Journal of Intelligent Information Systems 12, 165–190 (1999)

Functional Query Optimization over Object-Oriented Views for Data Integration

VANJA JOSIFOVSKI vanja@ida.liu.se
TORE RISCH torri@ida.liu.se
Laboratory for Engineering Databases, Linköping University, 58183 Linköping, Sweden

Editors: Peter M.D. Gray, Peter J.H. King and Larry Kevschberg

Abstract. AMOS is a mediator system that supports passive (non-intrusive) integration of data from heterogeneous and autonomous data sources. It is based on a functional data model and a declarative functional query language AMOSQL. Foreign data sources, e.g., relational databases, text files, or other types of data sources can be wrapped with AMOS mediators, making them accessible through AMOSQL. AMOS mediators can communicate among each other through the multi-database constructs of AMOSQL that allow definition of functional queries and OO views accessing other AMOS servers. The integrated views can contain both functions and types derived from the data sources. Furthermore, local data associated with these view definitions may be stored in the mediator database. This paper describes AMOS' multi-database query facilities and their optimization techniques. Calculus-based function transformations are used to generate minimal query expressions before the query decomposition and cost-based algebraic optimization steps take place. Object identifier (OID) generation is used for correctly representing derived objects in the mediators. A selective OID generation mechanism avoids overhead by generating in the mediator OIDs only for those derived objects that are either needed during the processing of a query or have associated local data in the mediator database. The validity of the derived objects that are assigned OIDs and the completeness of queries to the views are guaranteed by system generated predicates added to the queries.

Keywords: heterogeneous data integration, object-oriented views, query optimization

1. Introduction

The increasingly distributed modern computing environment often requires users to retrieve relevant data from many sources and present them in a comprehensible form. The user should then be provided with a location transparent and semantically coherent view of the data in the different data sources. In the *mediator* approach, described in (Wiederhold, 1992), data from multiple data sources can be accessed and merged through mediator software that exploits knowledge about the subsystems.

The purpose of the AMOS project is to develop and demonstrate a mediator architecture for supporting information systems where applications and users combine and analyze data from many different data sources. A data source can be a conventional database but also data exchange files, text files, programs that collect measurements, or even programs that perform computations and other services. The data sources are distributed over a communication network. Application areas for this kind of architecture include engineering, telecom, and decision support.

In the AMOS architecture the applications access the data sources through one or several *mediator databases*. A mediator database presents high-level abstractions (or views) of combinations of data from data sources using a *Common Data Model* (CDM). This makes the combined data accessible through high-level queries and views, and relieves the user as well as the application programmer from details of the data sources. Furthermore the mediator databases can also store local application-oriented data that is not tied to any particular data source, but rather to the mediator itself. For example, if the mediator extracts data for a set of customers from some company database, it is also desirable to be able to store data about the local sales to these customers locally in the mediator. The mediator views become *capacity augmenting views* (Rundensteiner et al., 1996) and we will describe how queries over such views are processed in AMOS.

The data model in AMOS, which is used both as a CDM for mediation and to store local data, is an extension of the DAPLEX (Shipman, 1981) functional data model with object-orientation and mediation primitives. Functional and OO views provide the user with a unified appearance of data in different data sources.

The mediation requires AMOS to manage views where types are defined in terms of meta-data of the data sources. This requires the query language of AMOS, AMOSQL, to have mediation primitives permitting types to be derived from other types. Analogously, the system also needs to manage OIDs where the OIDs are derived from data of the data sources. Furthermore, since AMOS also permits local data to be stored in the mediator it needs to both manage local OIDs and have means to mediate between local and derived objects. The processing of declarative queries in this environment requires several novel query-processing techniques to be described here. The paper elaborates the preliminary work in (Josifovski and Risch, 1998).

The approach to database mediation presented here is based on *passive* evaluation techniques, assuming no active functionality in the data sources. With the passive approach, the mediator database constructs complete and consistent answers to the queries over the views at the time when the queries are issued. Furthermore, our approach is based on describing integrated views using declarative functions. The system provides an efficient view support mechanism by combining the optimization of the view definitions with the user-specified query parts. By contrast, the *active approach* requires active database functionality in the data sources to notify the mediator when there are changes that influence view definitions. We show that our approach can also be used with the active approach with some minor modifications.

AMOS' query optimizer is applied when functions are defined. Ad hoc queries in AMOS are regarded as function definitions too. The paper describes query transformation techniques which, for a certain class of queries (function definitions), allow for a reduction of the number of function calls by applying calculus-based optimizations. The calculus-based optimizations remove redundant computations by merging system-specified and user-specified functions in the query. The cost based optimization executed later in the query processing is concerned with the order of the execution rather than the removal of redundant computations.

To integrate data from many data sources distributed over the network, many AMOS mediator servers can run in different locations and communicate with each other in a

client-server or peer-to-peer fashion. The cost of data communication between AMOS servers can however be high, and therefore the query optimizer tries to minimize the communication cost by query decomposition when constructing distributed execution plans. This paper concentrates on the calculus optimizations based on the functional paradigm and will only summarize the query decomposition process.

Our work should be contrasted with the traditional approach where the view support tasks are performed by procedural system code. In that approach, used, for example, in (Rundensteiner et al., 1996; Fang et al., 1993), most view support tasks are performed on an individual instance level, and the optimizations described here are therefore not applicable.

The paper is organized as follows. Section 2 gives an overview of the AMOS system. Section 3 introduces our architecture for database mediation based on functional and OO views. Section 4 describes the query transformation techniques used for processing queries over the views and how query rewriting reduces the queries. Section 5 discusses related work, and Section 6 summarizes.

2. Overview of the AMOS system

In the mediator approach of AMOS, a *translator* is developed for each kind of data source. A translator is an AMOS module that processes data from data sources of a particular class, such as relational databases (RDBs), object databases (ODBs), files, exchange formats, etc. A translator includes query primitives that translate data from its particular data model to the functional data model of AMOS. Using these translation primitives, OO views are defined for each data source. To process queries to these views a translator has knowledge of how to process functional queries that access its particular kind of data source.

For example, Fahl and Risch (1997) describes how relational data sources are wrapped using a relational AMOS translator. The relational translator contains knowledge about the capabilities of relational databases, such as their data model, their query processing capabilities, and how to efficiently map relational data to the CDM.

To resolve semantic heterogeneity between the translated views, integrated views are defined in which the semantic differences between data sources are resolved using mediator primitives that integrate semantically heterogeneous data. We will describe how to process queries to integrated views that integrate functional databases where semantically equivalent information is modeled differently.

The data model of AMOS has three basic constructs: *objects*, *types* (i.e., classes), and *functions*. Objects model entities in the domain of interest. An object can be classified into one or more types which makes the object *instances* of those types. 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* and *surrogate* types. The literal types, e.g., *int*, *real* and *string*, have a fixed (possibly infinite) extent 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 that

sets constraints on the classification of the objects. One example of such a constraint is: If an object is an instance of a type, then it 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 AMOS data model supports multiple inheritance, but requires an object to have a single most specific type.

The functions are divided by their implementations into three groups. The extent of a *stored* function is physically stored in the database. *Derived* functions are implemented in AMOS' query language AMOSQL. *Foreign* functions are implemented in some other programming language, e.g., Lisp or C++. Each foreign function can have several associated access paths and, to help the query processor, each access path has an associated cost and selectivity function (Litwin and Risch, 1992). This mechanism is called *multi-directional foreign functions*.

