Distributed Mediation using a Light-Weight OODBMS

Vanja Josifovski and Tore Risch
Linköping University, Sweden
{vanja, torri}@ida.liu.se

May 20, 1999

Abstract

An overview is given of a light-weight, Object-Oriented (OO), multi-database system, named AMOS II. Object-Oriented multi-database queries and views can be defined where external data sources of different kinds are translated through AMOS II and integrated through its OO mediation primitives. Through its multi-database facilities many distributed AMOS II systems can interoperate. Since most data reside in the data sources, and to achieve good performance, the system is designed as a main-memory DBMS having a storage manager, query optimizer, transactions, client-server interface, etc. The AMOS II data manager is optimized for main-memory and is extensible so that new data types and query operators can be added or implemented in some external programming language. Such extensibility is essential for data integration.

1 Introduction

The AMOS II system has its roots in the workstation version of the Iris system, WS-Iris [26], and the DAPLEX [34] functional data model. The core of AMOS II is an object-oriented, open, light-weight, and extensible database management system (DBMS). To achieve good performance AMOS II is designed as a main-memory DBMS. Each AMOS II server is also a DBMS of its own containing all the traditional database facilities, such as a storage manager, a recovery manager, a transaction manager, and an OO query language, named AMOSQL [12]. The system can be used as a single-user database or as a multi-user server to applications and to other AMOS II systems. The data manager is designed for main-memory and is optimized for efficient execution when the entire database fits in main-memory.

AMOS II is distributed mediator system [40] allowing a number of AMOS II mediator servers to communicate over the Internet. Applications can access data from several distributed data sources through a collection of mediators. The mediator servers have facilities for translating, combining, reconciling, and abstracting data through OO views over other mediators and external data sources. The abstraction services allow presenting different object view hierarchies in the different mediators. The mediator servers appear as virtual database servers having data abstractions and an OO query language. AMOS II mediators are *composable* since a mediator server can regard other mediator servers as data sources. A single AMOS II server can also assume more than one role and serve more than one application simultaneously. Different interconnecting topologies can be used to connect mediator servers depending on the integration requirements of the environment.

Some of the servers can be configured as *translators* [9] wrapping different kinds of data sources, e.g. access to relational databases through ODBC [4] or access to XML files [17]. We use the term translator (rather than the commonly used term *wrapper*) since translators are complete AMOS II systems which can wrap more than one data source and support semantic data abstractions and conversions from its data sources through OO views. A translator is thus also a mediator that provides a virtual OO database server layer that transparently translates data from some data sources. Wrappers are usually simpler interfaces to data sources while a translator can contain several wrapper subsystems for different data sources.

Mediator servers can also act as *integrators* which combine and convert data from other mediator servers through OO views. As for translators, these OO views provide a virtual OO database layer to be transparently accessed from clients and other mediator servers. Conflicts and overlaps between similar real-world entities being modeled differently in different data sources can be reconciled through the *mediation primitives* [18, 20, 22] of AMOSQL.

The declarative multi-database query language AMOSQL requires queries to be optimized before execution. The query compiler translates AMOSQL statements into object calculus and algebra expressions in an internal simple logic based language called ObjectLog [26], which is an OO dialect of Datalog [39]. As part of the translation into object algebra programs, many optimizations are applied on AMOSQL expressions relying on their OO and multi-database properties. During the optimization steps, the object calculus expressions are re-written into equivalent but more efficient expressions. For distributed multi-database queries the query decomposer distributes each object calculus query into local queries to be executed in the different distributed AMOS II servers and data sources. A cost-based optimizer on each site translates the local queries into procedural execution plans in an OO algebra, based on statistical estimates of the cost to execute each generated query execution plan expressed in the OO algebra. A query interpreter finally interprets the optimized algebra to produce the (partial) result of a query.

The query optimizer is *extensible* through a generalized foreign function mechanism, multi-directional foreign functions [26]. It gives transparent access from AMOSQL to special purpose data structures such as internal AMOS II meta-data representations or user defined storage structures. The mechanism allows the programmer to implement query language operators in an external language (Java, C or Lisp) and to associate costs and selectivity estimates with different user-define access paths. The architecture relies on extensible optimization of such foreign function calls [26]. They are important both for accessing external query processors [4] and for integrating customized data representations from data sources.

