorithm.

The rewrite rules (assertions and mapping rules), are applied uniformly, in any order. The control of this algorithm is similar to semi-naive evaluation, in deductive databases [32]. At each iteration, new queries obtained in the previous iteration are used to generate new queries, until no new more queries can be obtained.

It is possible that the set of reformulated queries is empty or infinite. If there is insufficient knowledge for reformulation (in the catalog), (*i.e.*, the set of reformulated queries is empty), then the CIS-MDBMS query processor would simply reject the query. If the rewrite rules are such that the set of reformulated queries is not finite, then the algorithm does not terminate. However, the execution of the algorithm can be explicitly terminated when one of the following conditions are met: (i) when at least one reformulated query is obtained; (ii) when at least one reformulated query having an acceptable cost is obtained; (iii) when an upper bound on execution time is reached.

[g] This is possible because of dependent joins.

5. Query Decomposition in the CIS-MDBMS

Although the focus and contribution of this paper is CIS-MDBMS query reformulation, we also describe query decomposition in the CIS-MDBMS for completeness. Query decomposition in the CIS-MDBMS takes as input a query Q, expressed against the union of the local interfaces, in the canonical form, and produces an equivalent *decomposed query*, or a set of local subqueries $Q_1, \ldots, Q_n$, to be sent to the wrappers for execution on the local databases, and a composite query, to group the local results at the CIS-MDBMS level. Each subquery is an expression constructed against the corresponding local interface. The input collections of the composing query are those produced by the subqueries $Q_1, \ldots, Q_n$. There can be more than one subquery for each local database. The algorithm is in Appendix C.

Decomposition proceeds in two steps. First, a partition of the set of expressions corresponding to the collections in the from clause of the input select-expression is found. Second, the local queries and the composite query are constructed, based on this partitioning. The decomposition partitioning must satisfy some correctness criteria, which are defined as follows.

Definition 6 *Given a select-expression Q constructed over the union of a set of interfaces*[h] *$I_1, \cdots, I_l$, in the canonical form:*

$$\begin{array}{l} \text{select } proj \\ \text{from } x_1 \text{ in } C_1, \cdots, x_n \text{ in } C_m \qquad\qquad (Q) \\ \text{where } p_1 \text{ and } p_2 \text{ and } \cdots \text{ and } p_q \end{array}$$

A decomposition partitioning for Q is a partitioning $(\mathcal{P}_i)_{1 \leq i \leq n}$ of the set $\{C_1, \cdots, C_m\}$ satisfying the following conditions:

(i) **locality:** *all the expressions C_j in one subset $\mathcal{P}_i$ are constructed in the same local interface I_i;*

(ii) **dependency-free:**[i] *if there exists some expression $C_j \in \mathcal{P}_i$ such that the expression C_j uses a lambda variable x_k associated with some other expression C_k, then $C_k \in \mathcal{P}_i$.*

The locality condition ensures that all the expressions in one partition are in the same interface so that the corresponding subquery is constructed over only one interface. The second condition ensures that in the case of a dependency between two collections C_k and C_j, then they are both in the same partition. When two dependent collections belong to two different partitions, the subqueries corresponding to these partitions would also be dependent and could not be executed independently.

Given a select-expression, Q, and a decomposition partitioning $\mathcal{P}_1, \cdots, \mathcal{P}_n$, the algorithm **Decomp**, of Appendix C, produces a set of select-expressions $\mathrm{Q}_1, \cdots, \mathrm{Q}_n$, or local subqueries, and a composite select-expression, Q′. The input collections for each Q_i are the expressions in the corresponding subset $\mathcal{P}_i$, and the input collections for the composite query Q′ are the select-expressions $\mathrm{Q}_1, \cdots, \mathrm{Q}_n$. The composite

[h] The union of interfaces is still an interface.

[i] This is possible because of the dependent joins.

query Q′ is logically equivalent to the input query (select-expression) Q. In the next section, we describe query reformulation and decomposition, using an example query.

There can be more than one possible decomposition partitioning. We can choose a "good" decomposition partitioning based on some heuristics. The obvious criterion for a partitioning to be efficient is that it must minimize the size of the results of the local subqueries. The next section includes an example of reformulation and decomposition.

6. Validation

The algorithms described in this paper have been validated within the Flora compiler prototype [12], which has been operational at INRIA since June 1994 (IDEA Project Review). The Flora compiler supports the ODMG data model and query language, and currently uses the O2 DBMS [4] for local database management. The prototype is implemented in C++. An important goal in designing the Flora compiler was to achieve a trade-off between extensibility and efficiency. In order to be able to take advantage of semantic knowledge described as rewrite rules, the Flora compiler implements the extended pattern matching algorithm described in this paper. For efficient rule-based rewriting, all the syntactic transformation rules used by the compiler, to obtain canonical form OQL expressions, are coded in C++.