The AMOSQL query language is based on the OSQL (Lyngbaek et al., 1991) language with extensions of mediation primitives, multi-directional foreign functions (Litwin and Risch, 1992), overloading, late binding (Flodin and Risch, 1995), active rules (Sköld and Risch, 1996), etc. It contains data modeling constructs as well as querying constructs. The following example illustrates the data definition constructs of AMOSQL by defining a type *person* and three stored functions over this type: *hobby* returning a bag of character strings, *name* returning a single character string, and *parent* returning a bag of *person* objects:

```
create type person;
create function hobby(person) -> bag of string as stored;
create function name(person) -> string as stored;
create function parent(person) -> bag of person as stored;
. . .
```

The general syntax for AMOSQL queries is:

```
select <result>
   from <type declarations for local variables>
   where <condition>
```

The following example illustrates OO views in AMOSQL. Assuming the functions *parent*, *name* and *hobby* from above, it defines a derived function that retrieves the names of those children of a person having 'sailing' as a hobby:

```
create function sailing_children(person p) -> string as
  select name(c)
    from person c
    where parent(c) = p
          and hobby(c) = 'sailing';
```

The query optimizer optimizes the function body and associates with the function the optimized query execution plan. Since functions are used to represent properties of objects (i.e., methods) as e.g., *sailing_children*, the function bodies are always optimized assuming that the variables in the function arguments are *bound* while the other variables are initially

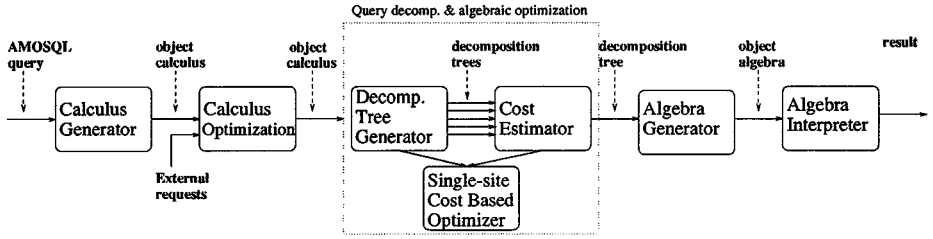

Figure 1. Query processing in AMOS.

unbound but will be assigned values when the function is executed. The term bound indicates that the variable has an assigned value before the execution of the function takes place.

The ad hoc queries in AMOSQL are treated as functions without arguments. For example, assume the following query that retrieves the names of the parents of all persons having 'sailing' as hobby (on the left):

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

```
create function query()->
  <person, string>
  as select p, name(parent(p))
     from person p
     where hobby(p) = 'sailing';
```

AMOS processes this query by generating an anonymous function with no arguments, `query()`, which is executed immediately and then discarded (on the right).

Figure 1, presents an overview of the query processing in AMOS. The first five steps, also called *query compilation* steps, translate the body of a function expressed in AMOSQL to a query execution plan which is stored with the function. To illustrate the query compilation we use the ad hoc query above.

From the parsed query tree, the calculus generator generates an *object calculus* expression. In the object calculus expressions, function symbols are annotated with their signatures. The left hand side of equality predicates in the calculus can be a single variable or a constant. It can also be a tuple of variables or constants when the function returns a tuple as a result. The right hand side of a predicate can be an unnested function call, a variable, or a constant. The head of the calculus query expression contains the result variables. Each derived function has an associated function body represented as a calculus expression. We note here that the calculus expressions can be viewed as type annotated list comprehensions. The calculus representation of the ad hoc query above is:

$$
\begin{aligned}
&\{p, nm \mid \\
&\ p = Person_{nil \to person}() \wedge \\
&\ pa = parent_{person \to person}(p) \wedge \\
&\ nm = name_{person \to string}(pa) \wedge \\
&\ 'sailing' = hobby_{person \to string}(p)\}
\end{aligned}
$$

The first predicate in the expression is inserted by the system to assert the type of the variable p. This *type check predicate* defines that the variable p is bound to one of the objects returned by the *extent function* for type *Person* named also *Person*(), which returns all the instances its type. The variables *nm* and *pa* are generated by the system.

AMOS supports overriding and overloading of functions on the types of their arguments and results, i.e., their signatures. Each function name refers to a *generic* function which can have several associated *type resolved* functions annotated with their signatures. During calculus generation each generic function call in a query is substituted by a type resolved one. Late binding is used for the calls which, due to polymorphism, cannot be resolved during query compilation (Flodin and Risch, 1995). However, our examples always assume early binding.

Next, the calculus optimizer applies rewrite rules to reduce the number of predicates. In the example, it removes the type check predicate:

$$
\begin{aligned}
&\{p, nm \mid \\
&\ pa = parent_{person \rightarrow person}(p) \wedge \\
&\ nm = name_{person \rightarrow string}(pa) \wedge \\
&\ 'sailing' = hobby_{person \rightarrow string}(p)\}
\end{aligned}
$$

The type check predicate can be removed because p is used in a stored function (*parent* or *hobby*) with an argument or result of type *person*. The referential integrity system of the stored functions constrain the instances of a stored function to be of correct type (Litwin and Risch, 1992). If there is no such constraining function the query processor will retain type check predicates to guarantee that derived functions return type correct values. If the argument types of the functions *parent* and *hobby* had been supertypes of *person*, the type check for p would have remained in the query. As will be shown, such type check removal is particularly important for multi-database queries where type checks often need to cross database boundaries and are expensive.

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 is to be executed by another AMOS server, the system will use primitives that allow for sending function definitions between the servers for local optimization and evaluation. These functions are generated by the query decomposer which is based on combinations of heuristic and dynamic programming enumeration strategies. Heuristics are used to divide the query (i.e., function) into a set of single-site functions. Then, dynamic programming is used to schedule the execution of these functions in order to obtain an execution plan distributed over multiple AMOS systems. Query decomposition is outside the scope of the paper and will not be elaborated.

The object calculus query representation is declarative and does not prescribe a certain evaluation order of the calculus predicates describing function calls. By contrast, the expressions in the query execution plans are represented in an object algebra (Fahl and Risch, 1997) having a well defined evaluation order.

An interested reader is referred to (Flodin et al., 1998) for a more detailed description of the AMOS system and to (Litwin and Risch, 1992; Fahl and Risch, 1997; Flodin and Risch, 1995) for more on 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 for relational databases. The design of a view mechanism in an OO environment is more complex in particular with regards to inheritance and object identity. Inheritance and views have common aims (i.e., data abstraction and code reuse) and therefore the two mechanisms must be combined in a semantically clear manner. Two important issues for OO view system design are the format of the OIDs of the view objects and their life span. Additional issues for views defined over data in multiple data sources are non-intrusive mechanisms for view maintenance, managing semantic heterogeneity, and representation of OIDs in a distributed environment.