To achieve good performance we have carefully optimized the representation of critical system data structures, e.g. the storage manager, object representation, type information, and the representation of function definitions. We use tailored main memory data structure representations of system objects, rather than, e.g., storing them in relational tables represented as B-trees [13]. For example, our object identifiers are represented as variable length records with pointers to data structures representing type-information, function definitions, dependent objects, etc. It is crucial that system information is represented efficiently, since it is extensively looked up during both compilation and interpretation of AMOSQL functions. The storage manager has an incremental garbage collector for removing unused data.

AMOS II is runs under Windows NT. The system uses around 350KB of code and 1500KB of meta data. The system has client-server and inter-database communication primitives whereby AMOS II servers can communicate over TCP/IP.

The rest of this paper first describes the multi-database architecture of AMOS II, then its data model and query language is overviewed, and finally the data mediation primitives and their processing are summarized.

2 Multi-database Architecture

The multi-database architecture of AMOS II allow several AMOS II systems to connect and communicate over a network using TCP/IP. There are furthermore AMOSQL data interoperability primitives to exchange data between different AMOS II systems and to mediate semantically heterogeneous data (Sec. 5).

Fig. 1 illustrates how AMOS II systems can communicate and how they can be configured in different modes with respect to how they interact with other systems. The lines indicate communication between sub-systems where the arrows indicate the servers.

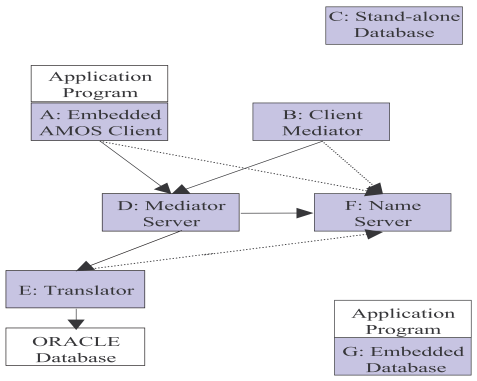

Figure 1: Distributed Mediator Architecture

The system can be configured in two dimensions:

- It can be a *single-user*, a *server* or an *embedded* system, where a single-user AMOS II system is a private database, a server is servicing several other AMOS II systems, and an embedded system is linked to some application.
- It can be a *stand-alone*, or a *mediator* system, where a stand-alone system is an isolated database and a mediator access data from some mediator(s) or data source(s).

The gray-shaded AMOS II systems in fig. 1 illustrate the following modes of operation along the two dimensions:

	Single-user	Server	Embedded
Stand-alone	C	F	G
Mediator	B	D,E	A

- (A) is an embedded AMOS II mediator linked to an application program. The small footprint of an embedded AMOS II system makes it easy to link it to applications. The system has interfaces to application programs in Java, C, and Lisp. Applications always access meditator servers by AMOSQL commands that are passed through an embedded AMOS II mediator.
- (B) is a single-user mediator importing and integrating data from AMOS II servers through the multi-database facilities, but not servicing other systems.
- (C) is a single user stand-alone database where the user can enter AMOSQL commands to populate, search, and update a private database.
- (D) is a mediator server servicing inter-database requests from other AMOS II systems and defining mediating OO views integrating data from other servers.

- (E) is a mediator server that translates data from a relational database. It has knowledge of how to translate AMOSQL queries to SQL and interfaces to call SQL through ODBC [4]. It can use the facilities of AMOSQL for semantic mediation of data from its data sources and its local database into views presented to other systems.
- (F) is a stand-alone database server accessed from mediator (D) by TCP/IP. It is also a *nameserver*. It keeps track of the mediator servers and clients in this group of mediators, as indicated by the dotted arrows. Every AMOS II mediator belongs to a group of mediators and must be given a unique name within the group. The nameserver is an ordinary AMOS II mediator server having the special task to store information about names, locations, and other meta-properties of the mediators in a group. A nameserver thus identifies a group of mediators and all mediators in the group will access meta-data about the federation of mediators from the nameserver (dotted lines in Fig. 1).
- (G) is a stand-alone embedded AMOS II system which provides database facilities for an application, e.g. for FEA analysis [30].

When you start running AMOS II you initially will have a stand-alone single-user database which cannot communicate with other AMOS II stand-alone databases or mediators. The stand-alone database can become a server by issuing some system function calls. There are furthermore system calls for making the stand-alone system join or leave a mediator federation through updates to the nameserver database.