The architecture of the compiler is modular, as suggested in [21]. Each module takes as input an OQL expression, and produces a set of equivalent OQL expressions, based on the knowledge specific to the module and using a specific control strategy. Each module has a goal which characterizes the resulting OQL expressions. This modular architecture allows decentralization of knowledge and finer control of the overall compiler.

Figure 5 gives a simplified view of the Flora compiler architecture, with emphasis on the major modules described in this paper. We do not show the modules for type checking, syntactic rewriting (to put an OQL expression in canonical form), and code generation (*i.e.* the wrapper from OQL to O2C code). The query processor receives the input OQL query expressed on the mediator interface, and produces the best query execution plan (QEP) by calling the other modules. The major modules for CIS-MDBMS query processing perform reformulation, decomposition and optimization; this last step is not emphasized in this paper. All the modules access the CIS-MDBMS catalog which stores meta-data information, (interface definition and semantic knowledge), cost-based information, (statistics regarding the local databases and cost functions), and information about previously computed queries (views).

Reformulation and decomposition work as previously described. Query optimization takes as input a decomposed query and produces the corresponding QEP together with a cost descriptor using a cost model. The query processor controls the optimization process by submitting the decomposed queries to the optimization module, and selection the QEP with least cost. The query processor may also use

heuristics to restrict the search space of reformulated queries and corresponding decomposed queries.

An OQL expression is represented as a direct acyclic graph. The internal representation is general enough to support the input OQL queries, as well as the reformulated queries, the decomposed queries and the optimized query execution plans, enabling seamless manipulation.

We now show the trace of a full derivation from an input query to one of the reformulated queries obtained by the running the input query of Example 1, and then the decomposition. At each step, several re-writings are investigated, but, for purpose of simplicity, we do not follow all the alternatives searched by the rewriting algorithm.

```
Q1:   select [code:=x.code, descriptor:=x.descriptor]
      from x in products
      where x.type="disks"and x.manufacturer like "HP" and
        (exists y in x.compatible : y.type="monitor" and y.descriptor like "19 inch")
```

The first step of query reformulation is syntactic rewriting. The canonical form of the query is:

```
Q2:   select distinct [code:=x.code, descriptor:=x.descriptor]
      from x in products, y in products
      where x.type="disks"and x.manufacturer like "HP" and y in x.compatible and
            y.type="monitor" and y.descriptor like "19 inch"
```

Next, the mapping rule of Example 5 is applied. The mapping rule is the following. Note that the left hand side and the right hand side of the rule are also in the canonical form.

```
M1: select distinct                    ⇒   select distinct
      [code:=x.code,                           [code:=x.code,
       descriptor:=x.descriptor,                descriptor:=x.capacity,
       manufacturer:=x.manufacturer,            manufacturer:=x.manufacturer,
       compatible:=x.compatible]                compatible:=
                                                    (select distinct y
                                                     from y in products, z in monitors
                                                     where z.code=y.code and
                                                        z.manufacturer in x.compatible)
      from x in products                       from x in disks
      where x.type="disk"
```

The first step in applying the rule is to force the left-hand side expression to be a subexpression of the query Q2, using the **FindSubquery** algorithm. The procedure succeeds and returns the substitution {} and the following equivalent query for Q2:

```
Q3:  select distinct [code:=x0.code, descriptor:=x0.descriptor]
     from x0 in (select distinct [code:=a.code,
                                  descriptor:=a.descriptor,
                                  manufacturer:=a.manufacturer,
                                  compatible:=a.compatible]
                 from a in products
                 where a.type="disk"), y in products
     wherex0.manufacturer like "HP" and y in x0.compatible
           and y.type="monitor" and y.descriptor like "19 inch"
```

The inner select expression, which corresponds to the left-hand side expression of the mapping rule M1, is replaced by the right-hand side of the rule, yielding:

```
Q4:     select distinct [code:=x0.code, descriptor:=x0.descriptor]
        from x0 in (select distinct
                            [code:=a.code,
                             descriptor:=a.capacity,
                             manufacturer:=a.manufacturer,
                             compatible:=(select distinct b
                                          from b in products, z in monitors
                                          wherez.code=b.code and
                                                z.manufacturer in a.compatible)]
                           from a in disks), y in products
        wherex0.manufacturer like "HP" and y in x0.compatible
              and y.type="monitor" and y.descriptor like "19 inch"
```

The resulting query is normalized. First, Syntactic rule 1, which deals with nested select expressions in the FROM clause, is applied. Then the tuple constructor followed by a field selection is simplified, using Syntactic rule 6. We obtain the following query:

```
Q5:     select distinct [code:=a.code, descriptor:=a.capacity]
        from a in disks, y in products
        where a.manufacturer like "HP" and y.type="monitor"
              and y.descriptor like "19 inch"
              and y in (select distinct b
                        from b in products, z in monitors
                        where z.code=b.code and z.manufacturer in a.compatible)
```