3.1. Derived types

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 types and types imported 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 in some other systems as, e.g., the integrated views and column adding in the Pegasus system (Du and Shan, 1996), 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 with sporting information. The solid ovals represent ordinary types while the dashed ovals are DTs. Stored functions

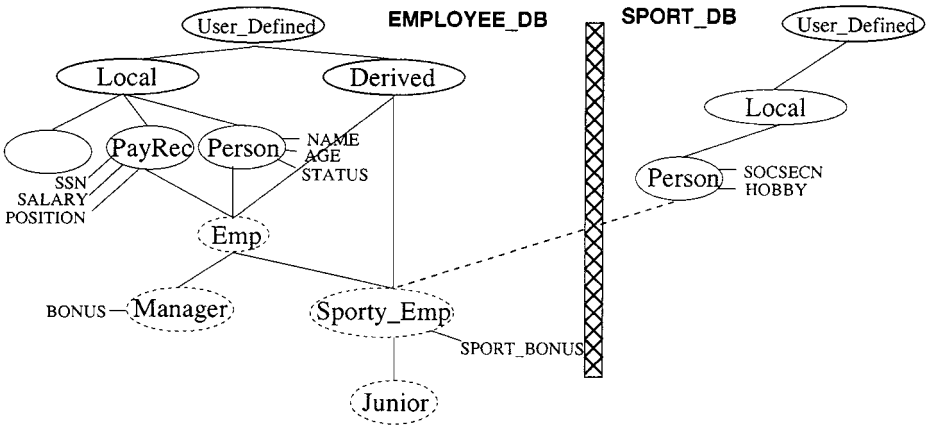

Figure 2. Integration by subtyping.

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 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 having a pay record. The DT *Manager* is a subtype of the DT *Emp* representing the employees for which the stored function *position* has the value 'Manager'. Type importation is done by subtyping from types in other database mediators, as illustrated by the DT *Sporty_Emp*. This DT is defined as a subtype of the local DT *Emp* and the type *Person* in the sport database. Its instances represent persons having an instance of type *Emp* in the employee database, and of type *Person* in the sport database.

The figure also illustrates some of our design choices. First, to be able to do data integration by subtyping, a DT needs 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 mechanism (Rundensteiner et al., 1996). DTs can be used in function definitions as ordinary types and any function can have DTs as argument or result domains.

3.2. Generation of OIDs for the DT instances

There are three basic choices for the format of OIDs representing DT instances. The first is to use the OIDs from the corresponding supertype objects (Santos, 1994). This is not suitable in our case because it is not compatible with multiple inheritance. The second alternative is to use a stored query expression instead of an OID and construct the required DT instances by evaluating this expression (Kelley et al., 1995). 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 DT instances (Rundensteiner et al., 1996). With this method, the same conceptual object (i.e., representing the same real world entity) is represented by different OIDs in different types. Therefore, to be able to evaluate inherited functions over the DT instances, their OIDs need to be mapped to the OIDs of the corresponding instances of the type over which a function was defined, by a process named *OID coercion* (in the text we use interchangeably the terms "OID coercion" and "instance coercion"). The cost of OID coercion is the main weakness of this approach. Nevertheless, we chose this approach for the following two reasons: First, the major cost of a query is in accessing the data sources and shipping data among the mediators, and not in the coercion. In AMOS, the hash tables used in the coercion are stored in a main-memory database which makes the coercion inexpensive. Second, expressing the coercion by predicates permits some query optimization that further reduces the coercion cost, as described in the next section.

Although the generation of OIDs for the DT instances allows for using the DTs as domains for function arguments and results, most queries over DTs require only a few or no OIDs and it would be a severe performance impairment to generate OIDs for the entire extents of all the DTs in each query. The OID generation cost includes the creation of a new OID and the storage of the coercion information in internal tables. To minimize this cost, and

to avoid unnecessary creation of OIDs, the query processor analyses the query to find out which query variables represent instances that need to be assigned OIDs. OID generation predicates are added only for query variables in the query result or used as arguments of foreign functions. Other queries are transformed so no OID generation is needed, as shown below. The query performance is thus not degraded by the OID generation mechanism. In queries requiring DT OIDs, these are generated selectively for those instances satisfying the rest of the query predicates, thus generating OIDs for only parts of the DT extents, in order to avoid unnecessary performance and storage overheads.

DT OIDs stored in local functions can be used in queries issued after their generation. Then the system has to assert that the instances they represent still comply with the declarative conditions stated in the DT definition, i.e., that they are still *valid*. Assuming non-active and autonomous data sources, the system has to add run-time checks in the queries to check the validity of those DT instances that are previously imported from external data sources and stored in local functions in the mediator. These validation checks must access the corresponding data sources to check the validity of the exported DT instances. If the query does not access imported DT OIDs stored locally, the instances are retrieved directly from the data sources and no validation is needed.

The validation checks could be removed if an active database capability is provided in data sources to notify the mediator when instances become invalid, similar to the interdatabase query invalidation mechanism described in (Grufman et al., 1997). The gain is faster query execution at the expense of slower updates. This is favourable for data sources with low update frequences.

The validity of a DT instance depends on the existence and validity of the corresponding supertype instances whose OIDs are stored in the coercion tables. 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 validation condition over a tuple of supertype OIDs, is consistent with the OO structure of the database, and is efficient to implement.

An instance is present in the mediator until it is used in a query where it fails the validation test. A garbage collection of the DT instances can be implemented to periodically run the validation test, deleting the instances not satisfying the 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 (Kim and Kelley, 1995). 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 the 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 each instance of the ordinary type has a corresponding instance in its supertypes (Kim and Kelley, 1995).

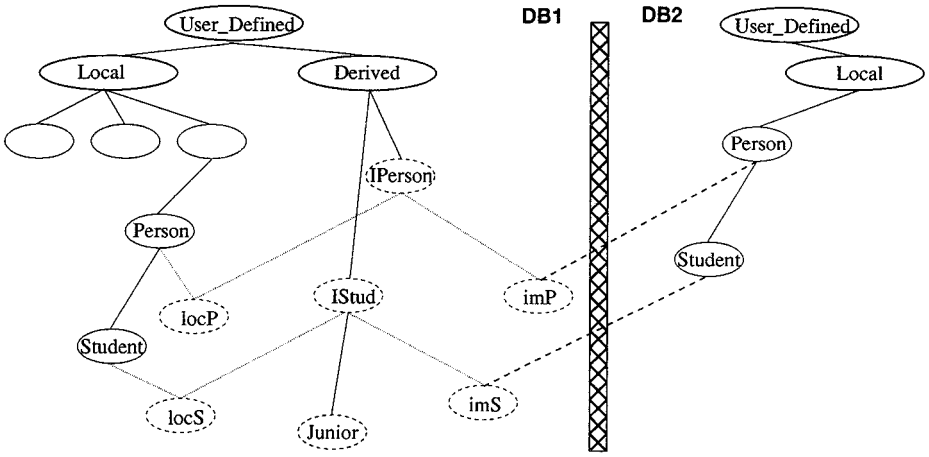

Figure 3. Integration by supertyping.

In figure 2 the derived part of the type hierarchy is constructed by subtyping. The AMOS integration framework also allows definition of DTs as explicit supertypes of other DTs. Although processing of queries over this kind of DTs is outside the scope of this paper, to complete the discussion on the integration framework, figure 3 presents an example of integration by supertyping. The 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* representing the instances from the types *Person* and *Student* in DB1 that 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* represents 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 integration framework can handle arbitrary schema heterogeneity by defining mappings using derived sub- and supertypes and derived functions.