3 Data Model

The data model of AMOS II is an OO extension of the DAPLEX [34] functional data model. It has three basic constructs: *objects*, *types* and *functions*.

Objects model all entities in the database. Everything in AMOS II is represented as objects managed by the system, both system and user-defined objects. There are two main kinds of representations of objects: *literals* and *surrogates*. The literal objects are self-described system maintained objects which do not have explicit OIDs, e.g. numbers and strings. Literal objects can also be *collections* of other objects, e.g. *vectors* (1-dimensional arrays of objects) and *bags* (unordered sets with duplicates). The surrogates have associated explicit object identifiers (OIDs) which are explicitly created and deleted by the user or the system. Examples of surrogates are objects representing real-world entities such as persons, meta-objects such as functions, or even AMOS II mediators as meta-mediator objects.

An object can be classified into one or more types making the object an *instance* of those types. The set of all instances of a type is called the *extent* of the type. The types are organized in a multiple inheritance, supertype/subtype hierarchy. 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 surrogate types can be *stored*, *derived*, *proxy*, or *integration union* types:

- **Stored types** have their extents explicitly stored locally in an AMOS II database. Their instances are maintained by the user.
- **Derived types** (DTs) are defined implicitly in terms of one or more *constituent* supertypes through a declarative query over the supertypes. Their extents are subsets of the *intersection* of the extents of the constituent types.
- **Proxy types** represent objects stored in other mediators or in some of the supported kinds of data sources.

- **Integration union types** (IUTs) are defined as supertypes of other types. An IUT extent contains one instance for each real-world entity represented by the (possibly overlapping) extents of the subtypes.

The proxy, derived and IUTs are the core of the integration framework in AMOS II. Composition of such types provide means for resolving a wide specter of semantic heterogeneities between the data and meta-data in the sources. Queries over the OO views are transformed into queries over data in multiple data sources.

Object attributes, queries, methods, and relationships are modeled by functions. Depending on their implementation the basic functions can be classified into *stored, derived, foreign,* and *proxy* functions; as well as *database procedures*:

- **Stored functions** represent properties of objects (attributes) stored in the database. Stored functions correspond to attributes in OO databases and tables in relational databases.

- **Derived functions** are functions defined in terms of queries over other AMOSQL functions. Derived functions cannot have side effects and the query optimizer is applied when they are defined. Derived functions correspond to side-effect free methods in OO models and views in relational databases. AMOSQL has an SQL-like select statement for defining derived functions which can also be used for ad hoc queries.

- **Foreign functions** are implemented through an external programming language (Java, Lisp or C). Foreign functions correspond to methods in OO databases and multi-directional foreign functions [26] provide access to external storage structures similar to data 'blades', 'cartridges', or 'extenders' in object-relational databases. To help the query processor, a multidirectional foreign function can have several associated access path implementations with cost and selectivity functions.

- **Proxy functions** represent functions in other mediators.

- **Database procedures** are functions defined using a procedural sublanguage of AMOSQL. They correspond to methods with side effects in OO models.

4 Query Processing

The AMOSQL query language is similar to OQL [6] but based on the functional query languages OSQL [28] and DAPLEX [34] with extensions of mediation primitives [20, 22], multi-directional foreign functions [26], late binding [11], active rules [35], etc. The functional data model is flexible and well suited for data integration, which actually was one of the motivations for the DAPLEX functional data model [34]. By describing type hierarchies and semantic heterogeneity using declarative functions and a functional Common Data Model there are many opportunities for the extensive query optimization needed in an OO mediation framework.

AMOSQL has data modeling as well as querying constructs. The general syntax for queries is:

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

For example, the following query retrieves the names of the parents of all persons having 'sailing' as hobby:

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

Fig. 2, presents an overview of the query processing in AMOS II. The first five steps, also called *query compilation* steps, translate the body of a query expressed in AMOSQL to a query execution plan which is stored with the query. To illustrate the query compilation we use the ad hoc query above.