Syntactic rule 2 is applied to obtain the following query:

```
Q6:     select distinct [code:=a.code, descriptor:=a.capacity]
        from a in disks, y in products, b in products, z in monitors
        where a.manufacturer like "HP" and y.type="monitor"
              and y.descriptor like "19 inch" and y=b and z.code=b.code
              and z.manufacturer in a.compatible
```

Since y must be equal to b, we can replace all occurrences of y by b and eliminate the second collection in the FROM clause of the previous query, yielding:

```
Q7:     select distinct [code:=a.code, descriptor:=a.capacity]
        from a in disks, b in products, z in monitors
        where a.manufacturer like "HP" and b.type="monitor"
              and b.descriptor like "19 inch" and z.code=b.code
              and z.manufacturer in a.compatible
```

Next, we apply the following mapping rule, which states that the information corresponding to products of type monitor are obtained from the local database #1.

```
M2: select distinct                    ⇒    select distinct
      [code:=x.code,                            [code:=x.code,
       descriptor:=x.descriptor,                 descriptor:=x.dimension,
       manufacturer:=x.manufacturer,             manufacturer:=x.manufacturer,
       compatible:=x.compatible]                 compatible:=
                                                       (select distinct y
                                                        from y in products, z in disks
                                                        where z.code=y.code and
```

```
                                                    z.disk-group in x.compatible)
        from x in products                    from x in monitors
        where x.type="monitor"
```

As in the previous rule application, the first step is to force the left-hand side expression of the rule to be a subexpression of the query Q7, using the procedure **FindSubquery**. It succeeds and returns the substitution {}. The query resulting after the application of this rewriting rule is then normalized using Syntactic rule 1:

```
Q8:     select distinct [code:=a.code, descriptor:=a.capacity]
        from x in monitors, a in disks, z in monitors
        where a.manufacturer like "HP" and x.dimension like "19 inch" and
              z.code=x.code and z.manufacturer in a.compatible
```

We eliminate the first collection in the FROM clause of the previous query, based on key information. We apply the assertion given in Example 2, which states that two objects of type Monitor are identical if and only if they have the same code. We now obtain an *un-nested form* of the reformulated alternative A2, described in the example 1, as follows:

```
Q9:     select distinct [code:=a.code, descriptor:=a.capacity]
        from a in disks, z in monitors
        where a.manufacturer like "HP" and x.dimension like "19 inch" and
              z.manufacturer in a.compatible
```

Using the data replication integrity assertion given in Example 4, we can substitute for the underlined expression to obtain the following query. It corresponds to an *un-nested form* of the reformulated alternative A1, described in the example 1, as follows:

```
Q10:    select distinct [code:=a.code, descriptor:=a.capacity]
        from a in disks, z in monitors
        where a.manufacturer like "HP" and x.dimension like "19 inch" and
              a.disk-group in z.compatible
```

For this query Q10, the decomposition algorithm, using the partitioning {disks}, {monitors} will produce the following nested form of the alternative A1, given in the example 1:

```
Q11:    select distinct [code:=y.F2, descriptor:=y.F3]
        from x in (select distinct [F1:=z.compatible]                  /* database #1 */
                   from z in monitors
                   where z.dimension like "19 inch"),
             y in (select distinct [F2:=a.code, F3:=a.capacity, F4:=a.disk-group]
                   from a in disks                                     /* database #2 */
                   where a.manufacturer like "HP")
        where y.F4 in x.F1
```

7. Comparison with Related Work

Much of the prior research on query processing is based on a common object-oriented model, and assumes a global schema which is the union of the local schemas, *eg.*, Pegasus [2], UniSQL [14], Garlic [7]. This simplifies query reformulation, but restricts the global interface by maintaining the autonomy of local interfaces. Furthermore, in these cited systems, semantic knowledge is not exploited, and prevents

expressing a view over local databases, reusing the results of previous queries, or exploiting data redundancy.

The initial use of semantic knowledge in the form of integrity constraints, in federated databases, is reported in [9]. They use Horn clauses, (definite databases), as their representation language. Schema mappings from the local databases to the global entities are defined using Horn clauses. As an aid to obtaining optimized compiled queries, they also express "data integration dependencies", functional dependencies based on primary keys, and inclusion dependencies, as integrity assertions. The research cited in [18,19], generalizes on this research.