3.4. Derived subtyping language constructs

For derived subtypes, AMOSQL has the following type definition construct:

```
CREATE TYPE type_name
   SUBTYPE OF sut1, sut2, ...
      COMPOSE compose_expression
      VALIDATE validate_expression
```

```
        HIDE fn1, fn2, ...
      KEY Type1 key1 [= exp1], ...
  END_TYPEDEF;
```

The *subtype of* clause establishes the DT as a subtype of other types in the hierarchy. The *compose_expression* and *validate_expression* are boolean expressions which when conjoined give the condition which a combination of supertype instances need to satisfy to compose a new DT object. The condition in *compose_expression* is evaluated only when an OID is generated for a new instance of a DT. By contrast, the condition specified with the *validate_expression* is also evaluated each time a query accesses OIDs of the DT stored locally in the mediator. The splitting of the composition and validation expressions was motivated by the observation that data integration is often performed based on some key 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. Alternatively, in order to avoid the burden of this splitting, the user could specify the condition as one expression, and then it could be separated by the system into composition and validation expressions based on the key information of the stored and the foreign functions used in the expression. The drawback of that approach is that it cannot detect conditions over non-key function that do not change during the existence of an instance (e.g., that the age of a person has passed some limit). In the following example, defining three of the DTs in figure 2, the condition expression is divided into the two parts:

```
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;
```

The function *adjust_ssn* converts a social security number stored in *SPORT_DB* to the format used in *EMPLOYEE_DB*. This can be any kind of function defined locally over integers and that returns integers.

There is one instance of type *Emp* for each person having a pay record and the status 'working'. Since the social security number does not change during the existence of a *Person*, the conditions involving the functions *ssn* and *socsecn* are in the *compose* clause of the definitions. 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 inherited by the DT, and to define new stored functions, respectively.

4. Querying derived types

DTs differ from ordinary types in a number of ways. First, the extents of DTs are not stored in the database as the extents of ordinary types, but are defined by declarative functions. Next, if a function inherited by a DT is called in a query, the system needs to coerce the argument DT OIDs to the corresponding OIDs of the supertype where the function is defined. Here, although the instances have different OIDs, they correspond to the same conceptual object. Finally, the system must check the validity of the DT OIDs stored in local functions, when used after their creation.

These differences make the queries over DTs more complex and time consuming than the queries over ordinary types. Naive evaluation of queries over the DTs, where the DTs are treated in the same way as the ordinary AMOS types, leads to a very inefficient query evaluation strategy. It would first retrieve the extents of the DTs in the query, generate OIDs for them, and then apply the selection condition of the query. Arguments to function calls used in the query must be coerced correctly.

An analysis of the execution plans showed that most of the overhead can be avoided by introducing query transformations to:

- avoid unnecessary OID generation,
- reduce the coercion to a minimum,
- allow for early application of selections in order to process only portions of the DT extents, and
- reuse OIDs stored in local functions instead of regenerating DT extents.

In order to achieve these goals, the OO views definitions are translated into system defined derived functions. The calculus generator analyses the query and, if the query is specified over DTs, inserts calls to these functions into the calculus representation of the query. Many OO view support tasks traverse the type hierarchy and have common subtasks. The predicate representation of the derived function bodies allows these common subtasks to be identified and eliminated from the query together with overlaps between user-defined and system-inserted predicates. Of particular interest in a view mechanism for data integration is to minimize operations that cross database boundaries in communication with other databases, or that access external data sources. Furthermore, the predicate-based view support approach allows selections from different query parts, such as user specified and DT subtyping conditions, to be unified, optimized together, and applied as close as possible to the data sources. When a data source supports selection application (e.g., relational databases), the selections can be applied in the data source itself (Fahl and Risch, 1997).

Although the common subexpression elimination mechanism allows for substantial reductions of queries over DTs, this alone does not remove all the redundances in the queries. Therefore, two additional DT specific transformations are introduced to further eliminate redundant computation: First, queries over DTs having all functions inherited from their supertypes are transformed into queries over their supertypes. This eliminates all OID generation and coercion, as will be shown. Second, for queries accessing locally stored functions over DTs the system tries to reuse the locally stored DT OIDs instead of naively regenerating the DT extent again. The presence of a locally stored function limits the instances of

interest to those stored in the function. However, since the OIDs stored in the local function were generated in previous transactions, and because of the autonomy of the data sources, the system needs to make sure that these OIDs still represent valid DT instances satisfying the validation condition of the DT. This validation could be avoided in some cases by, e.g., the distributed query invalidation mechanism of (Grufman et al., 1997).

In the rest of this section we will first describe how the DTs are modeled by AMOS types and derived functions. Then, the query transformations are described in detail using example queries entered in the *EMPLOYEE_DB* mediator, over the views defined in the previous section. The section concludes with an algorithm for the calculus transformations.

4.1. Modeling derived types and subtyping from other mediators

Each DT in AMOS is implemented by an ordinary local type named *implementation type*. The system automatically generates stored *coercion functions* over the implementation types to represent the mappings between those DT instances assigned OIDs and the tuples of corresponding instances of the DT's direct supertypes. All coercion functions are defined by the generic function *coerce*, overloaded on both its argument and result. Coercion between an instance 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 maintained by the system and used in system-defined functions generated from the DT definitions. For each DT the system generates three such functions: An *extent function*, a *validation function*, and an *OID generation function*. Informally, the extent function contains the subtyping condition and a call to the OID generation function. If invoked naively, it would generate all the tuples of the supertype objects that compose an object of the DT, and then invoke the OID generation function over these tuples to obtain OIDs for the DT instances. The OID generation function returns an already generated OID for each particular tuple of supertype instances, if such exists; otherwise, it creates a new OID and stores it in the coercion functions together with the tuple of supertype OIDs. Unlike the extent function, which contains the entire subtyping condition, the validation function contains only the DT validation condition. The validation function is used to check if a DT OID still represents a valid instance when used after its creation.

Subtypes inheriting from other AMOS systems make the basis for the data integration. 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. In the implementation, for each distinct imported type (distinguished by the type and database names) a corresponding *proxy* type is created. All proxy types are subtypes of the type *Proxy*. For example, there is a proxy type, *P_Person*, defined for the type *Person* from the sport database.

After defining a proxy type, the system retrieves the signatures of the functions defined over the type in the exporting mediator. If the argument and the result types of a remote function are known to the importing mediator (i.e., if they are system-defined or previously imported user-defined types) a local corresponding *proxy function* is defined. The proxy function has the same signature as the remote function and an empty body. Although the proxy functions and the proxy type extent functions are treated as ordinary functions throughout the calculus oriented query processing steps, they are not executed as ordinary

functions. The decomposition algorithm groups them, and schedules them for execution in other AMOS mediators. In the calculus-based query processing phases, they provide information for type checking and query transformation as described below.

For each proxy type, a system-defined stored function is generated that maps instances of the proxy type to instances of type *foreign_oid*. This system type is used to represent the OIDs received from other AMOS mediators when parts of query plans are evaluated there. The OIDs are transmitted and stored in their native format without origin or typing information added. The OIDs generated by an AMOS mediator are unique only within that mediator. The system makes no effort to generate "universal OIDs" unique in all mediators, like, for example, in the CORBA architecture (Object Management Group, 1993). In a CORBA environment, OIDs represent services and are designed to be transmitted alone. Therefore every OID contains all the information needed to identify its origin. In a bulk data processing environment such as ours, the OIDs are passed in large 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 the 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 a result of this kind of architecture, imported OIDs are stored in the mediator, but they cannot be interpreted there. The user does not have direct access to the imported OIDs, but only to their proxy type instances. The system uses the imported OIDs only in operations executed in the mediator where they originate from. The main benefits from this approach are simpler OID generation method, lower communication cost, and lower storage overhead due to smaller OIDs.

