Calculus-Based Transformations of Queries over Object-Oriented Views in a Database Mediator System

Vanja Josifovski and Tore Risch
Laboratory for Engineering Databases
Linköping University
Sweden

vanja@ida.liu.se, torri@ida.liu.se

Abstract

The concept of object-oriented (OO) views has been a popular approach to data integration. Nevertheless, there have been few reported results on optimization of queries over integrated OO views. In our work, we have developed an OO view system for data integration based on the AMOS database mediator system. The paper describes a system architecture and implementation that takes advantage of query optimization techniques to improve the performance of queries to integrated OO views. The main features of the system are: 1) A passive mediation framework that preserves the autonomy of the data sources. 2) A selective materialization mechanism that minimizes the number of materialized view objects. 3) A predicate based mechanism to guarantee the validity of the materialized view objects as well as the completeness of queries to the view. In order to reduce the overhead of the passive view integration, we use inexpensive calculus based transformations to generate minimal query expressions before the query decomposition and the cost-based algebraic optimization take place.

1. Introduction

An important factor of the strength of a modern enterprise is its capability to effectively store and process information. As a legacy of the mainframe computing trend in the previous decades, large enterprises often have several isolated data repositories used only within portions of the organization. The development of the network technology bridged the gap between these systems, but the access of the data in their diverse native formats is a burden to the user. Another trend is that terminals as access points are replaced by more powerful workstations having substantial processing capabilities, but which are nevertheless too small to hold all the data that a user might need. In this kind of computing environment we can assume that the big volume data will still reside in dedicated data processing servers. Beyond this, some data will reside in the users' workstations either because it is mainly of local interest, or because the user wishes to have local control of the access.

A user in such an environment should be provided with a location transparent and semantically coherent view of the data in the different repositories. The *wrapper-mediator* approach, described in [19], divides a system with this kind of functionality into two subsystems. The wrapper system translates the data expressed in the local data models of the data sources into a common data model (CDM). The task of the mediator is to provide a semantically coherent CDM representation of the data in the repositories that each may contain different (meta)data describing the same real world entities.

The research in the field of mediator systems has identified two basic approaches. One uses eager materialization of the queried data, trying to reduce the response time by performing most of the costly operations before the query is issued [8]. The other approach, which we name passive, fetches the required data when it is requested. Which approach yields better results depends on factors such as available resources, the size of the data and the query and update frequencies [8].

In this paper, we present an approach to database mediation based on passive evaluation techniques. We use object-oriented (OO) views to provide the user with a unified appearance of data in different repositories. The queries over the views are transformed to queries over the data in the repositories. When the passive approach is used, the mediator system has to provide for complete and consistent answers to the queries over the OO views at the time when the queries are issued. An advantages of the approach described

in this paper is that it provides an efficient view support mechanism by describing the system tasks using predicates inserted in the calculus representation of the queries over the integrated views. This allows for query optimization of the view support tasks together with the user-specified part of the query. Another advantage is that the view maintenance operations, as well as the user-specified operations, are specified and performed over a set of objects/tuples as opposed to individual instances.

The focus of the paper is a query transformation technique which, for a certain class of queries, allows for a reduction of the number of predicates by applying calculus-based optimization. The calculus-based optimization removes redundant computations that often result from merging system-specified and user-specified predicates in the query. This reduces the query complexity and, because it is performed by simple rewrite rules, it imposes minimal increase in the query processing time. The cost based optimization executed later in the query processing is concerned with the order of the execution rather than the removal the redundant computations.

The work presented in this paper should be contrasted with the traditional approach where the view support tasks are performed by code within the system. In this approach, used for example in [15] and [6], most of the view support tasks are performed on an individual instance level, and the optimizations described in this paper are therefore not applicable.

The paper is organized as follows. In section 2 we introduce the AMOS database mediator system that serves as a platform for the work presented later in the paper. Section 3 introduces our OO views architecture for database mediation. Section 4 describes the query transformation techniques for the queries over the OO views and the use of rewrite rules to reduce the number of query predicates. Approaches related to our work are outlined in section 5. A summary and future directions of our research are given in section 6.

2 Overview of the AMOS system

As a platform for our research we use the AMOS mediator database system [4] developed from a workstation version of the Iris system, WS-Iris [14]. The core of AMOS is an open, light-weight, and extensible database management system (DBMS). The aim of the AMOS architecture is to provide for efficient integration of data stored in different repositories by both active and passive techniques. To achieve better performance, and because most of the data reside in the data repositories, AMOS is designed as a main-memory DBMS. Nevertheless, it contains all the traditional database facilities, such as a recovery manager, a transaction manager, active rules, and a OO query language. An AMOS server provides services to applications and to other AMOS servers.

The data model used in AMOS, which is also used as a CDM for mediation, is an OO extension of the DAPLEX [17] functional data model. It has three basic constructs: *objects*, *types* and *functions*. Objects model entities in the domain of interest. An object can be classified into one or more types which makes the object an *instance* of that type(s). The set of all instances of a type is called the *extent* of the type. Object properties and their relationships are modeled by functions.

The types in AMOS are divided into literal types (e.g *real* and *charstring*) and surrogate types. The literal types have fixed, possibly infinite extents and self identifying instances. Each instance of a surrogate type is identified by a unique, system-generated object identifier (OID). The types are organized in a multiple inheritance, supertype/subtype hierarchy where an instance of a type is also an instance of all the supertypes of that type; conversely, the extent of a type is a subset of the extent of a supertype of that type (extent-subset semantics).

The functions are divided by their implementations into three groups. The extension of a *stored* function is physically stored in the database. *Derived* functions are implemented in the AMOS' query language AMOSQL. *Foreign* functions are implemented in other programming language as, for example, Lisp or C++.