The system described in [18] performs query reformulation using schema mapping knowledge. Their common object model is an object-oriented extension of the relational model based on a description logic. The representation language is Datalog-like, and thus, their queries are not as expressive as OQL queries. A concept in the *world view* (mediator interface in our CIS-MDBMS environment) may be expressed as a conjunctive Datalog-like query over the local relations, and they may also express a local relation as a (conjunctive) query over the world view relations. However, they are not able to express general integrity constraints in the local interfaces. The reformulation algorithm described in [18] is limited, since they try to match each global entity in the world view, against the mapping knowledge. Thus, they are not able to match all conjunctive queries expressed over the the world view entities, even if there exists a local entity defining this world view query (or a fragment of it).

They cite an extension of their algorithm [19], which is able to answer a larger class of queries, by matching a conjunctive query against a conjunctive view, to produce an equivalent query, and the algorithm is NP-complete. The intent is to obtain an equivalent query which is minimal, in that they reduce the number of literals that appear in the equivalent query. However, they note that minimality is not essential in obtaining an optimized equivalent query. This is especially true in a heterogeneous environment, where the view may be expressed over local information sources, which have dissimilar costs.

In comparison to [18], the OQL query language that we use to express semantic knowledge is much more expressive. We are able to express rewrite rules which replace a view in the mediator interface with an OQL query over the union of the local *and* the mediator interface. Thus, we are directly able to describe a mapping corresponding to an object in the mediator interface, which may have a reference, (ODMG relationship), with another object. Such a mapping for object references could not be explicitly expressed in any previous work. We are also able to utilize other semantic knowledge, e.g., data replication, for query reformulation.

The extended pattern matching of our reformulation algorithm allows us to identify (a subquery of) a user query which can be replaced by a rewrite rule. Since the result of query, which is essentially a view, can be used to replace a subquery in the user query, we are able to cover the same space as the the algorithm in [19], with the caveat that we are reformulating wrt a much more complex and expressive query

language. We also note that the space of query reformulation is not necessarily those queries in which we minimize the number of collections, as described in[19]. However, we are able to eliminate some collections in the query, based on semantic knowledge. This simplification is more general than the minimality criterion of [19], which does not exploit semantic knowledge.

The SIMS project [3] also performs some reformulation, but it is based on a fixed set of reformulation operators. They, too, do not use a standard object model or standard query language. They are not able to express a concept in the CIS-MDBMS level as a view over the local interfaces, nor can they express or exploit other semantic knowledge during reformulation, to generate alternate queries. Other recent proposals for transforming multidatabase queries are based on higher-order query languages [16], higher-order logics [17], or meta-models [5]. Each of these depends on using a query language or model that is not standard, (and more complex), compared to the relational or object models and languages.

8. Conclusions

In this paper, we address the problem of query reformulation in CIS-MDBMS, with a common model and language, based on the ODMG standard, and local databases that may be relational, object-oriented, or file systems. The MDBMS interface or mediator interface could be different from the union of the local interfaces, and may include views of particular local databases, integrity constraints, and knowledge about data replication in local databases.

Query reformulation is an important step in CIS-MDBMS query processing, and transforms an input OQL query in the mediator interface, into equivalent OQL queries in the (union of) the local interfaces. When this query spans several local interfaces, a process of query decomposition is used to obtain independent local sub-queries for each interface, and a composite query to regroup the results at the mediator interface.

In order to guarantee to correctness of the reformulation, (*i.e.*, the reformulated queries in the local interfaces produces the expected answer), it is necessary to use a variety of semantic knowledge, (*i.e.* integrity constraints, data redundancy, schema mappings) which describe the CIS-MDBMS.

Our solution to CIS-MDBMS query reformulation relies on the uniform expression of semantic knowledge, as rewrite rules, in a canonical form of OQL expression. OQL-based rewrite rules provide a very expressive language for specifying equivalent queries. Compared to previous reformulation work in [3], [18], we support reformulation using an expressive query language and a variety of semantic knowledge.

The CIS-MDBMS reformulation algorithm is based on pattern-matching, and uses both syntactic rewriting to express a query in canonical form, and semantic rewriting using the rewrite rules, to obtain alternate equivalent OQL queries. Furthermore, our query reformulation can exploit good optimization opportunities. For instance, the fact that data can be replicated in several local databases, that there are constraints that allow simplification, and that the results of previous query ex-

ecution may be stored in the CIS-MDBMS for later re-use, can yield alternative ways of computing the same query. This is important to ease the subsequent task of query optimization in the CIS-MDBMS, which must select the best reformulated query and produce an efficient execution plan, *eg.*, using a heterogeneous cost model or heuristics. Our ability to re-use the results of stored queries, and the ability to identify sub-queries during the rewrite procedure, can be exploited in a heterogeneous cost model. The cost model can store the cost of computing queries, in an implementation independent manner. This is important in a heterogeneous environment, where we wish to preserve the autonomy of each local database.