4.2. DT extent function and template

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

```
CREATE FUNCTION dt() -> dt AS
  SELECT genOID(s1, s2, ..., sn)
  FROM sut1 s1 , sut2 s2 ... sutn sn
  WHERE dt_compose_expression(s1, s2, ..., sn) AND
        dt_validate_expression(s1, s2, ..., sn);
```

where "dt" is the name of the DT, *sut1* . . . *sutn* are the supertypes from the *subtype of* clause, and $genOID_{<sut1,sut2,...sutn>\rightarrow dt}$ is the OID generation function for the DT. *Dt_compose-_expression* and *dt_validate_expression* are copied from the DT definition. If we represent these expressions as unexpanded derived functions the calculus form of the body of the extent function would be:

$$\{r \mid$$
$$s1 = sut1_{nil\rightarrow sut1}() \wedge$$
$$s2 = sut2_{nil\rightarrow sut2}() \wedge$$

$$
\begin{aligned}
&\ldots \\
&dt\_compose\_expression_{sut1,sut2\ldots sutn \rightarrow boolean}(s1, s2, s3, \ldots, sn) \wedge \\
&dt\_validate\_expression_{sut1,sut2\ldots sutn \rightarrow boolean}(s1, s2, s3, \ldots, sn) \wedge \\
&r = genOID_{sut1,sut2\ldots sutn \rightarrow dt}(s1, s2, s3, \ldots, sn)\}
\end{aligned}
$$

Now we consider the problem of calculating the result of a function inherited by a DT. To illustrate the steps needed for this we use the DT *Emp* and the function $name_{person \rightarrow string}$ from the example above, although the same principles apply for any DT and any inherited function. The query on the left below retrieves the names of all the employees; the calculus generated for this query is given on the right:

select name(se)
from Emp se;

$$
\begin{aligned}
&\{n \mid \\
&e = Emp() \wedge \\
&p = coerce_{emp \rightarrow person}(e) \\
&n = name_{person \rightarrow string}(p)\}
\end{aligned}
$$

The extent function *Emp*() produces the instances of the DT *Emp*. The stored function *name* stores OIDs of type *Person*. Since the instances of the DT *Emp* have OIDs different from the OIDs of the corresponding instances in the DT *Person*, they need to be coerced before applying the function *name* defined over *Person* instances. Expanding the *Emp*() extent function produces the following:

$$
\begin{aligned}
&\{n \mid \\
&p = Person_{nil \rightarrow person}() \wedge \\
&pr = PayRec_{nil \rightarrow payrec}() \wedge \\
&emp\_compose\_expression_{<person,payrec> \rightarrow boolean}(p, pr) \wedge \\
&emp\_validate\_expression_{<person,payrec> \rightarrow boolean}(p, pr) \wedge \\
&p = coerce_{emp \rightarrow person}(e) \wedge \\
&n = name_{person \rightarrow string}(p) \wedge \\
&e = genOID_{<person,payrec> \rightarrow emp}(p, pr)\}
\end{aligned}
$$

Notice that this query can be simplified by removing calls to the OID generation and coercion functions since the variable *e* is not used in the result.

In this simple example it is easy to spot and remove the unnecessary predicates. In a more elaborate example with several nested DT extent and coercion functions it would be difficult to do these removals. Therefore, for this type of optimization we have developed an approach in which the optimized query is generated by a set of transformations from the initial query calculus representation. During these transformations, instead of a complete extent function, an *extent template* (ET) is used. For each DT, an ET is generated from the calculus representation of the extent function. ETs have signatures and bodies. 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 OID generation function

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 without the OID generation predicate.

The term 'template' is used instead of 'function' because the ETs do not satisfy all the formal requirements to be classified as functions. Templates are used only for function transformations and have only calculus representations which cannot be executed. Also, the template expansion rules differ from the rules used for function expansion. The following example shows the ETs for the DTs *Sporty_Emp* and *Junior* and *Emp* in figure 2:

signature:
$ET\_sporty\_emp_{<P\_Person,emp>} : \_px, \_e$
body:
$\_px = P\_Person_{nil \rightarrow P\_Person}() \wedge$
$\_e = ET\_emp_{<person,payrec>} \wedge$
$sssn = socsec_{P\_Person \rightarrow string}(\_px) \wedge$
$essn = ssn_{person \rightarrow int}(\_e) \wedge$
$essn = adjust\_ssn_{string \rightarrow int}(sssn)$

signature:
$ET\_junior_{sporty\_emp} : \_se$
body:
$\_se = ET\_sporty\_emp_{<P\_Person,emp>} \wedge$
$a1 = age_{person \rightarrow int}(\_se) \wedge$
$26 > a1$

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

By convention, ET names begin with the *ET* prefix. Each template name is subscripted with the SV types, while the SVs are listed after the colon. An expression with a variable as the left-hand side and an ET as a right-hand side is named *ET declaration*. An ET declaration is added to the query for each variable declared with a DT. It asserts the type of a DT variable, analogous to the extent function of the ordinary types. When a DT is defined by subtyping from other DTs, its ET body can contain nested ET declarations, as for *ET_sporty_emp* and *ET_junior* above.

The ET body contains predicates to assert that a tuple of instances of the supertypes composes an instance of the DT. Because the ETs are not complete functions, a calculus expression containing ETs is considered *incomplete*. In the calculus generation 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 such a transformation, an ET declaration of a variable is removed from the query if the variable can be type checked by being used as a function argument of the same DT. Otherwise, *ET expansion* is performed. During ET expansion, first the ET declaration is substituted by the ET body. Then, each occurrence of the variable declared by the ET declaration is substituted in *the rest of the query predicates* by a SV in the ET signature having the same type or a supertype of the argument's type. An ET expansion transforms a query over a DT into a query over its supertypes, thus avoiding OID generation and run-time coercion. Notice that this kind of variable substitution differs from the substitution in normal function expansion where the argument and result variables in the function body are substituted to match the parameters.

The ET expansion process is illustrated through the example query below on the left over the schema in figure 2. It is first translated to an incomplete calculus expression given below on the right:

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

$$\{sal, a \mid \\ j = ET\_junior_{Sporty\_Emp} \wedge \\ sal = salary_{payrec \to int}(j) \wedge \\ a = age_{person \to int}(j) \wedge \\ 'golf' = hobby_{P\_Person \to string}(j)\}$$

The ET declaration of the variable j is not removed because j is not used as argument or result of type *Junior* in any function in the query. Therefore, this ET is expanded and all occurrences of j in the query body are substituted by the template variable *_se* in *ET_sporty_emp*. The expression produced by this expansion (on the left below) contains an ET declaration *ET_sporty_emp*. Analogous to the variable j ET declaration, this ET is also expanded yielding the expression on the right:

$$\{sal, a \mid \\ \_se = ET\_sporty\_emp_{<P\_Person, emp>} \wedge \\ a1 = age_{person \to int}(\_se) \wedge \\ 26 > a1 \wedge \\ sal = salary_{payrec \to int}(\_se) \wedge \\ a = age_{person \to int}(\_se) \wedge \\ 'golf' = hobby_{P\_Person \to string}(\_se)\}$$

