A Mechanism and Experimental System for Function-Based Sharing in Federated Databases*

Doug Fang, Joachim Hammer, and Dennis McLeod
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781
USA

Abstract

A function-based approach and mechanism to support sharing among the component database systems in a federation is described. In the context of a functional object-based database model, a technique to support inter-component information unit and behavior sharing is presented. An experimental system that implements the function-based sharing mechanism is described, its underlying algorithms are outlined, and its practical utility and effectiveness are assessed. This work is couched in the framework of the Remote-Exchange research project and experimental system.

Keyword Codes: H.2.5.
Keywords: Heterogeneous Databases

1 Introduction

A key challenge to supporting the interoperation of database systems is to provide facilities for the sharing and exchange of information units and units of behavior across database system boundaries. To provide a perspective on inter-component information sharing and exchange in a federated database environment, we propose a function-based viewpoint. In the context of the current Remote-Exchange research project, we employ a functional object-based database model, and develop a comprehensive mechanism for the transparent sharing of instance objects, type objects, and behavioral objects [Fang and McLeod, 1992].

In this paper, we provide an overview of our underlying perspective on function-based sharing in federated database systems. We specifically present an overview of the possible sharing patterns, using examples from our experimental prototype system based on the

*This research was supported in part by NSF grant IRI-9021028.

Omega [Ghandeharizadeh, 1991] and Iris [Fishman *et al.*, 1987] object-based database management systems. We address the essential problems that are associated with each sharing situation and describe our mechanism to support sharing as implemented in our experimental prototype system.

1.1 Related Research

At the top level, there are two distinct aspects of function-based sharing: (1) the location (local vs. remote) of the execution of functions, and (2) the location of the actual information units upon which functions operate. Research in the area of distributed programming languages has of course addressed issues of the remote execution of functions (which may also be termed operations, methods, or behavioral objects) [Liskov, 1988; Strom and Yemini, 1985]. The primary concern of this work is with the programming of the functions themselves, e.g., language constructs, communication primitives for sending and receiving data, etc. The location of data used by these functions is directly coded into the methods themselves.

Work in the area of database systems, on the other hand, predominantly focuses on the manipulation of information units; this is related to the second main aspect of function-based sharing. Research on object-oriented database systems has approached the problem of supporting behavior in the database itself [Atkinson, 1989; Fishman *et al.*, 1987; Kim *et al.*, 1990; Lecluse *et al.*, 1988; Maier *et al.*, 1986]. When these systems are extended to a distributed environment, it is the location of the (persistent) data that determines the location of the remote execution of the functions [Fishman *et al.*, 1987; Kim *et al.*, 1990; Maier *et al.*, 1986]. In the Remote-Exchange approach, functions are implemented without explicit knowledge of where they will be executed or where the data will reside.

1.2 The Functional Object-Based Context

The conceptual database model considered in this research draws upon the essentials of functional database models, such as those proposed in Daplex [Shipman, 1981], Iris [Fishman *et al.*, 1987], and Omega [Ghandeharizadeh, 1991]. Our functionally object-based model contains features common to most semantic [Afsarmanesh and McLeod, 1989; Hull and King, 1987] and object-oriented database models [Atkinson, 1989], such as GemStone [Maier *et al.*, 1986], *O*2 [Lecluse *et al.*, 1988], and Orion [Kim *et al.*, 1987]. In particular, the model supports complex objects (aggregation), type membership (classification), subtype to supertype relationships (generalization), inheritance of functions (attributes) from supertype to subtypes, run-time binding of functions (method override), and user-definable functions (methods).

In our function-based model, functions are used to represent inter-object relationships (attributes), queries (derived data), and operations (methods). Three types of functions can thus be distinguished:

- *Stored Functions*: A stored function records data as primitive facts in the database. Stored functions can be updated.

- *Derived functions*: A derived function is defined by a data manipulation language (DML) expression. The value of a derived function cannot always be updated directly.

- *Computed functions*: A computed function (sometimes termed a foreign function) is defined by a procedure written in some programming language. The value of a computed function cannot be directly updated.

For the purposes of this paper, derived and computed functions are treated uniformly, and will be termed "computed functions" herein.