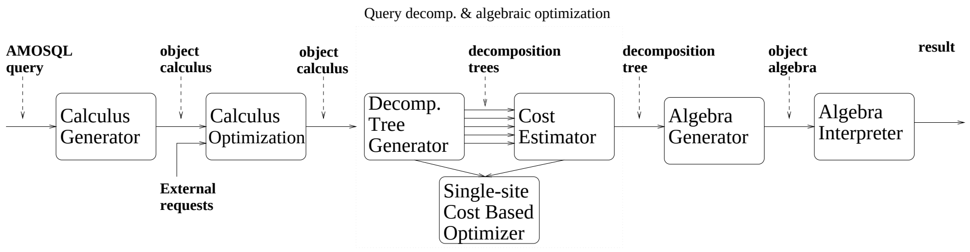

Figure 2: Query processing in AMOS II

From the parsed query tree, AMOS II first translates the AMOSQL queries into a type annotated *object calculus* representation [20]. For example, the query above is translated into the following calculus expression:

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

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 $Person$, $Person()$, which returns all the instances of its type. The variables nm and pa are generated by the system. Next, the calculus optimizer applies rewrite rules to reduce the number of predicates. In the example, it removes the type check predicate:

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

This transformation is correct because p is used in a stored function (e. g. $name$) with an argument or result of type $person$. The referential integrity system of stored functions constrains the stored instances to the correct type [26].

The query decomposition phase [18, 19] is invoked whenever a query is posed over data from more than one data source. It uses a combination of heuristic and dynamic programming strategies to produce an executable algebra plan from a query calculus expression operating over imported (proxy) and locally stored types. The query decomposition process is performed in 6 phases:

1. **Predicate grouping:** This phase attempts to reduce the problem of finding a suboptimal execution plan by reducing the number of predicates. Predicates executed at the same data source are grouped into one or more composite predicates that are treated afterwards as single predicates.

2. **Site assignment (predicate placement):** This phase uses cost-based heuristics to make the final decision which composite predicate is executed where,

eventually replicates some of the predicates, and assigns execution sites to those predicates that can be executed at more than one site (e.g. θ-joins specified by comparison operators). The output of this phase is a query graph where all the nodes are assigned to some site.

3. **Cost-based execution scheduling:** In order to translate the query graph from the previous phase into an executable query plan, the query processor must decide on the order of execution of the predicates in the graph nodes, and on the direction of data shipping between the nodes. Execution schedules for distributed queries in AMOS II are represented by *decomposition trees*. Each node in a decomposition tree describes one data cycle through a client mediator. In a cycle, first the intermediate results are shipped to the site where they are used. Then a subquery is executed at that site using the materialized data as input, and the result is shipped back to the client mediator. Finally, one or more post-processing subqueries are performed at the client mediator. The result of a cycle is always materialized in the client mediator. A sequence of cycles can represent an arbitrary execution plan. As the space of all execution plans is exponential to the number of participating databases, we examine only a subset of the family of left-deep decomposition trees using a dynamic programming approach.

4. **Tree distribution:** In this phase [19], a distributed compilation is performed at the participating mediators to rebalance and distribute the decomposition trees produced by the previous phase. One deficiency of the plans produced by the previous phase is that all the data flows are from or to one mediator. This can lead to many superfluous data flows when the data is to be transferred from one data source to another. The execution schedule resulting from the tree distribution can contain sequences where the data is shipped directly from one mediator to another, eliminating the bottleneck of shipping all data through a single mediator. This approach makes the set of mediators to function as one distributed mediation system. Some of the problems encountered here are how to transport OIDs and typed information in general through mediators that have no knowledge of these types.

5. **Object algebra generation:** The input to this phase is an executable decomposition tree, which is translated into equivalent sets of inter-calling local object algebra plans.

5 Data Integration by Object-Oriented Modeling

To provide data integration features, the type system is extended with *derived types* (DTs) [20] defined as subtypes of other types, and *integration union types* (IUTs) [22] defined as supertypes of other types. Data is integrated through DTs and IUTs by building an OO view type hierarchy based on local types, and types imported from other data sources, including other AMOS II servers. 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 specifications of the correspondence between the instances of the derived super/subtypes.

The DT instances are derived from the instances of their supertypes according to a declarative condition specified in the DT definitions. DT instances are assigned OIDs, which allows their use in locally stored functions (attributes) defined over the DTs in the same way as over the ordinary types. The DTs provide means for mediation based on operators such as join selection, and projection. However, these do not suffice for integration of sources having overlapping data. This is provided by the IUT framework, based on OO type hierarchies and late binding. IUTs are used to model unions of real-world entities represented by overlapping type extents. The integrated types become subtypes of the IUT. Equality among the instances of the integrated types is established based on a set of key attributes. IUTs can also have locally stored attributes, and attributes reconciled from the integrated types. Informally, while the DTs represent restrictions and intersections of extents of other types, the IUTs represent reconciled unions of data in one or more mediators or data sources.