The AMOSQL query language is based on the OSQL [14] language with extensions of multi-way foreign functions, active rules, late binding, overloading, etc. The following example illustrates the AMOSQL syntax. Assuming that three stored function *parent*, *name* and *hobby* are defined, it retrieves the names of the parents of the persons who have 'sailing' as a hobby:

```
select p, name(parent(p))
from person p
where hobby(p) = 'sailing';
```

Figure 1, presents an overview of the query processing in AMOS. The first five steps, also called *query compilation* steps, translate a query expressed in AMOSQL to an object algebra plan that can be stored and interpreted many times without repeating the compilation. To illustrate the query compilation we use the query from the previous example. From the parsed query tree, the calculus generator generates a calculus expression with flattened and type resolved predicates. Flattened predicates have a variable or constant as a left-hand side (or a tuple of variables or constants when the function returns a tuple as a result), and an unnested function call, variable or constant as a right-hand side. The head of the calculus query expression contains the result variables. A calculus expression can also have a name

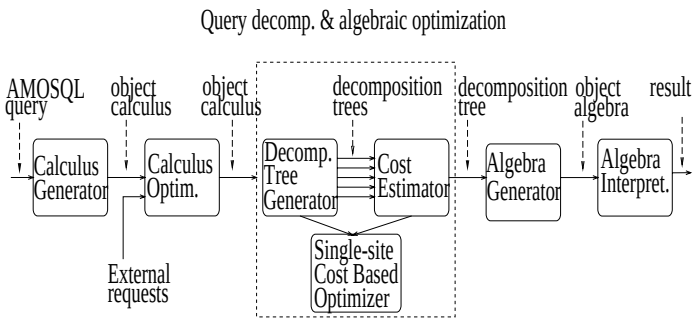

Figure 1. Query processing in AMOS

and input parameters, in which case it represents a derived function.

AMOS supports overriding and overloading of functions on the types of the arguments and the results. Each function name refers to a *generic* function which can have more than one associated *type resolved* functions. Each generic function call in a query is substituted by a type resolved one during the calculus generation process. Late binding is used for the calls which, due to polymorphism, cannot be resolved during query compilation. AMOS' late binding mechanism is described in [7]; in the examples throughout the paper we assume that all function calls are resolved during the compilation of the queries. Accordingly, the result of the calculus generation phase for the example query is given by the calculus expression below:

$$\{\, p, nm \mid$$
$$p = Person_{nil \to person}() \wedge$$
$$d = parent_{person \to person}(p) \wedge$$
$$nm = name_{person \to charstring}(d) \wedge$$
$$'sailing' = hobby_{person \to charstring}(p)\}$$

Notice that type resolved functions are annoted with the types of their arguments and results. The first predicate in the expression is inserted by the system to assert the type of the variable p. It defines the variable p with the *extent function* of type $Person$ that returns all the instances of this type.

Next, the calculus optimizer applies rewrite rules to reduce the number of predicates. In the example, it produces the expression below by removing the type check predicate. The typecheck can be removed because p is used in a stored function (e. g. $name$) with an argument or result of type $person$. The referential integrity system of the stored functions constrain the instances stored in the stored function to be of the correct type [14].

$$\{\, p, nm \mid$$
$$d = parent_{person \to person}(p) \wedge$$
$$nm = name_{person \to charstring}(d) \wedge$$
$$'sailing' = hobby_{person \to charstring}(p)\}$$

Because the example query is over local types, it passes unaffected through the query decomposition stage and is processed only by the cost-based single-site algebra optimizer. If some part of the query should be executed at another AMOS server, the system uses primitives that allow for sending and evaluating calculus expressions between the servers. These features of the system are not described in this paper and will be a topic of a forthcoming paper.

While object calculus query representation is unordered and contains function references that do not have specified binding patterns (i. e. which function parameters are input and which are output [14]), the algebra plan is ordered and contains type and binding pattern resolved function references. The calculus optimization process takes advantage of the declarative unordered format and the unspecified binding patterns of the object calculus for detection of optimization possibilities with goal to reduce the number of query predicates. This optimization is rule driven and much simpler than the transformations done during the cost-based query decomposition and algebraic optimization.

An interested reader is referred to [10] for more detailed description of the AMOS system and to [14], [5] and [7] for more comprehensive descriptions of the query processing in AMOS.

3. Object-Oriented View System Design

This section presents the design principles behind the OO view mechanism for data integration in AMOS. Views as a tool for data abstraction and restructuring, have been extensively studied in the context of the relational databases. The design of a view mechanism in an OO environment has to account for its increased complexity compared to the relational. This particularly refers to the inheritance and the object identity concepts. Inheritance and views have some common aims (e.g. data abstraction and reuse of code/queries) and therefore, the two mechanisms must be combined in a semantically clear manner. Two important issues in the OO view system design, related to the object identity, are the format of the view objects' identifiers and the definition of view objects' life span.

Additional issues arise when views are defined over data residing in multiple autonomous repositories. First, as mentioned above, because active maintenance of the views is not always possible, non-active mechanisms for view maintenance must be provided. Furthermore, the same real world concepts can have different representations in different repositories. To provide the user with a semantically consistent view of the data, the system needs to provide constructs for overcoming such differences in the view definition. Finally, there is also the issue of the representation of OIDs in the inter-database communication.

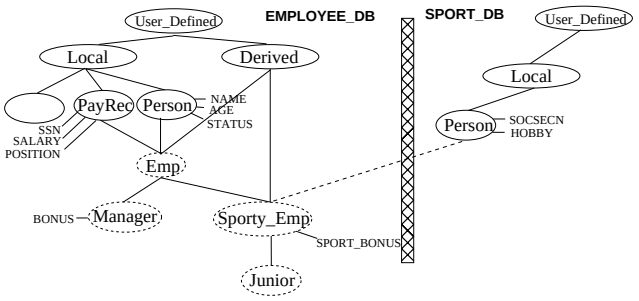

Figure 2. Integration by subtyping

3.1 Derived Types

In order to provide data integration features in AMOS, we extended the type system with a construct named *derived type* (DT). Data integration by DTs is performed by building a hierarchy of DTs based on local and imported data types (i.e. from other AMOS systems). DTs are defined by supertyping and subtyping from other types in the type hierarchy. The traditional inheritance mechanism, where the corresponding instances of an object in the super/subtypes are identified by the same OID, is extended with declarative specification of the correspondence between the instances of the derived super/subtypes. Integration by sub/supertyping is related to the mechanisms used in some other systems as, for example, the integrated views and column adding in the Pegasus system [3], but is better suited for use in an OO environment.

Figure 2 shows an example of integration by subtyping. In the example, the data stored in an employee database is integrated with the data from a database storing sporting information. The solid ovals represent ordinary types while the dashed ovals are DTs. Stored functions defined over the types in the figure are shown beside the type ovals. The types *User_Defined*, and *Derived* are system-defined and part of AMOS' meta-model. They are defined in both databases, but are not shown in $Sport\_Database$ for simplicity. There is a type *Person* in both databases, each storing information about a set of persons. The definition of the derived portion of the type hierarchy in the example is done as follows. First, the DT *Emp* is created to represent the persons who have a pay record. The DT *Manager* is created as a subtype of the DT *Emp* to represent the employees for which the inherited attribute (stored function) *position* contains the string 'Manager'. The type importation is done by subtyping from types in other database mediators, as illustrated by the DT *Sporty_Emp*. This DT is defined as subtype of the local DT Emp and the type $Person$ in the sport database. Its instances represent persons for which there is an instance in the Emp type in the employee database, and in the $Person$ type in the sport database.