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

In the *salary* and *age* functions, the variable *_se* of type *Sporty_Emp* is substituted by the SV *_e* of type *Emp* through which these functions are inherited in *Sporty_Emp*. By contrast, 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 variable *_e* is expanded. After this expansion the query expression does not contain any ET declarations:

$$\{sal, a \mid \\ \_px = P\_Person_{nil \to P\_Person}() \wedge \qquad (*) \\ assn = ssn_{person \to int}(\_p) \wedge \qquad (2) \\ assn = ssn_{payrec \to int}(\_pr) \wedge \\ 'working' = status_{person \to string}(\_p) \wedge \\ sal = salary_{payrec \to int}(\_pr) \wedge \\ sssn = socsec_{P\_Person \to string}(\_px) \wedge \qquad (*)$$

$$
\begin{aligned}
& essn = adjust\_ssn_{string \rightarrow int}(sssn) \wedge \\
& essn = ssn_{person \rightarrow int}(\_p) \wedge && (2) \\
& a1 = age_{person \rightarrow int}(\_p) \wedge && (1) \\
& 26 > a1 \wedge \\
& a = age_{person \rightarrow int}(\_p) \wedge && (1) \\
& 'golf' = hobby_{P\_Person \rightarrow string}(\_px)\} && (*)
\end{aligned}
$$

The first nine predicates are results of ET declaration expansions. 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 (Fahl and Risch, 1997). In case (1) there is an overlap between the user-specified query predicates and the validation expression of DT *Junior*. In case (2) the definitions of the DTs *Sporty_Emp* and *Emp* overlap. The query calculus expression now 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 further remove the type check predicate (the first predicate) since it has the needed information to deduce its redundancy. The queries produced by the decomposer in the two mediators are:

in EMPLOYEE_DB

$$
\begin{aligned}
& \{sal, a, sssn \mid \\
& assn = adjust\_ssn_{string \rightarrow int}(sssn) \wedge \\
& assn = ssn_{person \rightarrow int}(\_p) \wedge \\
& assn = ssn_{payrec \rightarrow int}(\_pr) \wedge \\
& 'working' = status_{person \rightarrow string}(\_p) \wedge \\
& sal = salary_{payrec \rightarrow int}(\_pr) \wedge \\
& a = age_{person \rightarrow int}(\_p) \wedge 26 > a\}
\end{aligned}
$$

in SPORT_DB

$$
\begin{aligned}
& \{sssn \mid \\
& sssn = socsec_{Person \rightarrow string}(\_px) \wedge \\
& 'golf' = hobby_{Person \rightarrow string}(\_px)\}
\end{aligned}
$$

The queries are executed in each of the mediators and then an equi-join over *sssn* is performed in the site determined by the query decomposer, based on the costs of execution and data transfer. The only data transferred between the mediators will be the set of social security numbers of the relevant persons, thereby avoiding generation of OIDs for the queried types.

The transformations of the extent templates shown above reduce the need for run-time coercing. In this example, where the query does not return OIDs and is not evaluated over local functions storing DT OIDs, no coercion or OID generation predicates are needed in the final query. By modeling the extent generation by predicates these predicates are unified with user specified selections which further reduces the processing.

4.3. Generation of OIDs for DT instances

The preceding subsection demonstrated calculus generation and optimization where the generation of OIDs for the DT instances can be avoided all together. This subsection briefly

describes how the DT instances are assigned OIDs in queries requiring this. Let's consider the following *set* command:

```
set sport_bonus(e) = 1000 from emp e where salary(emp) > 1000;
```

Here, the system first retrieves OIDs of type *Emp*, and then stores them with the bonus in the locally stored function *sport_bonus*. The generation of DT OIDs in this update query cannot be avoided.

An OID is generated for a DT instance if it is a part of the query result or used as an argument to a foreign function. OID generation functions are implemented as system generated foreign functions taking as arguments a tuple of DT supertype OIDs and returning a DT OID. If for the given arguments there is an already generated OID, it is returned without creating a new one. The OID generation functions are defined by the system as resolvents of the overloaded function *genOID*.

When, a calculus variable ranges over DT instances which are assigned an OID, the extent template defining this variable is replaced with the expanded extent function. The following example illustrates this process. The query on the left returns an instance of the DT *Manager*. The expanded object calculus generated for this query (shown on the right) contains two OID generation predicates. When an OID for a DT instance is generated, the OIDs of the corresponding instances in the derived supertypes need to be generated too. Therefore, in the example, the system also inserts an OID generation predicate for the DT *Emp*.

```
                          {m |
select m into :john        s = ssn_{person→int}(p) ∧
from manager m             s = ssn_{payrec→int}(pr) ∧
where name(m) = 'John'     'John' = name_{person→string}(p) ∧
                           'Manager' = position_{payrec→string}(pr) ∧
                           e = genOID_{<person,payrec>→emp}(p, pr) ∧
                           m = genOID_{emp→manager}(e)}
```

The *into* clause stores the query result in an AMOSQL variable.

To limit the OID generation to only the requested DT instances, the OID generation predicates should appear late in the final query execution plan after query conditions restricting the number of generated OIDs. The optimizer is aware of this, and after performing the cost based optimization it moves the OID generating expressions to the end of the query execution plan, preserving their relative order. Because the displaced expressions have low cost and selectivity 1, this transformation does not affect the overall query cost. This strategy is applicable to queries where OIDs are generated for DT instances in the query result, as in the example above. When the generated OIDs are used in some foreign functions, more elaborate interactions between the calculus generator and the algebra generator are required. That mechanism is not described in this paper.

4.4. Processing of queries using locally stored functions

As shown above, instances of a DT from a data source can be assigned OIDs and stored in local functions over the DT. These stored functions can be later referenced in user queries.

Then, because the data in the data source can change without the control of the mediator, DT OIDs retrieved from the locally stored functions need to be validated. Notice however that no action is needed when new instances are added in the data sources, since these new instances must be first stored in a local function in the mediator before any validation is needed. For example, if a person takes up golfing and thus becomes a *Sporty_Emp* that person's OID need not be validated until it is stored in a local function. Furthermore, the fact that the locally stored functions are cheap to access, and most often store only portions of the DT extent, can be used by the optimizer to produce plans operating only over the DT instances stored in these functions instead of the entire DT extent.

To illustrate the processing of queries with locally stored functions over DTs, we extend the example from Section 4.2 with a predicate (underlined) over the locally stored function *sport_bonus*, defined over the instances of the DT *Sporty_Emp*:

	$\{a, sal \mid$
select age(j), salary(j)	$j = ET\_junior_{Sporty\_Emp} \wedge$
from Junior j	$\underline{b = sport\_bonus_{sporty\_emp \rightarrow int}(j) \wedge b > 100 \wedge}$
where hobby(j) = 'golf' and	$a = age_{person \rightarrow int}(j) \wedge$
<u>sport_bonus(j) > 100;</u>	$sal = salary_{payrec \rightarrow int}(j) \wedge$
	$'golf' = hobby_{P\_Person \rightarrow string}(j)\}$

As in the previous example, first a reference to *ET_junior* is inserted and expanded. The resulting query contains an ET declaration of the variable *_se* with *ET_sporty_emp*. Furthermore, 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 $sport\_bonus_{sporty\_emp \rightarrow int}$, *ET_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*, stored in $sport\_bonus_{sporty\_emp \rightarrow int}$.