Fig. 3 illustrates the use of OO views for data integration in AMOS II. A computer science department is (CSD) formed out of the faculty members of two universities named A and B. The CSD administration needs to set up a database of the faculty members of the new department in terms of the databases of the two universities. The faculty members of CSD can be employed by either one of the universities. There are also faculty members employed by both universities. The full time members of a department are assigned an office in the department.

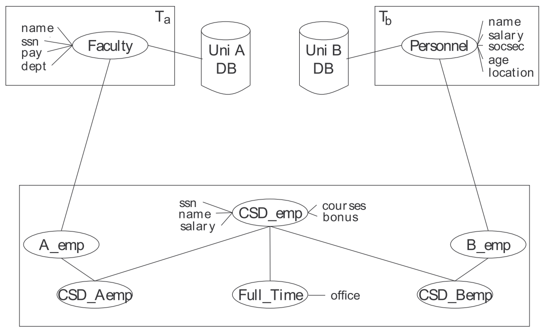

Figure 3: An Object-Oriented View for the Computer Science Department Example

In Fig. 3, the mediators are represented by rectangles; the ovals in the rectangles represent types; the solid lines represent inheritance relationships between the types; and the dashed lines represent inheritance relationship defined by IUTs. The two translators T_A and T_B provide a representation of the university databases in the of AMOS II data model. In translator T_A there is a type *Faculty* and in translator T_B a type *Personnel*. A mediator is setup in the CSD to provide the integrated view. Here, the types *CSD_A_emp* and *CSD_B_emp* are defined as subtypes of the types in the translators:

```
create derived type Faculty@Ta        create derived type Personnel@Tb
 subtype of A_emp                      subtype of B_emp
   where dept(A_emp) = ‘‘CSD’’;          where location(B_emp) = ‘‘G house’’;
```

The system imports the external types, looks up the functions defined over them in the originating mediators, and defines local proxy types and functions with the same signature but without any implementation.

The IUT *CSD_emp* represents all the employees of the CSD. It is defined over the *constituent types CSD_A_emp* and *CSD_B_emp*. *CSD_emp* contains one instance for each employee object regardless of whether it appears in one of the constituent types or in both. There are two kinds of functions defined over *CSD_emp*. The functions on the left of the type oval in Fig. 3 are derived from the functions defined in the constituent types. These *reconciled* functions have more than one overloaded implementation, one for each possible combination of constituent types instances, matching a IUT instance. The functions on the right are locally stored.

The data definition facilities of AMOSQL include constructs for defining IUTs as described above. The type *CSD_emp* is defined as follows:

```
CREATE INTEGRATION TYPE csd_emp
  KEYS ssn INTEGER;
  SUPERTYPE OF
    csd_A_emp ae: ssn = ssn(ae);
    csd_B_emp be: ssn = id_to_ssn(id(be));
  FUNCTIONS
    CASE ae
      name = name(ae);
      salary = pay(ae);
    CASE be
      name = name(be);
      salary = salary(be);
    CASE ae, be
      salary = pay(ae) + salary(be);
  PROPERTIES
      courses BAG OF STRING;
      bonus integer;
END;
```

For each of the constituent subtypes, a *key* expression is given. The instances of different constituent types having the same key values will map into a single IUT instance. The key expressions can contain both local and remote functions.

The `FUNCTIONS` clause defines the reconciled functions of *CSD_emp*, derived from functions over the constituent types. For different subsets of the constituent types, a reconciled function of a IUT can have different implementations specified by the `CASE` clauses. For example, the definition of *CSD_emp* specifies that the *salary* function is calculated as the salary of the faculty member at the university to which it belongs. In the case when s/he is employed by the both universities, the salary is the sum of the two salaries. When the same function is defined for more than one case, the most specific case applies. If no single most specific case exists (e.g. *name*), the system assumes "any" semantics and chooses one based on a heuristics to improve the performance of the queries over these functions. Finally, the `PROPERTIES` clause defines the two stored functions over the IUT *CSD_emp*.

The IUTs can be subtyped by DTs as other types. In Fig. 3, the type *Full_Time* is defined as a subtype of the *CSD_emp* type, representing the instances for which the number of courses exceeds certain number. The locally stored function *office* stores information about the offices of the full time CSD employees.