2 Function-Based Sharing

Let us assume the existence of a function $\mathcal{F}$, which can be shared among components of a federation; without loss of generality, assume $\mathcal{F}$ takes as input the argument a[1]. The argument type can be a literal (i.e., **Integer**, **String**, ...) or a user-defined type such as **Research-Papers**, for example. Sharing takes place on a component-pairwise basis, meaning that $\mathcal{F}$ is exported by a component $C1$ and imported by a component $C2$. The importing component is called the *local database*, while the exporting component is called the *remote database*. There are several ways components $C1$ and $C2$ can share the service provided by $\mathcal{F}$, depending upon the location where $\mathcal{F}$ executes and upon where its input argument a resides (i.e., there are two degrees of freedom). At this level of abstraction there are four distinct *function-argument* combinations:

- local function - local argument

- local function - remote argument

- remote function - local argument

- remote function - remote argument

Upon closer analysis we can note that it is also necessary to differentiate between stored functions and computed functions. At this (finer) level of granularity, we can now distinguish between a total of eight different sharing scenarios[2], as presented in Figure 1. In this table of Figure 1, "Local" refers to the domain of the local database while "Remote" refers to the domain of the remote database. Local objects are those that belong to the local database, while remote objects belong to the remote database.

It is important to note that a principal goal of our approach is to provide a mechanism for function-based sharing that makes the location of a function and its argument transparent to the user. The details of how this transparency can be achieved are more fully described in Section 3, but we will briefly highlight the process here. In particular, we note that the state of a remote object (i.e., its functional values) always resides in

[1]Since the argument can be a complex unit of information, this is not a limitation; multiple arguments can be handled by an obvious extension of our approach.

[2]We now have three degrees of freedom.

Figure 1: Eight different situations for function-based sharing

the remote database, but when the object is imported to a local database, a surrogate is created for it in the local component. The creation of such surrogates is necessary in order to refer to remote objects using local database system tools without modification [Fang *et al.*, 1991]. Since these surrogates are created locally, the local system is able to interpret and manipulate remote objects as usual, for example, when using them as arguments in local function calls. However, when retrieving the actual state of a remote object, the use of surrogates alone is not sufficient. Our approach exploits the extensible nature of our object-based database model by rewriting the functions encapsulating surrogate objects as computed functions. These higher-level computed functions serve as place holders, and retrieve the results of applying the function on the remote component where the object is actually stored.

Given the above observations, it is now possible to consider each of the sharing situations summarized in Figure 1. We first focus on stored functions, and then turn our attention to computed functions.

2.1 Sharing Stored Functions

As a framework for analysis, consider the a scenario of two collaborating researchers: Researcher-A and Researcher-B maintain separate databases of journal and conference publications. Figure 2 specifies the meta-data (conceptual schemas) for two example components[3]. Let Researcher-A denote the local component and Researcher-B denote the remote component.

[3] In order to keep the signature of each function as short as possible, we have omitted the input argument whenever it is obvious; the result type, on the other hand, is always explicitly shown.

Figure 2: Two component databases

The four situations for the sharing of stored functions among components can be analyzed as follows:

1. *Local function - Local object*
   This is what we term the *base case*. Both objects, the stored function $\mathcal{F}$ and its argument a, reside in the local component and can be executed as usual without any additional mechanisms.

2. *Local function - Remote object*
   In this case, $\mathcal{F}$ is a local stored function that is applied to remote argument a. For example, Researcher-A's $Author()$ function can be applied to the **IEEE-Papers** that have been imported from Researcher-B. As previously mentioned, surrogates are created in Researcher-A's database for each remote object. These surrogates are collected in a newly created subtype of **Publications** called **IEEE-Papers**. All local functions, for example the functions defined on **Publications** in Figure 2, operate normally on the surrogates (i.e., the instances of **IEEE-Papers**). In the case of local functions that do not have counterparts in the remote component (e.g., $Author()$), new function values can be created locally for each imported instance. As a result, each imported object also has local state in Researcher A's database and can be altered by Researcher-A without affecting the original object in Researcher-B's database. The three functions $Title()$, $Pub\_Date()$, and $Text\_Body()$ that are shown next to Researcher-A's **IEEE-Papers** in Figure 2 are actually remote functions defined in Researcher-B's schema; these have been included in Researcher-A's database for completeness. Exact details of how these functions are created and invoked are presented in Section 3. Provided that Researcher-A has created new values for $Author()$ for each of the imported **IEEE-Papers**, s/he might pose the

following OSQL query in this context:

```
select Author(PAPERS)
for each IEEE-Papers PAPERS
```