For a correct expression, the transformed query expression needs to be extended with predicates to perform the coercion and validation of the instance OIDs of *Sporty_Emp*. This can be described as:

$\{a, sal \mid$
$b = sport\_bonus_{sporty\_emp \rightarrow int}(\_se) \wedge b > 100 \wedge$
validate _se $\wedge$ (1)
coerce _se to p of person $\wedge$ (2)
$a = age_{person \rightarrow int}(p) \wedge$
$a1 = age_{person \rightarrow int}(p) \wedge 26 > a1 \wedge$
coerce _se to pr of payrec $\wedge$ (3)
$sal = salary_{payrec \rightarrow int}(pr) \wedge$
coerce _se to px of P_Person $\wedge$ (4)
$'golf' = hobby_{P\_Person \rightarrow string}(px)\}$

The lines in bold give abstract descriptions of the operations added by the system. The numbers on the far right are for reference purposes. The predicates containing the variable $a1$ are inserted when the ET of 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 these instances still holds. Its general form is:

```
CREATE FUNCTION validate_DT(DT obj) -> boolean AS
  SELECT TRUE
  FROM sut1 st1, sut2 st2, ...
  WHERE st1 = coerce(obj) AND
        validate_st1(st1) AND
        st2 = coerce(obj) AND
        validate_st2(st2) 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 e) -> boolean
   SELECT TRUE
   FROM Person p, Payrec pr
   WHERE p = coerce(e) AND status(e) = 'working' AND pr = coerce(e);
```

The validation function of a proxy type performs a check if the corresponding *foreign_OID* instance exists in the database it originates from. This is implemented by a single type check predicate.

The coercion and validation in the example above require the following 11 predicates to be inserted in the query:

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

The numbers on the left match the predicate groups with the corresponding task in the previous query. After inserting these predicates in the query, the optimizer, by predicate unification and type check removal, reduces the number of system inserted predicates from 11 to 6. In addition to this, the query optimizer removes one of the calls to the *age* function.

The resulting query is:

$$
\begin{aligned}
&\{a, sal \mid \\
&\ b = sport\_bonus_{sporty\_emp \rightarrow int}(\_se) \wedge b > 100 \ \wedge \\
&\ e = coerce_{sporty\_emp \rightarrow emp}(\_se) \wedge \\
&\ p = coerce_{emp \rightarrow person}(e) \wedge 'working' = status_{person \rightarrow string}(p) \wedge \\
&\ a = age_{person \rightarrow int}(p) \wedge 26 > a \ \wedge \\
&\ pr = coerce_{emp \rightarrow payrec}(e) \wedge sal = salary_{payrec \rightarrow int}(pr) \wedge \\
&\ px = coerce_{sporty\_emp \rightarrow P\_Person}(\_se) \wedge \\
&\ px = P\_Person_{nil \rightarrow P\_Person}() \wedge 'golf' = hobby_{P\_Person \rightarrow string}(px)\}
\end{aligned}
$$

The query decomposer will divide the query predicates into two functions: one executed in *EMPLOYEE_DB* and the other in *SPORT_DB*. The *EMPLOYEE_DB* function contains all the predicates except the last two. The function in *SPORT_DB* is compiled from the last two predicates and the typecheck is removed by the optimizer (the *EMPLOYEE_DB* function below is shortened for brevity):

in EMPLOYEE_DB

$$
\begin{aligned}
&\{a, sal, px \mid \\
&\ b = sport\_bonus_{sporty\_emp \rightarrow int}(\_se) \wedge \\
&\ \ldots \\
&\ px = coerce_{sporty\_emp \rightarrow P\_Person}(\_se)\}
\end{aligned}
$$

in SPORT_DB

$$
\begin{aligned}
&\{px \mid \\
&\ 'golf' = hobby_{Person \rightarrow string}(px)\}
\end{aligned}
$$

Notice that in this case OIDs are shipped from one mediator to another. Assuming that the function *sport_bonus* in *EMPLOYEE_DB* has a smaller extent than the function *hobby* in *SPORT_DB*, the decomposer will generate a schedule in which the function on the left above is executed first and the stored OIDs are shipped to *SPORT_DB*. There, the function on the right is executed, performing an equi-semi-join of the shipped OIDs with the function *hobby*.

4.5. The transformation algorithm

We conclude the discussion in this section with an algorithm for the described transformations. The input of the algorithm is a conjunction of predicates and a list of result variables. The output is a predicate in which all the DT extent functions have been transformed or expanded. The algorithm assumes that the input predicate is a conjunction of simple (non-derived) predicates and DT extent functions. Nevertheless, it can easily be expanded to predicates containing nested disjunctions and derived predicates. Also, single argument functions are assumed to simplify the presentation.

The following functions are assumed to be predefined: *et_body*(*dt*) returns the body of the extent template of *dt*; *et_sv*(*dt*) returns the substitution variables from the signature of the extent template of *dt*; *type*(*var*) returns the type of a calculus variable; *expand_function*

substitutes a function call with its already expanded function body; the '$<$' and '$\leq$' operators represent subtype/supertype comparison; the $\cup$ operator is used for appending conjunctions of predicates and adding a predicate to a conjunction.

```
1.  expand_DT_extent_functions( input: P, resVars; output: PR)
2.  oidGen := resVars;
3.  PR := P;
4.  while ∃ J ∈ PR : J ≡ (X = dt_nil→dt()) ∧ dt() is extent func. of the DT dt
5.    REST := PR − dt_nil→dt();
6.    if X ∈ oidGen do
7.      oidGen := oidGen ∪ et_sv(dt);        /* generate OIDs for the supertypes */
8.      PR := expand_function(dt_nil→dt()) ∪ REST;
9.    else
10.     if ∃J ∈ REST : J ≡ ( f_at(X)) ∧ at ≤ dt ∧ f_at is stored function then
11.       PR := expand_function(validate_at(X)) ∪ REST;
12.     else
13.       for each R ∈ REST
14.         if R ≡ (Q_bt(X)) then        /* R is over the variable X */
15.           PR := PR ∪ T : T ≡ (Q_bt(Y)) ∧ Y ∈ et_sv(dt) ∧ type(Y) ≤ bt;
16.         else
17.           PR := PR ∪ R;
18.         end if
19.       end for each
20.       PR := PR ∪ et_body_type→predicate(dt)
21.     end if
22.   end if
23. end while
```

The **while** loop is executed until there are no more DT extent functions in the predicate. For a chosen DT extent function, the first **if** checks if the DT variable belongs to the set of variables representing instances that are to be assigned OIDs. If so, the DT extent function is substituted with its body and the variables representing instances of the supertypes are added to the list of types for which OIDs are generated. Else, if there is a predicate containing a locally stored function over the DT *dt* in *PR*, then the validation function is inserted and expanded; otherwise the predicate is traversed, all occurrences of the variable X are substituted with the supertype variables, and the template body is appended.

5. Related work

The work presented in this paper is related to research in the areas of OO query processing (Straube and Özsu, 1995), OO views and database integration. This section references and briefly compares some representative research in these areas with our work.

The Multiview (Rundensteiner et al., 1996) OO view system provides multiple inheritance and a capacity-augmented view mechanism implemented with a technique called Object

Slicing (Kuno et al., 1995) using OID coercion 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 autonomous and dislocated repositories. Furthermore, it does not use a predicate-based implementation so our query optimization methods do not apply. Other similar OO view systems are described in (Santos, 1994; Bertino, 1992).