The figure also points at some design choices we committed to in the development of the system. First, to be able to do data integration by subtyping, a DT needs to inherit from more than one type, i.e. multiple inheritance. Second, it can be noticed in the example that stored functions (e.g. *sport_bonus* in *Sporty_Emp*) can be defined over DTs, which makes the DTs a *capacity-augmented* view mechanisms [15]. One of our design goals was to allow the DTs to be used in function definitions in the same manner as the ordinary types. This means that any function can have DTs as argument or result domains.

3.2 Derived Types and Object Identifiers

There are three basic choices when it comes to the format of the DT instance OIDs. The first is to use the identifiers from the corresponding objects in the supertypes [16]. This is not suitable for our DT instances because it is not compatible with multiple inheritance. The second alternative is to use a stored query expression instead of an identifier and construct the required DT instances by evaluating this expression [11]. With that approach, it would be difficult to have functions whose argument domain is a DT since it is not convenient to manipulate expressions as database objects. The third alternative is to generate new unique OIDs for the materialized DT objects [15]. With this method, the same conceptual object (i.e. representing the same real world entity) is represented by instances having different OIDs in different types. Therefore, to be able to evaluate inherited functions over materialized DT instances, their OIDs need to be mapped to the OIDs of the corresponding instances of the type over which the function was defined, a process named *OID coercion* [1]. The cost of the OID coercion is the main weakness of the approach to have unique OIDs for DT instances. Nevertheless, we chose this approach for the following two reasons: First, the major cost of a query is in the predicates that ship data between database mediators and not in the coercion. In AMOS, the hash tables used in the coercion are stored in main-memory which makes the coercion a relatively inexpensive operation. Second, expressing the coercion by predicates permits optimization of the calculus representation of the query, which further reduces the coercion cost, as described in the next section.

Although materialization of OIDs for the DT instances allows for using the DTs as domains for function arguments and results, not all queries require materialization and it will be a severe performance impairment to materialize the extents of all the DTs in each query. To minimize the materialization cost, the query processor analyses the query to find out which query variables represent instances that need to

[1] In the text we use interchangeably the terms "OID coercion" and "instance coercion"

be materialized. Materialization predicates are added only for query variables that are part of the query result or used as an argument of a foreign function. In many other cases query transformations are used to transform the query so that no materialization is needed, as shown in the next section. The performance of these queries is thus not degraded by the materialization mechanism. In the queries that do require materialization it is performed selectively only over the DT instances which also satisfy the rest of the query predicates, thus materializing only parts of the DT extents, in order to avoid unnecessary performance and storage overhead.

Materialized DT instances can also be used in queries issued at a time after their materialization. At this time, the system has to assert that they still comply with the declarative conditions stated in the DT definition, or, in other words, that they are still *valid*. Assuming non-active basic capability of the data sources, we have to provide a mechanism to check the validity of these instances in such queries.

The validity of a materialized DT instance depends on the existence and validity of the corresponding instances of the supertypes identified when the instance was materialized. When a DT instance is validated, the validation condition is executed only over these instances. This definition of the validity of a DT instance based on a combination of the OIDs of the supertype instances and a validation condition, is consistent with the OO structure of the database, and is efficient to implement.

The instance is present in the database until it is used in a query where it fails the validation test. A garbage collection of the DT instances can be implemented using the active capabilities of the AMOS system to periodically run the validation test.

3.3 Derived Types and Inheritance

An important issue in designing an OO view system is the placement of the DTs in the type hierarchy. The obvious approach would be to place the DTs in the same hierarchy as the ordinary types. However, mixing freely the DTs and ordinary types in a type hierarchy can lead to semantically inconsistent hierarchies [12]. In order to provide the user with powerful modeling capabilities along with a semantically consistent inheritance hierarchy, the ordinary and derived types in AMOS are placed in a single type hierarchy where it is not allowed to have an ordinary type as a subtype of a DT. This rule preserves the extent-subset semantics for all types in a hierarchy. If DTs were allowed to be supertypes of ordinary types, due to the declarative specification of the DTs, it is not possible to guarantee that for each instance of the ordinary type there will be a corresponding instance in its supertypes [12].

In the example in Figure 2 the derived part of the type

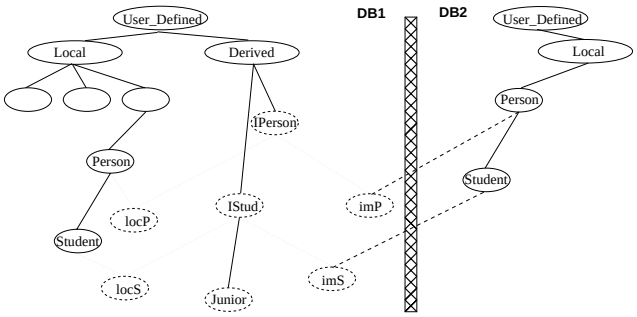

Figure 3. Integration by supertyping

hierarchy is constructed by subtyping. The AMOS mediation framework also allows definition of DTs as explicit supertypes of other DTs. Although processing of queries over this kind of DTs in not discussed in this paper, to complete the discussion on the mediation framework, Figure 3 presents an example of integration by supertyping. This example shows a definition of an integrated view of two person databases *DB1* and *DB2*. The data in both databases is structured in two user-defined types: a type named *Person* which contains data about a set of persons, and its subtype *Student* representing the persons that are students. The example establishes the *derived supertypes IPerson* and *IStud* in DB1 to provide an integrated view of the data in the databases. These types are supertypes of the types *locP* and *locS* which represent the instances from the types *Person* and *Student* in *DB1* which participate in the integration. The types *imP* and *imS* represent data imported into *DB1* from the types *Person* and *Student* in *DB2*. Derived supertypes can be subtyped as other DTs. In this example the type *Junior* is created to represent a specialization of the type *IStud* containing all junior students. The same schema was used in both databases in order to simplify the example. The presented mediation framework can handle arbitrary schema heterogeneity by defining mappings using derived sub- and supertypes and derived functions.