3. *Remote function - Local object*
   This situation is somewhat meaningless, since stored functions only have a meaning in the local context of the component in which they were initially created. For example, suppose that Researcher-A, who does not have the *Pub_Date*() function defined in his/her schema, wants to see the publication date of all his/her **Conference-Papers**. Researcher-A would first need to create a new function and then populate it with the appropriate values, rather than being able to use Researcher-B's *Pub_Date*() function. However, when looking at this situation from Researcher-B's point of view, one can argue that this case is merely the mirror image of the second case (Local function - Remote object). Rather than executing Researcher-B's function remotely in Researcher-A's database, one can integrate Researcher-A's **Conference-Papers** into Researcher-B's schema (create a new subtype, **Conference-Papers**, of **Research-Papers** and populate it with Researcher-A's **Conference-Papers**) and execute the *Pub_Date*() function locally.

4. *Remote function - Remote object*
   This case serves as the basis for *instance level sharing*. All remote objects of interest to a component, say Researcher-A, must already be imported into the local database using surrogates. Each time a remote object is referenced, the surrogate "points" to the remote object and the desired functional value is fetched using a remote procedure call to the other component (see Section 3). In this situation (see Figure 2), Researcher-A might pose the following OSQL query against his/her schema:

   ```
   select Pub_Date(PAPERS)
   for each IEEE-Papers PAPERS
   ```

   In order to find the *Pub_Date*() of all remote **IEEE-Papers**, surrogates for each of these papers are used to locate the remote papers. For each remote paper, the publication date is determined using the remote *Pub_Date*() function and the result of this function is returned to the local database.

2.2 Sharing Computed Functions

In order to consider sharing for computed functions, consider again the example of two collaborating researchers. Figure 3 shows the two example component databases as before, but with additional (computed) functions: **DviView()**, a dvi format previewer, **PostView()**, a postscript previewer. The four situations for the sharing of computed functions among components can be analyzed as follows:

Figure 3: Two component databases with extended functionality

1. *Local function - Local object*
   As in the case of stored functions, this is the base case. Computed function $\mathcal{F}$ as well as its argument a resides in the local component and the execution is local (e.g., **latex(Text_Body(**a**))**).

2. *Local function - Remote object*
   This situation can be reduced to the base case described in case 1. For example, if Researcher-A wants to view one of Researcher-B's papers, s/he will run *latex* on the *Text_Body*() of the surrogate for that paper, say *inst*, and apply $DviView()$ to the result (e.g., **DviView(latex(Text_Body(***inst***))))**.

3. *Remote function - Local object*
   This is the reverse of the previous case: the function executes remotely and the input argument is supplied from the local database. For example, Researcher-A may desire to view postscript text but does not have a postscript previewer in his/her own local database. In this case, s/he will invoke Researcher-B's $PostView()$ function remotely through a previously created handle in his/her own database and supply it with a local argument. In effect, the remote database is providing a non-local "service".

4. *Remote function - Remote object*
   This situation is similar to the first case (Local function - Local object) in that both the state of the object and execution of the function are in the same component. For example, Researcher-A views one of Researcher-B's **IEEE-Papers** using Researcher-B's original $PostView()$ function. Compared to the first case (Local function - Local object) where no sharing takes place and execution occurs locally,

in this case all the processing is done on the remote site. In order to invoke a remote computed function using remote arguments from within the local component, surrogates for the remote objects must be created locally. These surrogates enable the local component to access the actual state of desired objects which reside with the remote component. This procedure is similar to instance level sharing (see Section 3.2) where surrogates for shared instances are created locally in order to provide access to the actual state of each instance in the remote component.

In the examples above, the functions being shared have returned a literal type (e.g., the *Author*() function returns a **String**). However, functions with signatures involving abstract (user-defined) types can also be shared. In this case, both the input and output argument types must be defined locally; if they are not, their meta-data must be imported beforehand. The location of the result argument is determined by the location where the function executes.

2.3 Observations on the Practical Use of Function-Based Sharing

With the above analysis and framework in place, it is now possible to make some observations on the practical utility of the function-based sharing capabilities supported by our mechanism. In the above analysis of function-based sharing, we stressed the separation of the location where the function executes from the location where the data resides. However, from a user's perspective, this separation of function execution and argument location is completely transparent.

Our analysis of eight different sharing patterns can be reduced to two "most interesting" cases: (1) executing an imported function on a local argument, and (2) executing a local function on an imported (shared) argument. The first case involves the reuse of a previously defined function in a different environment; this may be termed "behavior sharing" or *function level sharing*. The second case can be described as extending the "characteristics" of a remote object while at the same time respecting the autonomy of the originating site. That is, the importer can customize the remote object according to his local conceptual schema. These local attributes are managed entirely by the local component avoiding any unnecessary modification to the originating (remote) component.