We have validated all the proposed algorithmic solutions by extending the Flora compiler prototype [12]. We used this extended prototype to represent a variety of semantic knowledge, and to experiment with reformulation of several sample queries in our schema. The Flora compiler supports the ODMG data model and query language, and produces code for a wrapper for the O2 DBMS [4].

References

1. S. Abiteboul, A. Bonner, Objects and views, *Proc. ACM Sigmod Conference*, (1991).
2. R. Ahmed *et al .*, The Pegasus heterogeneous multidatabase system, *IEEE Computer* **24(12)**, (1991).
3. Y. Arens, C.Y. Chee, C.-N. Hsu and C.A. Knoblock, Retrieving and integrating data from multiple information sources, *International Journal of Cooperative Information Systems*, **2(2)**, (1993).
4. F. Bancilhon, C. Delobel, and P. Kannelakis (eds.), *Building an Object-Oriented Database System - The Story of O2* (Morgan Kaufmann, 1992).
5. T. Barsalou and D. Gangopadhay, M(DM): An open framework for interoperation of multimodel multidatabase systems, *Proc. IEEE Intl. Conference on Data Engineering*, Tempe, AZ, (February 1992).
6. P. Buneman and D. Maier, The data that you won't find in databases: tutorial panel on data exchange formats, *Proc. ACM Sigmod, panel session*, San Jose, (May 1995).
7. M. Carey *et al .*, Towards heterogeneous multimedia information systems: the Garlic approach, *Technical Report, IBM Almaden Research*, (1995).
8. R.G.G. Cattell *et al .*, *The Object Database Standard - ODMG 93*, (Morgan Kaufmann, 1993).
9. Chakravarthy, S, Whang, W-K. and Navathe, S.B., A Logic-based approach to query processing in federated databases, *Technical Report, University of Florida*, (1993).
10. S. Chaudhuri and K. Shim, Query optimization in the presence of foreign functions, *Proc. International Conference on Very Large Data Bases*, Dublin, Ireland, (August 1993).
11. W. Du, R. Krishnamurthy and M.-C. Shan, Query optimization in heterogeneous DBMS, *Proc. International Conference on Very Large Data Bases*, Vancouver, Canada, (September 1992).
12. D. Florescu and P. Valduriez, Rule-based query processing in the IDEA system, *Proc. Intl. Symp. on Advanced Database Technologies and Their Integration*, Nara, Japan, (October 1994).
13. D. Florescu, L. Raschid, and P. Valduriez, Using heterogeneous equivalences for query rewriting in multidatabase systems, *Proc. International Conference on Cooperative Information Systems*, (1995).

14. W. Kim *et al* ., On resolving schematic heterogeneity in multidatabase systems, *Distributed and Parallel Databases*, **1(3)**, (1993).
15. W. Kirk, A.Y. Levy, Y. Sagiv and D. Srivastava, The Information Manifold, *Distributed and Parallel Databases*, **1(3)**, (1993).
16. R. Krishnamurthy, W. Litwin and W. Kent, Language features for interoperability of databases with schematic discrepancies, *Proc. ACM SIGMOD Intl. Conf.*, Denver, CO, (May 1991).
17. L.V.S. Lakshmanan, F. Sadri and I.N. Subramanian, On the logical foundations of schema integration and evolution in heterogeneous database systems, *Proc. Intl. Conf. Deductive and Object-Oriented Databases*, (1993).
18. A.Y. Levy, D. Srivastava and T. Kirk, Data model and query evaluation in global information system, *Intl. Journal on Intelligent Information Systems - special issue on Networked Information Retrieval*, (1995).
19. A.Y. Levy, A.O. Mendelzon, Y. Sagiv and D. Srivastava, Answering queries using views, *Proc. ACM Principles of Database Systems Symp.*, (1995).
20. R.J. Miller, Y.E. Ioannidis and R. Ramakrishnan, The use of information capacity in schema integration and translation, *Proc. International Conference on Very Large Data Bases*, (1993).
21. G. Mitchell, U. Dayal, and S. Zdonik, Control of an extensible query optimizer: a planning-based approach, *Proc. International Conference on Very Large Data Bases*, Dublin, (1993).
22. *The Common Object Request Broker: Architecture and Specification*, (Object Management Group, Framingham, MA, 1992).
23. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems.* (Prentice Hall, 1991).
24. T. Özsu, U. Dayal and P. Valduriez (eds.), *Distributed Object Management*, (Morgan Kaufmann, San Mateo, CA, 1993).
25. Y. Papakonstantinou, H. Garcia-Molina and J. Widom. Object exchange across heterogeneous information sources, *Proc. IEEE Intl. Conference on Data Engineering*, (1995).
26. Y. Papakonstantinou, A. Gupta, H. Garcia-Molina and J. Ullman. A query translation schema for rapid implementation of wrappers, *Proc. Intl. Conference on Deductive and Object-Oriented Databases*, (1995).
27. Y. Papakonstantinou, H. Garcia-Molina and J. Ullman. MedMaker: a mediation system based on declarative specifications, *Proc. IEEE Intl. Conference on Data Engineering*, (1996).
28. L. Raschid, Y. Chang and B. Dorr, Query transformation techniques for interoperable query processing in cooperative information systems, *Proc. Intl. . Conf. on Cooperative Information Systems*, (1994).
29. Raschid, L. and Chang, Y.-H., Interoperable query processing from object to relational schemas based on a parameterized canonical representation, *Intl. Journal of Cooperative Information Systems*, (1995).
30. A. Sheth and J. Larson, Federated database systems for managing distributed heterogeneous and autonomous databases, *ACM Computing Surveys*, **22(3)**, (1990).
31. A. Tomasic, L. Raschid and P. Valduriez. Scaling heterogeneous databases and the design of DISCO, *Proc. Intl. Conference on Distributed Computer Systems*, (1996).
32. J.D. Ullman, *Principles of Database and Knowledge-Base Systems*, (Computer Science Press, Vol. 1, 1988).