3.4 Derived Subtyping Language Constructs

In order to be able to define derived subtypes with properties as described above, we have extended the AMOSQL type declaration construct as:

```
CREATE TYPE type_name
   SUBTYPE OF sut1, sut2, ...
      COMPOSE compose_expression
      VALIDATE validate_expression
      HIDE fn1, fn2, ...
    PROPERTIES
      function definitions;
END_TYPEDEF;
```

The *subtype of* clause establishes the new created type as a subtype of other types in the hierarchy. The *compose_expression* and *validate_expression* are boolean expressions which conjuncted give the condition that a combination of supertype instances need to satisfy to compose a new DT object. The condition in *compose_expression* is evaluated only when a new instance of a DT is materialized and assigned a new OID. By contrast, the condition specified with the *validate_expression* is also evaluated each time a materialized instances is used in a query to the materialized object. The separation of the composition and validation expressions was motivated by the observation that data integration is often performed based on some functions that do not change over the lifetime of the instance (i.e. that are functionally dependent on the OIDs of the integrated instances). In these cases, it is not necessary to evaluate the full condition every time a DT instance is validated, but instead only the *validate_expression* is evaluated over the corresponding instances of the supertypes. The following example illustrates the intended use of these two clauses by defining three DTs from Figure 2:

```
create type Emp
 subtype of Person P, PayRecord PR
  compose ssn(P) = ssn(PR)
  validate  status(P) = 'working';

create type Sporty_Emp
 subtype of Person@SPORT_DB p, Emp e
  compose ssn(e) = adjust_ssn(socsecn(p));

create type Junior
 subtype of Sporty_Emp se
  validate age(se) > 26;
```

There is one instance of type *Emp* for each person for whom there is a pay record and the status is 'working'. Since the social security number does not change during the existence of a *Person* instance, the conditions involving the functions *ssn* and *socsecn* are placed in the *compose* clause in the definition of *Emp* and *Sporty_Emp*. On the other hand, the status and the age of a person can change and therefore the conditions over these functions are placed in the *validate* clauses.

The clauses *hide* and *properties*, which for brevity were not used in the examples, serve to list the functions of the supertypes not to be inherited by the newly defined DT, and to define new stored function, respectively.

4. Querying the Derived Types

The previous section presented the architecture of the view system for database mediation in AMOS. This section describes how the system evaluates queries over a type hierarchy extended with DTs. The presentation will concentrate on translation of queries to a correct and optimized object-calculus representation. The query decomposition and algebra optimization phases are not topics of this paper. Also, we will focus on processing of the queries over DTs defined by subtyping. In our current work, we are developing similar techniques for queries over the derived supertypes.

In the implementation presented in this section, as much as possible of the system support tasks are expressed by predicates incorporated in the calculus representation of the query. Many of these task traverse the type hierarchy and have common subtasks. The predicate representation allows this common subtasks to be identified and eliminated from the query. Beside this, overlaps between the user-defined and system-inserted predicates can be exploited to further reduce the number of predicates. Of particular interest in a view mechanism for mediation is to reduce the operations which cross the database boundary in communication with other databases.

Each DT in AMOS is implemented by an ordinary type named *implementation type*. The system automatically generates *coercion functions* over the implementation types to store the mappings between the materialized instances of each DT and its supertypes. All coercion functions are represented by the generic function *coerce*, overloaded on its argument and result. Coercion between instances of a DT and its indirect supertypes is done by composition of coercion functions. The coercion functions are not accessible by the user. They are manipulated by the system and used in other system-defined functions that are generated from the derived type definition. For each DT the system also generates three other system support functions: an *extent function*, a *validation function*, and a *materialization function*. The calculus generator analyses the query and, if the query is specified over DTs, inserts the functions and predicates needed to provide the required semantics. Later in this section, we will present how these transformations are performed for different classes of queries over DTs whose definitions contain a *subtype of* clause. First, an overview of the implementation of the subtyping from types in other AMOS mediators is presented.

4.1 Subtyping from Other Mediators

The subtypes that inherit from other AMOS systems make the basis for the mediation process. In Figure 2, an example was presented in which the type *Sporty_Emp* in the *Employee* database inherits from the type *Person* in the *Sport* database. The type hierarchy shown in Figure 2 is the user's view of the definition of the DTs in the example. In the implementation, for each distinct imported type (distinguished by the type name and the database name) a *proxy* type is created. All proxy types are subtypes of the type *Proxy* in the type hierarchy. Figure 4 illustrates this for

the definition of the DT *Sporty_Emp* type in the example in Figure 2. The type *P_Person* is a proxy for the type *Person* from the sport database.

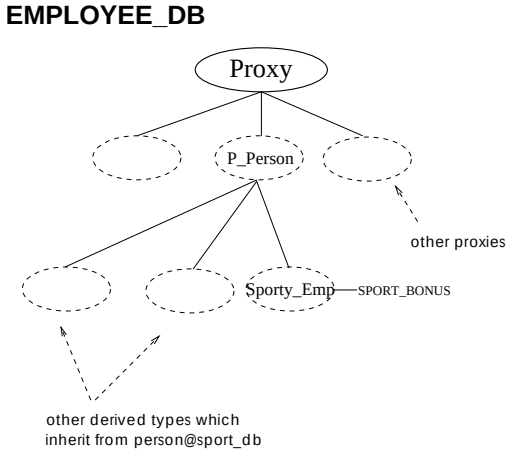

Figure 4. Placing the proxy types in the type hierarchy

After defining the proxy type, the system retrieves the functions over the imported type from the exporting mediator. For each function whose argument and result domains are system-defined types or types already represented by proxy types, the system defines a *proxy function*. The purpose of the proxy function is to provide information for local type checking of the queries calling functions defined in the other mediators.