In both cases above, the user is not aware of the environment in which a shared function executes, and need not worry about where the state of an imported object actually resides. Instead, components are able to freely browse the meta-data and functions that have been made available (i.e., exported) by others in the federation in order to select the services that they would like to share[4] (i.e., import).

[4]Our discussion has in a sense assumed that there are no access restrictions in place that would further complicate the sharing process. An investigation of access control and authorization in object-based databases is the subject of a related research project at USC.

3 Experimental Prototype Implementation

An experimental implementation of our function-based sharing mechanism has been designed and built using our Remote-Exchange testbed consisting of a federation of Omega and Iris database components [Fang and McLeod, 1992]. In what follows, we describe the essential aspects of this testbed, and examine critical implementation issues we faced in our experiments.

3.1 Sharing in the Remote Exchange Testbed

In the current Remote-Exchange testbed, we have implemented the seamless (transparent) importation of objects from remote databases. In our functional object-based model, objects can be instances, types, or functions. The various function-based sharing patterns described above can be examined in the context of instance, type, and function level sharing.

Conceptually, in instance level sharing, a remote instance object is imported directly into a local type. This remote instance behaves in the same manner as a local instance object from the user's perspective. However, the actual state of the remote instance exists in the remote component database; retrieval of any state of the remote object is done by accessing the remote database transparently. Hence, access to remote instance objects corresponds to the Remote function - Remote object situations described above[5].

A special case of the Remote function - Remote object situation arises when the remote object belongs to a type that is not present in the local schema. Up until now we have assumed that all remote instance objects belong to a type that is also present in the local component. However, we can envision a scenario in which a component wishes to import the services of a function operating on a type that does not exist in the local schema. For example, Researcher-B is interested in authors of Researcher-A's conference papers (see Figure 1). In this case, the remote type must first be imported to the local schema in order for our sharing mechanism to work. It is important to note that "creating" a new local type corresponding to a remote type requires some additional work in the sense that there may be problems with the value types for new functions, as well as finding the proper place for it in the existing type hierarchy. This special case involves *type level sharing*, which is investigated in more detail in [Fang and McLeod, 1992].

The importation of a remote function object corresponds to the sharing of behavior (function level sharing). Intuitively, when an instance object is imported, only data is being shared. On the other hand, importing a function gives the importer access to services not provided by his/her local system. This corresponds to the Remote function - Local object situations described above.

The principal remaining useful situation is the important case of Local function - Remote object. Among other things it allows users to add additional state to remote objects without modification of the exporting database, thereby preserving the autonomy of the exporter. This ability to create local state for remote objects is achieved automat-

[5]Note that this does not depend upon whether the function is stored or computed.

Figure 4: Sharing instance objects

ically from the way we implement instance level sharing and is analogous to simple local database access.

3.2 Instance Level Sharing Implementation

Our mechanism for importing instance level objects follows three steps:

1. Create local surrogates for remote objects.

2. Create computed functions for retrieving data from remote components.

3. Overwrite functions defined on surrogates to use (or refer to) the newly created computed functions in step 2.

The local surrogate serves as a local handle for accessing a remote instance. By using the surrogate, differences between remote representations of objects, e.g., object identifiers (OIDs) can be masked out (made transparent). Since the state of the remote object exists externally, computed functions for accessing that state must be created. These computed functions use the Remote Procedure Call (RPC) paradigm [Birrell and Nelson, 1984] for accessing the remote component. Finally, in order to have the surrogates use the remote functions, any existing functions defined on the surrogate must be overridden to use the RPC defined functions. This is implemented by dynamically binding functions to objects.

In terms of the collaborating researchers, we can envision a scenario where Researcher-B wants to import instances of another remote type **IEEE-Papers** into his/her local schema using the above approach. This is depicted in Figure 4. Surrogates are created as instances of both a local type (i.e., Researcher-B's **IEEE-Papers**) and a remote type called **R-IEEE-Papers** in Figure 4. The purpose of the new surrogate type is to override the local functions that surrogates inherit from the local type to which they belong. By additionally creating the surrogate as a member of this surrogate type, the functions that the surrogate instance originally inherited are overridden. The two thin dotted arrows from **R-IEEE-Papers** ("R" for remote) and **IEEE-Papers** to the surrogate instance serve to indicate that surrogate instances (i.e., remote instances) of **IEEE-Papers** are created as members of both **IEEE-Papers** and **R-IEEE-Papers**. Thus these (remote) instances inherit functions multiply from both **R-IEEE-Papers** and **IEEE-Papers**; any duplicately named function from the two types is overridden by the function defined for **R-IEEE-Papers**. The functions defined on **R-IEEE-Papers** are computed functions, which make RPC requests to the remote component database and retrieve the values of functions on remote instances.