Appendix A

Syntactic Rule 1 *: Unnesting the select subexpressions nested in the from clause*

select [distinct] $proj_1$
from var_{11} in C_{11}, $\cdots$, var_{1i-1} in C_{1i-1},
  var_{1i} in (select [distinct] $proj_2$
    from var_{21} in C_{21} , $\cdots$, var_{2m} in C_{2m}
    where $pred_2$), var_{1i+1} in C_{1i+1}, $\cdots$, var_n in C_{1n}
where $pred_1$

is rewritten as

select [distinct] $proj'_1$
from var_{11} in C_{11}, $\cdots$, var_{1i-1} in C_{1i-1},
  var_{21} in C_{21}, $\cdots$, var_{2m} in C_{2m},
  var_{1i+1} in C'_{1i+1}, $\cdots$, var_n in C'_{1n}
where $pred'_1$ and $pred_2$

where $proj'_1$, C'_{1i+1}, $\cdots$, C'_{1n} and $pred'_1$ are obtained from $proj_1$, C_{1i+1}, $\cdots$, C_{1n}, and $pred_1$ respectively, by replacing all occurrences of the variable var_{1i} by $proj_2$. The necessary condition for this rule is that either both inner and outer select-expressions include distinct, or neither of them does.

Syntactic Rule 2 *: Testing membership in the result of a nested select*

select [distinct] $proj$
from var_1 in C_1, $\cdots$, var_n in C_n
where $pred$ and $elem$ in (select [distinct] $proj'$
    from var_{n+1} in C_{n+1}, $\cdots$, var_{n+m} in C_{n+m}
    where $pred'$)

is rewritten as

select [distinct] $proj$
from var_1 in C_1, $\cdots$, var_n in C_n, var_{n+1} in C_{n+1} , $\cdots$, var_{n+m} in C_{n+m}
where $pred$ and $pred'$ and $elem$=$proj'$

The necessary condition for this rule is that at least the inner or the outer select-expressions includes distinct. The resulting select also includes distinct, if and only if the outer select-expression includes distinct.

Syntactic Rule 3 *: Existential quantifier transformation*

select [distinct] $proj$
from var_1 in C_1, $\cdots$, var_n in C_n
where $pred$ and (exists var_{n+1} in C_{n+1} : $pred'$)

is rewritten as

select [distinct] $proj$
from var_1 in C_1, $\cdots$, var_n in C_n, var_{n+1} in C_{n+1}
where $pred$ and $pred'$

This rule can be applied if at least one of the following conditions hold: the initial select-expression includes distinct or C_{n+1} is a set expression. The resulting select-expression includes distinct if and only if the initial select-expression includes distinct.

Syntactic Rule 4 *: Transformation of a navigation into a join*

Suppose $expr$ is a subexpression of $pred$ whose corresponding type τ is an object whose extension is maintained. Then, the following select-expression:

select [distinct] $proj$
from var_1 in C_1, $\cdots$, var_n in C_n
where $pred$

is rewritten as

select [distinct] $proj$
from var_1 in C_1, $\cdots$, var_n in C_n , var_{n+1} in $extent(\tau)$
where $pred'$ and var_{n+1}=$expr$

where $pred'$ is obtained from $pred$ by replacing the occurrences of $expr$ by var_{n+1}. The resulting select-expression includes the distinct if and only if the initial selectexpression includes distinct.