The Remote-Exchange project at University of Southern California (Fang et al., 1993) uses a CDM similar to ours for instance and behavior sharing. Three dimensions of freedom are explored for function application in a federated database environment: the location of the function (local or remote), the location of the arguments (local or remote) and the type of the function (stored or computed). Each case is elaborated and an abstract implementation is described. Most of the cases correspond to the ones present in AMOS, although the terminology differs considerably. An important disadvantage is that late binding is always used to choose between local and remote implementations of a function which is then called by an RPC for every single instance. Another performance degradation is caused by the size of the surrogate identifiers for remote instances which are 300 bytes long and contain all the information needed to perform the remote function evaluation over each instance.

There are few research reports describing the use of OO view mechanisms for data integration. The Multibase system (Dayal and Hwang, 1984) 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 that scenario was not the focus of the paper, we can note that an important difference between the systems is that the data model used in Multibase does not contain the concept of OIDs and therefore the coercion and validation techniques presented here are not applicable.

The UNISQL (Kim and Kelley, 1995; Kelley et al., 1995) 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 prohibits 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 in two types of dependencies among the virtual classes.

Finally, advanced commercial products have emerged lately (Carey et al., 1998), moving in the directions described in this paper.

6. Summary

An overview was presented of the query transformation techniques used in the implementation of a passive mediation framework based on functional queries and OO views. The passive approach preserves the autonomy of the data sources and is suitable for mediation in environments where data sources are autonomous, non-active, have large data volumes, or have high update frequencies.

The OO views mechanism is integrated into the inheritance mechanism by introducing derived types (DTs). The DTs are placed in the same type hierarchy as the ordinary types. The instances of the DTs are derived from the instances of their super- or subtypes

by declarative functions specified in the DT definitions. DT instances are assigned OIDs, which allows the user to have locally stored data associated with them.

Queries over DTs are expanded by including system-inserted predicates that perform the DT system support tasks. The DTs system support is covered in three mechanisms: (i) providing consistency of queries over DTs; (ii) generation of OIDs for the DT instances; and (iii) validation of the DT instances with assigned OIDs. The system generates templates and functions to 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 optimized coercion of local DT OIDs; OID generation is performed by including OID generation functions for selected query variables; DT instance validation is performed by inserting and expanding the validation functions. The separation of the validation from extent generation (instance composition) gives smaller validation functions. The separation of the OID generation from the extent generation allows selective generation of DT instance OIDs where only portions of the DT extents are materialized locally.

The functions 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 computations introduced from the overlap among the system-inserted expressions, and between the system-inserted and user-specified parts of the query. The calculus-based transformations and optimizations do not require cost calculations and search space transitions making them simple to implement and inexpensive to perform.

The following conclusions can be drawn: First, although traditional object orientation allows for mediation by some remote method invocation protocol, its performance can be unacceptable. There is an apparent need for set-oriented query processing as used in the relational databases. Second, the multidatabase environment requires even greater optimization efforts to achieve predictive performance for a wide range of queries. Third, describing type hierarchies and semantic heterogeneity using declarative functions and a functional Common Data Model provides many opportunities for the extensive query optimization needed in an OO mediation framework.

The AMOS system is implemented on Windows NT/95 platform using TCP/IP for the communication. In our current work, we have expanded the concepts presented in this paper from the derived subtypes to include derived supertypes. Also, a multidatabase query processing engine is designed and implemented to process the queries after the calculus optimization phase. Our future work will include: research on wrapper design and implementation, parallel and asynchronous execution strategies, multidatabase query processing in presence of replication and limited availability, etc.

Acknowledgments

This work has been supported by the ECSEL program of the Swedish Foundation for Strategic Research (SSF).

References

Bertino, E. (1992). A view mechanism for object-oriented databases. *Third Intl. Conf. on Extending Database Technology (EDBT'92)*. Vienna, Austria.

Carey, M., Haas, L., Kleewein, J., and Reinwald, B. (1998). Data Access Interoperability in the IBM Database Family. *IEEE Data Engineering*, 21(3), 4–11.

Dayal, U. and Hwang, H. (1984). View Definition and Generalization for Database Integration in a Multidatabase System. *IEEE Trans. on Software Eng.*, 10(6).

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

Fahl, G. and Risch, T. (1997). Query Processing over Object Views of Relational Data. *The VLDB Journal*, 6(4), 261–281.

Fang, D., Ghandeharizadeh, S., McLeod, D., and Si, A. (1993). The design, implementation, and evaluation of an object-based sharing mechanism for federated database system. *Ninth Intl. Conf. on Data Engineering (ICDE'93)*. Vienna, Austria: IEEE.

Flodin, S. and Risch, T. (1995). Processing object-oriented queries with Invertible late bound functions. *Twenty-First Conf. on Very Large Databases (VLDB'95)*. Zurich, Switzerland.

Flodin, S., Josifovski, V., Risch, T., Sköld, M., and Werner, M. *AMOS II User's Guide*, available at http://www.ida.liu.se/∼edslab.

Grufman, S., Samson, F., Embury, S.M., Gray, P.M.D., and Risch, T. (1997). Distributing semantic constraints between heterogeneous databases. *Thirteenth International Conf. on Data Engineering (ICDE'97)*. Birmingham, England: IEEE.

Josifovski, V. and Risch, T. (1998). Calculus-based transformations of queries over object-oriented views in a database mediator system, *3rd IFCIS International Conf. on Cooperative Information Systems*. New York City.

Kelley, W., Gala, S., Kim, W., Reyes, T., and Graham, B. (1995). Schema Architecture of the UNISQL/M Multidatabase System. In W. Kim (Ed.), *Modern Database Systems—The Object Model, Interoperability, and Beyond*. New York, NY: ACM Press.

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

Kuno, H., Ra, Y., and Rundensteiner, E. (1995). The Object-Slicing Technique: A Flexible Object Representation and its Evaluation. Univ. of Michigan Tech. Report CSE-TR-241-95.

Litwin, W. and Risch, T. (1992). Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates. *IEEE Transactions on Knowledge and Data Engineering*, 4(6), 517–528.

Lyngbaek, P., et al. (1991). OSQL: A Language for Object Databases. Tech. Report, HP Labs, HPL-DTD-91–94.

Object Management Group (1993). *The Common Object Request Broker: Architecture and Specification*, Object Request Broker Task Force.

Rundensteiner, E., Kuno, H., Ra, Y., Crestana-Taube, V., Jones, M., and Marron, P. (1996). The MultiView project: Object-oriented view technology and applications. *ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'96)* (pp. 555–563). ACM Press.

Santos, C. Design and Implementation of an Object-Oriented View Mechanism. GOODSTEP ESPRIT-III Technical Report, ESPRIT-III Project No. 6115.

Shipman, D. (1981). The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1). ACM Press.

Sköld, M. and Risch, T. (1996). Using partial differencing for efficient monitoring of deferred complex rule conditions. *Twelvth International Conf. on Data Engineering (ICDE'96)*. New Orleans, Louisiana: IEEE.

Straube, D. and Özsu, T. (1995). Query Optimization and Execution Plan Generation in Object-Oriented Database Systems. *IEEE Transactions on Knowledge and Data Engineering*, 7(2), 210–227.

Wiederhold, G. (1992). Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3).