3.3 Function Level Sharing Implementation

As in instance level (and type level) sharing, meta-information containing the location (e.g., remote OID and remote component name) of the remote function object (unit of behavior) being imported must be stored locally. However, by contrast with the case for instance level sharing, this meta-information is associated directly with the function being imported. In instance level sharing, the meta-information is indirectly kept for remote functions such as *Title()* and *View()* via the surrogate instance object (see Figure 4). Thus, in our implementation we distinguish between two kinds of "remote" functions: those implicitly defined through instance level importation and those directly imported through function level importation. Figure 5 shows our mechanism for incorporating meta-information for function level importation. We exploit the fact that meta-information is also represented using our functional object-based model. Imported functions are created as instances of the type **Remote-Functions** and can thus store and access the additional location meta-information required to execute the imported function.

Figure 5 depicts a slightly different sharing pattern from Figure 4. In this scenario, the *View()* function is imported from some remote component. In addition, **R-IEEE-Papers** no longer supplies a *View()* function. Hence both the remote and local instances of **IEEE-Papers** use the same imported *View()* function for displaying research papers. This is evident in Figure 5 by the absence of the *View()* function from the type **R-IEEE-Papers** and the addition of a new (italicized) *View()* function defined on **Research-Papers**[6].

In order to explain how the *View()* function works, we must first explain how our implementation addresses the issue of side-effects. By side-effect we really mean two things: (1) any kind of implicit input other than the input argument that is necessary to compute the result, and (2) any modifications to the state of the database where the function executes other than to the input argument. For functions whose arguments are literals, this simply requires that the function being imported computes its result value

[6]The italicized font is used to indicate that the *View()* function is imported and no longer local.

Figure 5: Sharing function objects

solely based on its input argument without modifying any database state. Functions whose input argument is non-literal pose additional difficulties. In this case, the input argument is the OID of an instance. The problem then lies in determining what information a computed function accesses in order to compute its results. Strictly applying our definition of side-effects would restrict computed functions on non-literals to solely accessing and then manipulating the input OID. But realistically, a computed function must be able to access some state of the instance corresponding to the OID when computing its result. In our implementation, we take the position that the only state that a computed function can access are those functions that serve to encapsulate that object. In other words, the only state the computed function will possibly access are those functions that are defined on the types of which the instance is a member. In the case of the *View()* computed function, these functions are *Title()*, *Text_Body()*, and *Pub_Date()*.

Having determined what information a computed function on a non-literal type can access, a problem arises when trying to execute such a function remotely on a local object. The problem occurs when supplying local arguments to a remote computed function. Although we know the computed function is limited to only accessing the functions that encapsulate the instance, we do not know exactly which ones it does need. Even if we pass all the possible values the computed function can access, the computed function must be written in such a way as to retrieve these arguments from the remote component and not the local one. This would be undesirable and contrary to our goal of relieving the computed function writer of needing to know where the data on which it operates is located.

The best approach to this problem in our autonomous environment is to allow computed function writers to define functions without concern as to whether the function is to be exported. Thus, in our implementation, whenever a remotely executing computed function needs state from the local component, it performs a callback to the local component to retrieve that state. For the instance level sharing described in Section 3.2, the local component simply runs as a client and makes RPC requests to the exporter running as a server. However, for function level sharing, when the callback mechanism is used, the local component must in addition run as a server to accept the callback requests. Computed functions can be written using any programming language that can be compiled to re-entrant object code. This object code is then dynamically linked into the database management system kernel when the computed function is accessed. In our prototype using the Omega database management system, a computed function accesses the local component through *Omega_eval()*, which has two parameters: an argument and the function that is to be applied to the argument.