Although the proxy functions and the proxy type extent functions are treated the same as the ordinary functions throughout the calculus oriented query processing steps described in this paper, they are not actually executed as the ordinary functions. The decomposition algorithm assembles them in groups and schedules them to be executed in other AMOS mediators. [2]

Over each proxy type, a system-defined stored function is generated that maps instances of the proxy type (i. e. proxy objects) to instances of type *foreign_oid*. This system type is used to represent the OIDs received from the other AMOS mediators when portions of query plans are evaluated there. Among the mediators, OIDs are transmitted and stored in their native format. The OIDs generated by an AMOS mediator are unique within the mediator itself. The system makes no effort to generate "universal OIDs" unique in all mediators present in the integration environment, like, for example, in the CORBA architecture [9]. In a CORBA environment, OIDs represent services and are designed to be transmitted alone. Therefore each OID contains all the information needed to identify its origin. In a bulk data processing environment, as the one described in this work, the OIDs are passed in larger collections having few different types and a common origin. Consequently, it is advantageous to condense the meta-information about the structure (types) and the origin of the transmitted OIDs with a transmission protocol. When a mediator receives OIDs from another mediator it stores them in their native format, while the meta-information is captured in the mediator's schema and the functions generated from the DT definitions. As the result of this kind of architecture, imported OIDs are stored in a mediator, but they cannot be interpreted there. The user does not have access to the imported OIDs, but only to the proxy (type) instances. The system uses the imported OIDs only in operations executed in the mediator where they originate from. The major benefits from this approach are a simple OID generation method, lower communication cost and lower storage overhead due to a smaller literal representation of the OIDs.

[2] In one pilot implementation, these functions were implemented to perform RPC-like shipping of the operands and function execution in other databases; in this case the calculus representation was directly translated to correct, but very inefficient execution plan.

4.2 DT Extent Function and Template

The extent function of each DT is a system-generated derived function. The general form of the extent function is:

```
CREATE FUNCTION dt_name() -> dt_name AS
  SELECT MAT(s1, s2, ... sn)
  FROM sut1 s1 , sut2 s2, ... ,sutn sn
  WHERE compose_expression AND
validate_expression
```

where *dt_name* is the name of the DT, *sut1* ... *sutn* are the supertypes from the *supertype of* clause, and $MAT_{<sut1,sut2,...sutn>\rightarrow dt\_name}$ is the materialization function for the DT *dt_name*. *Compose_expression* and *validate_expression* are copied from the DT definition. The function returns materialized instances of the newly created DT. As noted earlier the system tries to reduce the materialization of the DT instances whenever possible. Therefore, when processing the queries where no materialization is needed, instead of a complete extent function, an *extent template* (ET) is used. For each DT, the system generates an ET from the calculus representation of the extent function. It has a signature and a body. The signature contains a name, a list of *substitute variables* (SVs) and list of types associated with the SVs. The SVs are the variables used as arguments of the materialization predicate in the extent function ($s1 \ldots sn$ in the general form of the extent function above). There is one SV for each supertype of the DT. The body is a predicate template consisting of the extent function body with the materialization predicate removed.

The following example shows the ETs for the DTs *Sporty_Emp* and*Junior* and *Emp* in Figure 2:

signature:
$Textent\_sporty\_emp_{<P\_Person,emp>} : \_px, \_e$
body:
$\_px = Textent\_p\_person_{<foreign\_OID>} \wedge$
$\_e = Textent\_emp_{<person,payrec>} \wedge$
$sssn = socsec_{p\_person \rightarrow charstring}(\_px) \wedge$
$essn = ssn_{person \rightarrow integer}(\_e) \wedge$
$essn = adjust\_ssn_{charstring \rightarrow integer}(sssn)$

signature:
$Textent\_junior_{sporty\_emp} : \_se$
body:
$\_se = Textent\_sporty\_emp_{<p\_person,emp>} \wedge$
$a1 = age_{person \rightarrow integer}(\_se) \wedge$
$26 > a1$

signature:
$Textent\_emp_{<person,payrec>} : \_p, \_pr$
body:
$assn = ssn_{payrec \rightarrow integer}(\_pr) \wedge$
$assn = ssn_{person \rightarrow integer}(\_p) \wedge$
$'working' = status_{person \rightarrow charstring}(\_p)$

By convention, ET names begin with the $Textent$ prefix. The SV types are in the subscript of the template name, while the SVs are listed after the colon. An expression with a variable as a left-hand side and an ET as a right-hand side is named *ET declaration*. An ET declaration is inserted in the query for each variable declared with a DT. It asserts a type of a DT variable, analogous to the extent function of an ordinary type. When a DT is defined by subtyping from other DTs, its ET body can contain nested ET declarations, as is the case with the ETs $Textent\_sporty\_emp$ and $Textent\_junior$ in the example above.

An ET body contains predicates to assert that a combination of instances of the supertypes compose an instance of the DT. However, an ET cannot be evaluated as an extent function, and is used only during the calculus generation phase. In this phase, the incomplete calculus expression containing ET declarations is transformed to a complete calculus expression by a series of transformations performed until there are no more ET declarations. In a such transformation, an ET declaration is removed from the query if the declared variable can be typechecked by being used as a function argument of the same DT as the declared DT of the variable. Otherwise, an *ET expansion* is performed. During an ET expansion, first the ET declaration is substituted by the ET body. Then, each occurrence of the variable declared by this ET declaration as a function argument is substituted in the rest of the query by a SV in the ET signature having the same type or a supertype of the argument's type. This rule also applies when a variable is used to represent a result of a function. An ET expansion transforms a query over a DT into a query over its supertypes, avoiding materialization and run-time coercion.

The ET expansion process will be illustrated using an example query over the schema in Figure 2. The correctness and the termination of the the procedure can be proved based on the directed and acyclic properties of the inheritance hierarchy. Due to space limitations, this proof is not presented in this paper. In the following example, the query is first translated to an initial incomplete calculus expression given following the query:

select age(j), salary(j)
from Junior j
where hobby(j)='golf';

$\{ sal, a \mid$
$j = Textent\_junior_{Sporty\_Emp} \wedge$
$sal = salary_{payrec \rightarrow integer}(j) \wedge$
$a = age_{person \rightarrow integer}(j) \wedge$
$'golf' = hobby_{P\_Person \rightarrow charstring}(j)\}$

The ET declaration of the variable j is not removed because j is not used as argument or result of type *Junior* in any of the query functions. Next, the ET is expanded and all occurrences of j in the query body are substituted by the template variable $\_se$[3]. The expression produced by this expansion (the first below) contains an ET declaration with the ET $Textent\_sporty\_emp$. Analogous to the case with the j variable, this ET is also expanded yielding the second expression below:

$\{ sal, a \mid$
$\_se = Textent\_sporty\_emp_{<P\_Person,emp>} \wedge$
$a1 = age_{person \rightarrow integer}(\_se) \wedge$
$26 > a1 \wedge$
$sal = salary_{payrec \rightarrow integer}(\_se) \wedge$
$a = age_{person \rightarrow integer}(\_se) \wedge$
$'golf' = hobby_{P\_Person \rightarrow charstring}(\_se)\}$

$\{ sal, a \mid$
$\_px = P\_Person_{nil \rightarrow P\_Person}() \wedge$
$\_e = Textent\_emp_{<person,payrec>} \wedge$
$essn = ssn_{person \rightarrow integer}(\_e) \wedge$
$sssn = socsec_{P\_Person \rightarrow charstring}(\_px) \wedge$
$essn = adjust\_ssn_{charstring \rightarrow integer}(sssn) \wedge$
$a1 = age_{person \rightarrow integer}(\_e) \wedge$
$26 > a1 \wedge$
$sal = salary_{payrec \rightarrow integer}(\_e) \wedge$
$a = age_{person \rightarrow integer}(\_e) \wedge$
$'golf' = hobby_{P\_Person \rightarrow charstring}(\_px)\}$

[3]The expansion mechanism actually generates unique names for the template and the local variables each time a function or an ET is expanded.