5.1 Implementation of OO Mediation Primitives

Queries over DTs are expanded by system-inserted predicates performing the DT system support tasks [20]. These tasks are divided into 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 derived templates and functions to perform these tasks. During query processing the query is analyzed and, where needed, the appropriate functions/templates are inserted. A selective OID generation mechanism avoids overhead by generating 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 functions specifying the view support tasks often have overlapping parts. [20] 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.

Each IUT is mapped by the system to a hierarchy of system generated DTs, called auxiliary types (ATs) [22]. The ATs represent disjoint parts (a join and two anti-semi-joins) of the outerjoin needed for this type of data integration. The reconciliation of the attributes of the integrated types is modeled by a system generated set of overloaded derived functions generated by the system from the specification in the IUT definition. Several novel query processing and optimization techniques are developed for efficiently processing the queries containing overloaded functions over the ATs, as described in [22].

6 Related Work

AMOS II is related to research in the areas of OO views, data integration, distributed databases and general query processing. There has been several projects on intergration of data in a multi-database environment [2, 5, 7, 8, 10, 14, 15, 23, 25, 27, 37, 38]. The integration facilities of AMOS II are based on work in the area of OO views [1, 3, 16, 24, 29, 32, 33, 36]. Due to the space constraints we do not detail the related approaches in this paper. [31] gives a good overview of distributed databases and query processing. An interested reader is referred to [21] for an elaborate comparison of AMOS II with other data integration projects.

Some unique features of AMOS II are:

- A distributed mediator architecture where query plans are distributed over several communicating mediator database servers. This direction has been noted as a promising future research direction in, e.g., [5].
- Modeling reconciled OO views spanning over multiple mediators and specified through declarative functional queries.
- Query processing and optimization of queries to reconciled views using OO concepts such as overloading, late binding, and type aware query rewrites.
- Query optimization strategies for OO mediator views that combine data stored in the mediator with the reconciled data.

7 Summary

We have given an overview of the architecture of the AMOS II mediator system where federations of distributed mediator servers can be composed by AMOS II servers. Each AMOS II system has DBMS facilities for query compilation, and exchange of data and meta-data with other AMOS II systems. OO views can be defined where data from several mediator servers are abstracted, transformed, and reconciled. AMOS II systems can furthermore be embedded in applications and used as stand-alone databases.

We described the OO and functional data model and query language forming the basis for data integration in AMOS II. We gave an overview of type-aware query re-writes of OO calculus query representations used by the query processor. The distributed multi-mediator query decomposition strategies used were summarized.

References

[1] S. Abiteboul and A. Bonner: Objects and Views. *ACM SIGMOD'91 Conf.*, ACM Press, 1991.

[2] R.Bayardo, et al: Infosleuth: Agent-based semantic integration of information in open and dynamic environments, *ACM SIGMOD'97 Conf*, 1997.

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

[4] Silvio Brandani: Multi-database Access from AMOS II using ODBC. *Linköping Electronic Press*, 3(19), Dec., 1998, *http://www.ep.liu.se/ea/cis/1998/019/*.

[5] O. Bukhres, A. Elmagarmid (eds.): *Object-Oriented Multidatabase Systems*, Pretince Hall, 1996.

[6] R.Cattell: *The Object Database Standard: ODMG-93 2.0*, Morgan Kaufman, 1996

[7] U. Dayal, H. Hwang: View Definition and Generalization for Database Integration in a Multidatabase System, *IEEE Trans. on Softw. Eng.* 10(6), Nov. 1984.

[8] C. Evrendilek, A. Dogac, S. Nural, F. Ozcan: Query Optimization in Multidatabase Systems. *Distributed and Parallel Databases*, 5(1), Jan. 1997.

[9] G. Fahl, T. Risch: Query Processing over Object Views of Relational Data. *The VLDB Journal*, 6(4), November 1997.

[10] D. Fang, S. Ghandeharizadeh, D. McLeod and A. Si: The Design, Implementation, and Evaluation of an Object-Based Sharing Mechanism for Federated Database System. *9th Data Engineering Conf. (ICDE'93)*, 1993.