We can now consider how the callback mechanism works transparently and how it allows a computed function to be written uniformly without regard as to whether that function is to be exported. Consider again the example using the *View()* computed function. Suppose that the imported *View()* function retrieves the *Text_Body()* of an instance (say in latex format), computes the dvi formatted version, and displays the formatted version through a dvi previewer (say xdvi). When the user of the local component invokes the *View()* function on a local object, the *View()* function is passed the local OID of a **Research-Papers** instance. The *View()* on the remote server makes a call to *Omega_eval()* to retrieve the *Text_Body()*, *Omega_eval()* recognizes that the OID passed in as its argument is not local and performs a callback to the server of the local component that invoked the *View()* function. Since the local server recognizes the OID as a local OID, it performs the request and passes back the *Text_Body()* to the remote server which can than complete its computation and display the results on the local database monitor.

4 Conclusions and Future Directions

In this paper, we have presented a function-based approach and mechanism for sharing in a federated database system. An experimental implementation of the function-based sharing mechanism using the Remote-Exchange experimental testbed was examined. In our approach, we have considered the importance of decoupling the location of (persistent) data and the location of the execution of methods that operate on it. Traditional approaches inextricably link the location of the data and the execution of the operation.

An area of research that has not been directly considered in this paper involves local component updates of objects at a remote component. Other research efforts specifically address the issues in this area [Breibart *et al.*, 1990]. However, our approach allows the local component to create local (e.g., stored) functions on remote objects. This has the overall benefit of allowing the local component to create local state for remote objects that completely conform to the local system's mechanism for updates.

The focus of our experimental prototype implementation has been mainly on instance level sharing and function level sharing. We have found these sharing patterns to be very natural and easy to use in our environment. We are currently further investigating abstract (complex) object sharing, and are proceeding to more substantially quantify the performance efficiency of our mechanisms.

References

[Afsarmanesh and McLeod, 1989] H. Afsarmanesh and D. McLeod. The 3DIS: An Extensible, Object-Oriented Information Management Environment. *ACM Transactions on Office Information Systems*, 7:339–377, October 1989.

[Atkinson, 1989] M. Atkinson, et al. The Object-Oriented Database System Manifesto. In *Proceedings of the 1st Intl. Conf. on Deductive and Object-Oriented Databases*. Kyoto, Japan, December 1989.

[Birrell and Nelson, 1984] A. Birrell and B. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Breibart *et al.*, 1990] Y. Breibart, A. Silberschatz, and G. Thompson. Reliable Transaction Management in a Multidatabase System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 215–224. ACM SIGMOD, May 1990.

[Fang and McLeod, 1992] D. Fang and D. McLeod. A Testbed and Mechanism for Object-Based Sharing in Federated Database Systems. Technical Report USC-CS-92-507, Computer Science Department, University of Southern California, Los Angeles CA 90089-0781, February 1992.

[Fang *et al.*, 1991] D. Fang, J. Hammer, D. McLeod, and A. Si. Remote-Exchange: An Approach to Controlled Sharing among Autonomous, Heterogenous Database Systems. In *Proceedings of the IEEE Spring Compcon*. IEEE, San Francisco, February 1991.

[Fishman *et al.*, 1987] D. Fishman, D. Beech, H. Cate, E. Chow, T. Connors, T. Davis, N. Derrett, C. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. Neimat, T. Ryan, and M. Shan. Iris: An Object-Oriented Database Management System. *ACM Transactions on Office Information Systems*, 5(1):48–69, January 1987.

[Ghandeharizadeh, 1991] S. Ghandeharizadeh, et al. Design and Implementation of OMEGA Object-based System. Technical Report USC-CS, Computer Science Department, University of Southern California, Los Angeles CA 90089-0781, September 1991.

[Hull and King, 1987] R. Hull and R. King. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[Kim *et al.*, 1987] W. Kim, J. Banerjee, H. T. Chou, J. F. Garza, and D. Woelk. Composite Object Support in an Object-Oriented Database System. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 118–125, 1987.

[Kim *et al.*, 1990] W. Kim, J. Garza, N. Ballou, and D. Woelk. Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge Data Engineering*, 2(1):109–117, March 1990.

[Lecluse *et al.*, 1988] C. Lecluse, P. Richard, and F. Velez. O_2, an Object-Oriented Data Model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Chicago, Ill., June 1988. ACM SIGMOD.

[Liskov, 1988] B. Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3):300–312, 1988.

[Maier *et al.*, 1986] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an Object-Oriented DBMS. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 472–482. ACM, 1986.

[Shipman, 1981] D. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 2(3):140–173, March 1981.

[Strom and Yemini, 1985] R. Strom and S. Yemini. The NIL Distributed Systems Programming Language: A Status Report. *ACM Sigplan Notices*, 20(5):36–43, May 1985.