In the $salary$ and age functions, the variable $\_se$ is substituted by the SV $\_e$ that corresponds to the type Emp through which these functions are inherited in $Sporty\_Emp$ (the type of $\_se$). On the other hand, in the $hobby$ function, $\_se$ is substituted by the variable $\_px$ since this function is inherited through the P_Person type.

Finally, the ET declaration of the the variable $\_e$ is expanded. After this expansion the query expression does not contain ET declarations:

$$
\begin{array}{ll}
\{\ sal, a\ \mid & \\
\_px = P\_Person_{nil \rightarrow P\_Person}() \wedge & (*) \\
sssn = socsec_{P\_Person \rightarrow charstring}(\_px) \wedge & (*) \\
essn = adjust\_ssn_{charstring \rightarrow integer}(sssn) \wedge & \\
essn = ssn_{person \rightarrow integer}(\_p) \wedge & (2) \\
a1 = age_{person \rightarrow integer}(\_p) \wedge & (1) \\
26 > a1 \wedge & \\
assn = ssn_{person \rightarrow integer}(\_p) \wedge & (2) \\
assn = ssn_{payrec \rightarrow integer}(\_pr) \wedge & \\
'working' = status_{person \rightarrow charstring}(\_p) \wedge & \\
sal = salary_{payrec \rightarrow integer}(\_pr) \wedge & \\
a = age_{person \rightarrow integer}(\_p) \wedge & (1) \\
'golf' = hobby_{P\_Person \rightarrow charstring}(\_px)\} & (*)
\end{array}
$$

The first nine predicates are result of the ET declaration expansions described above. The last three predicates originate in the original query.

The calculus optimizer further reduces the example expression by unifying pair-wise the predicates indicated by the same number on the far right (the re-write rule is described in [5]). In case (1) there is an overlap between the user-specified query predicates and the DT *Junior* validation expression. In case (2) the overlap is between the definitions of the DTs *Sporty_Emp* and *Emp*. The query calculus expression after the calculus optimization contains six system-inserted predicates. The result of the query optimization is then processed by the query decomposition algorithm which, in this example, combines the three predicates marked by (*) for execution in the sport database. There, the local optimizer will furthermore remove the typecheck predicate (the first query predicate). The only data transferred between the AMOS mediators will be the hobbies and the social security numbers of the relevant persons. This is clearly advantageous over a naive instance level processing in which for each instance a request is issued to get the values of requested data over the network

The transformations of the extent templates shown above reduce the need for run-time coercing. In the cases as in the example above where no function was evaluated over materialized DT instances, no coercion predicates are needed in the query.

4.3 Validation and Coercion of Materialized DT Instances

The previous example illustrated a query transformation where no function application needed materialized DT instances as arguments. The following example extends the query from the previous example with such a function application, by adding a query condition over the *sport_bonus* function defined in the DT *Sporty_Emp* (the underlined predicate):

```
select age(j), salary(j)
from Junior j
where hobby(j)='golf' and
      sport_bonus(j) > 100;
```

$$
\begin{array}{l}
\{\ sal, a\ \mid \\
j = Textent\_junior_{Sporty\_Emp} \wedge \\
\underline{b = sport\_bonus_{sporty\_emp \rightarrow integer}(j)\ \wedge\ b > 100 \wedge} \\
a = age_{person \rightarrow integer}(j) \wedge \\
sal = salary_{payrec \rightarrow integer}(j) \wedge \\
'golf' = hobby_{P\_Person \rightarrow charstring}(j)\}
\end{array}
$$

As in the previous example, first a reference to *Textent_junior* is inserted and expanded. The resulting query contains an ET declaration of the variable $\_se$ with the ET *Textent_sporty_emp*. Also, the variable j is substituted by the variable $\_se$ throughout the query. At this point, since the variable $\_se$ is used as an argument of the function $sport\_bonus_{sporty\_emp \rightarrow integer}$, the ET *Textent_sporty_emp* is not expanded, but instead removed. The variable $\_se$ in this case iterates only over the already materialized portion of the extent of $Sporty\_Emp$.

To produce a correct expression, the query expression resulting from the previous transformation needs to be extended with predicates to perform the coercion and validation of the materialized $Sporty\_Emp$ instances. The required result can be described as:

$$
\begin{array}{ll}
\{\ sal, a\ \mid & \\
\quad b = sport\_bonus_{sporty\_emp \rightarrow integer}(\_se) \wedge & \\
\quad b > 100 \wedge & \\
\quad \textbf{validate \_se}\ \wedge & (1) \\
\quad \textbf{coerce \_se to p of person}\ \wedge & (2) \\
\quad a = age_{person \rightarrow integer}(p) \wedge & \\
\quad a1 = age_{person \rightarrow integer}(p)\ \wedge\ 25 < a1 & \\
\quad \textbf{coerce \_se to pr of payrec} \wedge & (3) \\
\quad sal = salary_{payrec \rightarrow integer}(pr) \wedge & \\
\quad \textbf{coerce \_se to px of P\_Person} \wedge & (4) \\
\quad 'golf' = hobby_{P\_Person \rightarrow charstring}(px)\} &
\end{array}
$$

The lines in bold font give abstract descriptions of the operations that the system needs to add to the query. The numbers on the far right are added for reference purposes. The

predicates containing the $a1$ variable are inserted when the ET of the type $Junior$ is expanded.