Syntactic Rule 5 *: Eliminating the dependency between input collections*

Suppose the lambda variable var_i appears in some expression C_j (i<j), and the type of the elements of the collection C_j is an object type τ whose extension is maintained. Then, the following select-expression:

select [distinct] $proj$
from var_1 in C_1, $\cdots$, var_i in C_i, $\cdots$, var_j in C_j , $\cdots$, var_n in C_n
where $pred$

is rewritten as

select [distinct] $proj$
from var_1 in C_1, $\cdots$, var_i in C_i, $\cdots$, var_j in $extent(\tau)$, $\cdots$, var_n in C_n
where $pred$ and (var_j in C_j)

This rule is applied if at least one of the following conditions hold: the initial select-expression includes distinct or C_j is a set expression. The resulting select-expression includes distinct if and only if the initial select-expression includes distinct.

Syntactic Rule 6 *: Simplification of tuple constructors*

$[field\_name_1 := expr_1, \cdots, field\_name_n := expr_n].field\_name_i$ is rewritten as $expr_i$.

Syntactic Rule 7 *: Introducing* **distinct** *in a select-expression*

The *distinct* is necessary for the application of many transformations. The criteria for introducing distinct is that the select-expression corresponding to a projection represents a unique identifier for the tuple x_1, $\cdots$, x_n [j]. This condition is verified based on key information.

select $proj$
from x_1 in C_1, $\cdots$, x_n in C_n
where $pred$

is rewritten as

select distinct $proj$
from x_1 in C_1, $\cdots$, x_n in C_n
where $pred$

[j] The condition to be verified is that $proj(x_1, \cdots, x_n) = proj(x'_1, \cdots, x'_n)$ implies $x_1 = x'_1 \& \cdots \& x_n = x'_n$.

Appendix B

procedure **FindSubquery**(Q′, Q)
/*
Input: two select-expressions Q′ and Q of the following form:

Q′= select $proj_1$ / from x_{11} in C_{11}, ⋯, x_{1n} in C_{1n} / where p_{11} and p_{12} and ⋯ and p_{1q}

Q= select $proj_2$ / from x_{21} in C_{21}, ⋯, x_{2m} in C_{2m} / where p_{21} and p_{22} and ⋯ and p_{2r}

Output: a substitution θ and a select-expression Q″, logically equivalent with Q, and containing Q′θ as a subexpression
*/
{ let θ={};

/* STEP 1. Matching the expressions corresponding to the collections in the from clause */
Let UC=$\{C_{21}, \cdots, C_{2m}\}$ /* the set of unmatched collections */
Let MC={} /* the set of matched collections */
foreach C_{1i} $(i = 1 \ldots n)$
{find a $C_{2j_i} \in$ UC so Match($C_{1i}\theta$, C_{2j_i}) succeeds with substitution θ'
if found then θ=$\theta \cup \theta' \cup \{x_{1i}/x_{2j_i}\}$, remove C_{1j_i} from UC and add C_{1j_i} to MC
else fail}

/* STEP 2. Matching the expressions corresponding to the conjunctions in the where clause */
Let UP=$\{p_{21}, \cdots, p_{2r}\}$ /* the set of unmatched conjunctions */
foreach p_{1i} $(i = 1 \ldots q)$
{find a $p_{2j_i} \in$ UP so Match($p_{1i}\theta$, p_{2j_i}) succeeds with substitution θ'
if found then θ=$\theta \cup \theta'$ and remove p_{1j_i} from UP else fail}

/* STEP 3. Constructing the resulting select-expression */
Let x_0 be a new lambda variable, distinct from $x_{11}, \cdots, x_{1n}, x_{21}, \cdots, x_{2m}$
if $proj_1$ is of the form [field$_1$:=$expr_1$, ⋯, field$_p$:=$expr_p$] then
replace each $expr_i\theta$ in $C_{21}, \cdots, C_{2m}$, $proj_2$ and the unmatched predicates by x_0.field$_i$
else
replace each $proj_1\theta$ in $C_{21}, \cdots, C_{2m}$, $proj_2$ and the unmatched predicates by x_0
Let X=the subset of matched collections MC for which the corresponding lambda var. appear in some unmatched predicate or in $proj_2$ (after replacement)
Find $\overline{X}$=the minimal subset of matched collections MC verifying the following:
◇ X $\subset \overline{X}$
◇ for each $C_{2i} \in \overline{X}$, if exists $C_{2j} \in X$, with x_{2j} appearing in C_{2i}, then, $C_{2j} \in \overline{X}$
Let *link_predicate* = {} /* *link_predicate* is a set of conjunctions */
foreach $C_{2j} \in \overline{X}$
{if $proj_1$ is a tuple and it contains a field of the form: $field_l$:=x_{2j}.*key*
where *key* is the key field of some collection C_{2j},
and x_{2j} is a lambda variable associated with C_{2j}
then add the conjunction x_{2j}.key= x_0.field$_l$ to the *link_predicate*
else fail }

return the substitution θ and the following equivalent query for Q

select $proj_2$
from x_0 in Q′θ, x_{2i_1} in C_{2i_1}, ⋯, x_{2i_s} in C_{2i_s}, x_{2j_1} in C_{2i_1}, ⋯, x_{2i_s} in C_{2j_t}
where p_{2j_1} and ⋯ and p_{2j_t} and p_{2l_1} and ⋯ and p_{2l_z}

where UC= $\{C_{2i_1}, \cdots, C_{2i_s}\}$, $\overline{X}$=$\{C_{2j_1}, \cdots, C_{2j_t}\}$, UP=$\{p_{2k_1}, \cdots, p_{2k_w}\}$
and *link_predicate*=$\{p_{2l_1}, \cdots, p_{2l_z}\}$
}