[11] S. Flodin, T. Risch: Processing Object-Oriented Queries with Invertible Late Bound Functions, *21st VLDB Conf.*, Zurich, Switzerland, 1995

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

[13] H.Garcia-Molina and K.Salem: Main Memory Database Systems: An Overview, *IEEE TKDE Journal*, 4(6), Dec. 1992.

[14] H.Garcia-Molina, et al: The TSIMMIS Approach to Mediation: Data Models and Languages. *Intelligent Information Systems (JIIS)* 8(2), Kluwer, 1997

[15] L. Haas, D. Kossmann, E. Wimmers, J. Yang: Optimizing Queries accross Diverse Data Sources. *23th VLDB Conf.*, Athens Greece, 1997

[16] S. Heiler and S. Zdonik: Object views: Extending the Vision. *6th Data Engineering Conf. (ICDE'90)*, 1990

[17] H.Lin: *Querying XML Data from an Object-Oriented and Extensible Database Mediator System*, Technical Report, EDSLAB/IDA, Linköping University, 1999.

[18] V. Josifovski: *Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration*, Ph.D. Thesis, Linköping U., Sweden, 1999

[19] V.Josifovski, T.Katchaounov, T.Risch: *Optimizing Queries in Distributed and Composable Mediators*, Technical Report, (submitted for publication) EDSLAB/IDA, Linköping University, 1999.

[20] V.Josifovski and T.Risch: Functional Query Optimization over Object-Oriented Views for Data Integration, *Intelligent Information Systems (JIIS)* Vol. 12, No. 2/3, Kluwer, 1999.

[21] V.Josifovski and T.Risch: *Comparison of Amos II with Other Data Integration Projects* Technical Report, EDSLAB, Linköping University, 1999, *http://www.ida.liu.se/~edslab/amosII_comp.pdf*.

[22] V.Josifovski, T.Risch: Integrating Heterogeneous Overlapping Databases through Object-Oriented Transformations, *25th VLDB Conf.*, Edinburgh, Scotland, Sept. 1999.

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

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

[25] E-P. Lim, et al: Myriad: Design and Implementation of a Federated Database System. *Software - Practice and Experience*, 25(5), May 1995.

[26] W. Litwin and T. Risch: Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates. *IEEE TKDE Journal* 4(6), 1992

[27] L.Liu, C.Pu: An Adaptive Object-Oriented Approach to Integration and Access of Heterogeneous Information Sources, *Distributed and Parallel Databases*, 5(2), April 1997.

[28] P. Lyngbaek et al: *OSQL: A Language for Object Databases*, Tech. Report, HP Labs, HPL-DTD-91-4, 1991.

[29] A. Motro: Superviews: Virtual Integration of Multiple Databases. *IEEE Transaction on Software Engineering*, Vol. SE-13, No. 7, July 1987.

[30] K.Orsborn, T.Risch: Next Generation of O-O Database Techniques in Finite Element Analysis. *Intl. Conf. on Computational Structures Technology*, Budapest, Hungary, 1996.

[31] M.T.Özsu, P.Valduriez: *Distributed Database Systems*, Prentice Hall, 1999.

[32] E. Rundensteiner, H. Kuno, Y. Ra, V. Crestana-Taube, M. Jones and P. Marron The MultiView project: object-oriented view technology and applications, *ACM SIGMOD'96 Conf.*, 1996.

[33] M. Scholl, C. Laasch and M. Tresch: Updatable Views in Object-Oriented Databases. *Second Deductive and Object-Oriented Databases Conference (DOOD91)*, Dec, 1991.

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

[35] M. Sköld, T. Risch: Using Partial Differencing for Efficient Monitoring of Deferred Complex Rule Conditions. *12th Data Engineering Conf. (ICDE'96)*, 1996.

[36] C. Souza dos Santos, S. Abiteboul and C. Delobel: Virtual Schemas and Bases. *Intl. Conf. on Extending Database Technology (EDBT'92)*, Vienna, Austria, 1992.

[37] S. Subramananian and S. Venkataraman: Cost-Based Optimization of Decision Support Queries using Transient Views. *ACM SIGMOD'98 Conf.*, 1998

[38] A. Tomasic, L. Raschid, P. Valduriez: Scaling Access to Heterogeneous Data Sources with DISCO. *IEEE TKDE Journel*, 10(5), 1998

[39] J.D.Ullman: *Principles of Database and Knowledge-Base Systems*, Volume I & II, Computer Science Press, 1988, 1989.

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