The validation function assures that the corresponding instances of the supertypes are still present and valid in the data sources, and that the validation condition evaluated over this instances still holds. The general form of the validation function is:

```
CREATE FUNCTION validate_DT(DT obj)
 -> boolean AS
  SELECT TRUE
  FROM st1 st1_obj, st2 st2_obj, ...
  WHERE st1_obj = coerce(obj) AND
        validate_st1(st1_obj) AND
        st2_obj = coerce(obj) AND
        validate_st2(st2_obj) AND ...
        validate_predicate;
```

The function coerces the argument to each of the corresponding supertype instances, validates these instances, and then evaluates the validation condition. For example, the validation function for the DT *Emp* in Figure 2 is as follows:

```
CREATE FUNCTION validate_emp(emp obj)
 -> boolean
   SELECT TRUE
   FROM Person person_obj
   WHERE person_obj = coerce(obj) AND
         status(person_obj) = 'working';
```

The validation function of the proxy type instances performs a typecheck of the associated $foreign\_OID$ instances in the database they originate from. Therefore, the validate function contains a single proxy type typecheck predicate.

At this point we turn the attention back to the example from the beginning of this subsection. The system-inserted tasks described in the example require the following 10 predicates:

$$e = coerce_{sporty\_emp \rightarrow emp}(\_se) \ \wedge \quad (1)$$
$$p = coerce_{emp \rightarrow person}(e) \wedge$$
$$'working' = status_{person \rightarrow charstring}(p) \wedge$$
$$pi0 = coerce_{sporty\_emp \rightarrow P\_Person}(\_se) \wedge$$
$$pi0 = P\_Person_{nil \rightarrow P\_Person}() \wedge$$
$$e1 = coerce_{sporty\_emp \rightarrow emp}(\_se) \wedge \quad (2)$$
$$p = coerce_{emp \rightarrow person}(e1) \wedge$$
$$e2 = coerce_{sporty\_emp \rightarrow emp}(\_se) \wedge \quad (3)$$
$$pr = coerce_{emp \rightarrow payrec}(e2) \wedge$$
$$px = coerce_{sporty\_emp \rightarrow P\_Person}(\_se) \quad (4)$$

The numbers on the left match the predicate groups with the corresponding task in the previous description of the query. After inserting these predicates in the query the optimizer, by variable unification and typecheck removal, reduces the number of system inserted predicates from ten to six. In addition to this, the query optimizer removes one of the predicates referencing the age function. The resulting query after the query optimization is:

$$\{\ sal, a, b \mid$$
$$b = sport\_bonus_{sporty\_emp \rightarrow integer}(\_se) \wedge$$
$$b > 100 \wedge$$
$$e = coerce_{sporty\_emp \rightarrow emp}(\_se) \wedge$$
$$p = coerce_{emp \rightarrow person}(e) \ \wedge \ \wedge$$
$$'working' = status_{person \rightarrow charstring}(p) \wedge$$
$$a = age_{person \rightarrow integer}(p) \ \wedge \ 25 < a \wedge$$
$$pr = coerce_{emp \rightarrow payrec}(e) \wedge$$
$$sal = salary_{payrec \rightarrow integer}(pr) \wedge$$
$$px = coerce_{sporty\_emp \rightarrow P\_Person}(\_se) \wedge$$
$$'golf' = hobby_{P\_Person \rightarrow charstring}(px)\}$$

The only predicate not executed locally is the typecheck of the proxy objects of type *P_Person*. This predicate makes sure that objects for which proxy objects are created locally are still present in the sport database.

4.4 Materialization

The preceding examples demonstrated calculus generation and optimization for cases when no materialization was needed. This section briefly describes how the instances of the DTs are materialized.

DT instances are materialized if they are part of the query result or are used as an argument to a foreign function. Materialization of DT instances is performed by a materialization function, implemented as a foreign function. It takes as arguments instances of the DT supertypes and returns a new materialized DT instance. In the case when for the given arguments there is already a materialized DT instance, it is returned without creating a new one. The materialization functions are defined by the system as resolvents of the generic function *MAT*.

When instances represented by a DT variable are to be materialized, instead of extent templates for that variable, the whole extent function is inserted and expanded. The following example illustrates this process. The query in the example below materializes an instance of the DT *Manager*. The expanded object calculus generated for this query (shown following the query) contains two materialization predicates.

select m
from manager m
where name(m) = 'John'

$$\{\ m \mid$$
$$s = ssn_{person \rightarrow integer}(p) \wedge$$
$$s = ssn_{payrec \rightarrow integer}(pr) \wedge$$
$$'John' = name_{person \rightarrow charstring}(p) \wedge$$
$$'mng' = position_{payrc \rightarrow charstring}(pr) \wedge$$
$$e = MAT_{<person,payrec> \rightarrow emp}(p, pr) \wedge$$
$$m = MAT_{emp \rightarrow manager}(e)\ \}$$

5. Related Work

The work presented in this paper is related to research in the areas of OO views and database integration. This section references and briefly compares some representative examples in these areas with the work described in this paper.

The Multiview [15] OO view system provides multiple inheritance and a capacity-augmented view mechanism implemented with a technique called Object Slicing [13] that uses OID coercing in an inheritance hierarchy. However, it assumes active view maintenance and does not elaborate on the consequences of using this technique for integration of data in dislocated repositories. Furthermore, it does not use predicate-based implementation as described in this work. Other related OO view systems are described in [16] and [1].

The Remote-Exchange project at University of Southern California [6] uses a CDM similar to the one used in our work to establish a framework for instance and behavior sharing. However, it always uses late binding to choose between local and remote implementations of a function which is then called by an RPC for every single instance.

There are few research reports describing use of views mechanisms for data integration. The Multibase system [2] is also based on a derivative of the DAPLEX data model and uses function transformations for queries in a scenario similar to the supertyping scenario in this paper. Although this scenario was not the focus of the paper, some differences in the approaches can be identified. The data model used in Multibase does not contain the concept of OIDs. Furthermore, no materialization is used, which makes the coercion and validation techniques presented here not applicable.

The UNISQL [12, 11] system also provides views for database integration. The virtual classes (corresponding to the DTs) are organized in a separate class hierarchy. The virtual class instances inherit the OIDs from the corresponding instances in the ordinary classes, which does not allow definition of stored functions over virtual classes defined by multiple inheritance. There is no corresponding supertype integration mechanism, but rather a set of queries can be used to specify a virtual class as an union of other classes. This imposes relationships among the classes not included in the class hierarchy, resulting into two types of dependencies among the virtual classes.