Appendix C

procedure **Decomp**(Q, $\mathcal{P}_1$, ..., $\mathcal{P}_n$)
{Let UP=$\{p_1, \ldots, p_q\}$ /* the set of initial conjunctions in the predicate of Q*/
foreach $\mathcal{P}_i$ (i=1...n) /* construct the expression Q_i corresponding to $\mathcal{P}_i$*/
{Let y_i be a new lambda variable /* whose domain will be Q_i*/
Let $proj_i$=empty list /* the list of projections of Q_i*/
Let $pred_i$=empty list /* the list of conjunctions in the predicate of Q_i*/
foreach p_j∈UP
if p_j does not use any lambda variables associated with some collection
C_k ∈$\{C_1, \ldots, C_m\}\backslash\mathcal{P}_i$
then add p_j to $pred_i$ and remove p_j from UP
Let $\mathcal{E}_i$ be the set of maximal subexpressions of *proj*, and of the conjunctions in UP,
in which any lambda variables are only associated with collections from $\mathcal{P}_i$.
We use the term maximal in that no supraexpression verifies the same property.
foreach $e_j \in \mathcal{E}_i$
{Add a new field with name f_j and value e_j to $proj_i$ /* the projection of Q_i*/
Replace all occurrences of e_j in *proj* and the conjunctions of UP
with the expression $y_i.f_j$ }
Construct Q_i as the following expression, where $\mathcal{P}_i$=$\{C_{i1}, C_{i2}, \ldots, C_{il_i}\}$
select $proj_i$
from x_{i1} in C_{i1} and x_{i2} in C_{i2} and ... and x_{il_i} in C_{il_i}
where $pred_i$
} /* end of construction of Q_i*/
Construct the composing query as the following expression, where UP=$\{p_{j_1}, \ldots, p_{j_t}\}$:
select *proj*
from y_1 in Q_1 and y_2 in Q_2 and ... and y_i in Q_i
where p_{j_1} and ... and p_{j_t}
}

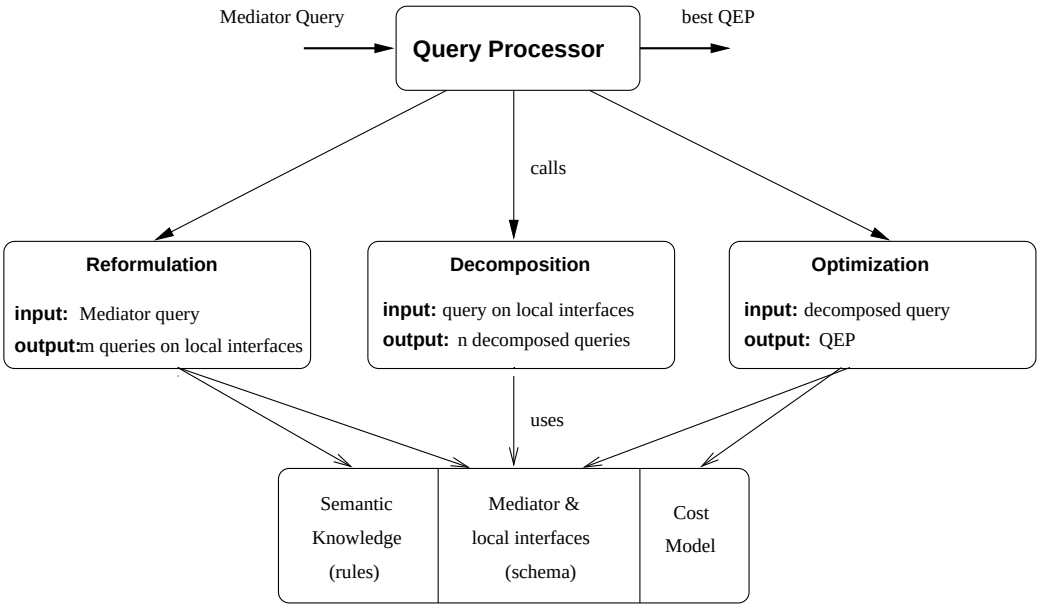

Figure 5: Simplified Flora compiler architecture.