6. Summary and Future Work

In this paper, we presented an overview of the design and the implementation of a passive mediation framework based on OO views in the AMOS system. The passive approach preserves the autonomy of the data sources and is suitable for mediation in environments with non-active, large volume data sources or data sources with high update frequencies.

The OO views mechanism is integrated in the AMOS inheritance mechanism by introducing derived types (DTs). The DTs are placed in the same type hierarchy along with the ordinary types. The instances of the DTs are derived from the instances of their super- or subtypes by declarative conditions specified in the DT definition. DT instances can be materialized (assigned OIDs), which allows the user to have locally stored data associated with them.

Queries over DTs are expanded by system-inserted predicates that perform the DT system support tasks needed to provide correct query results. The DTs system support is divided in three mechanisms: (i) providing consistency of queries over DTs; (ii) materialization of DT instances; and (iii) validation of the materialized DT instances. The system generates templates and functions which perform these tasks. During the calculus generation phase, the query is analyzed, and where needed, the appropriate functions/templates are inserted. The final calculus representation is generated by a series of transformations aimed to produce a correct and efficient query calculus expression. In these transformations, query consistency is achieved by extent template expansions and removals, and by coercion for the materialized DT instances; materialization is performed by including materialization predicates for selected query variables; DT instance validation is performed by inserting and expanding the validation function. The separation of the validation from extent generation (instance composition) gives smaller validation functions. The separation of the materialization from the extent generation allows selective materialization where only portions of the DTs extents are materialized.

The predicate expressions specifying the view support tasks describe relationships of the DTs in the type hierarchy and often have overlapping parts. The paper demonstrates how calculus-based query optimization can be used to remove redundant predicates introduced from the overlap among the system-inserted predicate expressions, and between the system-inserted and user-specified parts of the query. The calculus-based transformations and optimizations do not require cost function calculation and search space transitions which makes them simple to implement and inexpensive to perform.

The conclusion from this research is twofold. First, although the object orientation allows for mediation in which the functions are evaluated by some remote method invocation protocol, this is, from performance reason, unacceptable. There is an apparent need for a bulk-processing based query processor as the ones used in the relational

databases. Second, the multidatabase environment requires even greater optimization effort to achieve predictive performance for a wide range of queries. This reflects on the system architecture which has to be suitable for application of extensive optimization techniques.

Our current research shows that the concepts presented in this paper for the derived subtypes can be extended to DTs defined as explicit supertypes. The derived supertypes have the same basic functions, but they are implemented in a different way. For example, for the explicit supertypes, the coercion functions do key mappings and possible materialization, while the extent function are implemented over functions of the subtypes.

References

[1] E. Bertino: A View Mechanism for Object-Oriented Databases. In *Proc. 3rd Intl. Conference on Extending Database Technology (EDBT'92)*, Vienna, Austria, 1992.

[2] U. Dayal, H. Hwang: View Definition and Generalization for Database Integration in a Multidatabase System, In *IEEE Trans. on Software Eng.* Vol SE-10, No 6., (IEEE), November 1984.

[3] W. Du and M. Shan: Query Processing in Pegasus, In *Object-Oriented Multidatabase Systems*, O. Bukhres, A. Elmagarmid (eds.), Pretince Hall, Englewood Cliffs, NJ, 1996.

[4] G. Fahl, T. Risch, M. Sköld: AMOS - An Architecture for Active Mediators. In *Proc. Workshop on Next Generation Information Technologies and Systems (NGITS'93)*, Haifa, Israel, June 1993.

[5] G. Fahl, T. Risch: Query Processing over Object Views of Relational Data. To be published in *VLDB Journal*, November 1997.

[6] D. Fang, S. Ghandeharizadeh, D. McLeod and A. Si: The Design, Implementation, and Evaluation of an Object-Based Sharing Mechanism for Federated Database System. In*The 9th International Conference on Data Engineering (ICDE'93)*, (IEEE), Viena, Austria, April 1993.

[7] S. Flodin, T. Risch: Processing Object-Oriented Queries with Invertible Late Bound Functions, In *Proc. of the 1995 Conf. on Very Large Databases (VLDB'95)*, Zurich, Switzerland, 1995

[8] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio and Y. Zhuge: The Stanford Data Warehousing Project, *IEEE Data Engineering*, 18(2), pp. 40-48, June 1995.

[9] Object Management Group: The Common Object Request Broker: Architecture and Specification, Object Request Broker Task Force, 1993.

[10] J. Karlsson, S. Flodin, K. Orsborn, T. Risch, M. Sköld and M. Werner: AMOS User's Guide, available at *http://www.ida.liu.se/∼edslab*.

[11] W. Kelley, S. Gala, W. Kim, T. Reyes, B. Graham: Schema Architecture of the UNISQL/M Multidatabase System, In *Modern Database Systems - The Object Model, Interoperability, and Beyond*, W. Kim (ed.), ACM Press/ Addison-Wesley Publishing Company, New York, NY, 1995.

[12] W. Kim and W. Kelley: On View Support in Object-Oriented Database Systems, In *Modern Database Systems - The Object Model, Interoperability, and Beyond*, W. Kim (ed.), ACM Press/ Addison-Wesley Publishing Company, New York, NY, 1995.

[13] H. Kuno, Y. Ra and E. Rundensteiner: The Object-Slicing Technique: A Flexible Object Representation and Its Evaluation, University of Michigan Technical Report CSE-TR-241-95, 1995.

[14] W. Litwin, T. Risch: Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates. In *IEEE Transactions on Knowledge and Data Engineering* 4(6), pp. 517-528, 1992

[15] E. Rundensteiner, H. Kuno, Y. Ra, V. Crestana-Taube, M. Jones and P. Marron The MultiView project: object-oriented view technology and applications, In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD'96)*, ACM SIGMOD Record, Vol. 25, 2, pp. 555-563, ACM Press, June 1996.

[16] C. Santos: Design and Implementation of an Object-Oriented View Mechanism, GOODSTEP ESPRIT-III Tecnical Report, ESPRIT-III Project No. 6115

[17] D. Shipman: The Functional Data Model and the Data Language DAPLEX. ACM Transactions on Database Systems Vol 6 No. 1, (ACM), 1981.

[18] D. Straube, M. Özsu: Query Optimization and Execution Plan Generation in Object-Oriented Database Systems. *IEEE Transactions on Knowledge and Data Engineering* Vol 7, No. 2, pp 210-227 (IEEE), 1995.

[19] G Wiederhold: Mediators in the Architecture of Future Information Systems, *IEEE Computer*, (IEEE), March 1992.