A Federated Architecture for Information Management

DENNIS HEIMBIGNER
University of Colorado, Boulder
DENNIS McLEOD
University of Southern California

An approach to the coordinated sharing and interchange of computerized information is described emphasizing partial, controlled sharing among autonomous databases. Office information systems provide a particularly appropriate context for this type of information sharing and exchange. A federated database architecture is described in which a collection of independent database systems are united into a loosely coupled federation in order to share and exchange information. A federation consists of components (of which there may be any number) and a single federal dictionary. The components represent individual users, applications, workstations, or other components in an office information system. The federal dictionary is a specialized component that maintains the topology of the federation and oversees the entry of new components. Each component in the federation controls its interactions with other components by means of an export schema and an import schema. The export schema specifies the information that a component will share with other components, while the import schema specifies the nonlocal information that a component wishes to manipulate. The federated architecture provides mechanisms for sharing data, for sharing transactions (via message types) for combining information from several components, and for coordinating activities among autonomous components (via negotiation). A prototype implementation of the federated database mechanism is currently operational on an experimental basis.

Categories and Subject Descriptors: H.2.1 [**Database Management**]: Logical Design—*data models*; *schema and subschema*; H.2.4 [**Database Management**]: Systems—*distributed systems*; H.4.1 [**Information Systems Applications**]: Office Automation

General Terms: Algorithms, Design, Languages, Management

Additional Keywords and Phrases: Office information systems, distributed information management, federated databases

1. INTRODUCTION

The office information environment presents many new information management challenges [8, 28, 35]. In particular, the types of information and the patterns of information access are quite different from those of application

This research was supported in part by the Joint Services Electronics Program through the Air Force Office of Scientific Research under contract F49620-85-C-0071, in part by the National Science Foundation under grant MCS-8203485, and in part by the Defense Advanced Research Projects Agency under contract MDA903-81-C-0335.

Authors' addresses: D. Heimbigner, Computer Science Department, University of Colorado, Boulder, CO 80309. D. McLeod, Computer Science Department, Los Angeles, CA 90089-0782; ARPANET: McLeod@USC-ISIB.

environments for which conventional database management technology and systems are intended. The integrated database system was primarily developed to support large, integrated, centralized databases. In a decentralized information environment there may not be a single, integrated database containing all of the organizational information under the control of a centralized data processing organization. Rather, databases tend to proliferate throughout an organization with little or no control by any one group. In such an environment there is a need for substantial information management flexibility, partial integration/sharing, and autonomy [23].

A possible response to the decentralization of information is to attempt to return to centralization by reintegrating the data into a "composite database," sometimes called a "heterogeneous database." To form a composite database, a new, global/virtual conceptual schema is introduced, which describes the information in the databases being composed; database access and manipulation operations are then mediated through this new conceptual schema. This return to strict integration does, however, require centralization.

Inherent in the concept of integration is the existence of some authority (namely, a database administrator) responsible for designing and maintaining the conceptual and physical (implementation) schemas of the database. Thus, the process of integrating existing databases forces control over their structure to be ceded to some central authority. The users of the existing databases may have expended considerable resources (hardware, software, and human) in developing their databases and may be reluctant to lose control of them. Furthermore, changes to the structure of a database must pass through the database administrator, and they must be weighed against competing demands for change.

The integration of existing databases into a composite database is in general quite difficult. An important problem in this regard is that the same fact may be contained in several databases yet be represented using different conceptual structures. For example, one database may represent a memo by links/mappings among the writer, recipient(s), and memo content, while another database may represent the memo as a distinct object with links to the writer, recipients, and memo content. If a composite database approach is used, one or the other of these conceptual representations must be selected, and applications using the other representation may need modification. In sum, the composite structure may be difficult to construct, and if it can be constructed, it may be suboptimal for many users' needs.

In light of the difficulties posed by composite databases, it is appropriate to pursue an alternative architecture that allows the existing database systems to maintain their autonomy, yet provides a substantial degree of information sharing. The goal of this paper is to define an architecture and supporting mechanisms for interconnecting databases that minimizes central authority, yet supports partial sharing and coordination among database systems. This *federated database architecture* [10, 12, 13] allows a collection of database systems (*components*) to unite into a loosely coupled federation in order to share and exchange information. The term *federation* refers to the collection of constituent databases participating in a federated database.

Without the constraint of a central authority, the mechanisms provided for the federated architecture must balance two conflicting requirements: the components must maintain as much autonomy as possible; however, the components must be able to achieve a reasonable degree of information sharing. The first of these requirements, autonomy, specifically refers to four capabilities:

(1) A component must not be forced to perform an activity for another component. The role of centralized authority must be replaced by cooperative activity among components and supporting protocols.

(2) Each component determines the data that it wishes to share with other components. Since partial, controlled sharing is a fundamental goal of the federated approach, each component must be able to specify the information to be made available as well as to specify which other components may access it and in what ways.

(3) Each component determines how it will view and combine existing data. In a composite system, all access to the underlying data is mediated by a global schema. In a federation, each component must be able to, in effect, build its own "global" schema that is best suited to its needs.

(4) A component must have "freedom of association" with respect to the federation. Since the federation is a dynamic entity, components must be able to dynamically enter or leave the federation. Further, a component must be able to modify its shared data interface, adding new data and withdrawing access to previously shared data.

As a counterpoint to the capabilities required to support component autonomy, the federated architecture must provide mechanisms to support information sharing. Specifically, components can communicate in the following three ways:

—*Data communication.* Each component has a collection of data, and other components may be interested in accessing some portion of those data. Exchanging the data is the primary activity in a federation, and so a mechanism to support data sharing is essential to the operation of a federation.

—*Transaction sharing.* A component may not wish to share its data directly, but rather to share operations upon its data. This may be the case if the data are sensitive or have consistency constraints attached to them. In any case, components must be able to define transactions that can be invoked by other components.

—*Cooperative activities.* In a system of autonomous components, the only way that the federation can function correctly is by cooperation. Components must be able to initiate a potentially complex series of actions involving cooperation with other components. Cooperation in this context refers to negotiated data sharing.

Using these facilities, a collection of components in an office information system can collectively achieve a substantial amount of sharing while maintaining essential control over their data.

2. DISTRIBUTED DATABASE SYSTEMS

Databases, viewed as structured collections of information, can be roughly classified along two dimensions representing: (1) conceptual/logical structure (semantics), and (2) physical organization and structure. Each dimension may in turn be divided into two parts: centralized and decentralized. Using this framework, four classes of databases can be identified: (1) logically centralized and physically centralized databases, which include the conventional integrated databases; (2) logically centralized and physically decentralized databases, including *distributed databases*,[1] as well as a number of recent approaches to composite database support; logically decentralized and either (3) physically centralized or (4) physically decentralized databases, which represent the province of *federated databases*.

Integrated databases, distributed databases, and most approaches to composite databases represent the logically centralized approach: They provide a single conceptual schema for users and application programs. Multiple external schemas (views) may be provided in such systems, but a single central conceptual schema is nonetheless required. In this approach, an integration of the data associated with an application environment is attempted.

The distributed database architecture, for example, as described in [19], [29], [30], and [34], fits into the category of logically centralized and physically decentralized databases. In such a database the users and applications access data described through a single conceptual schema, but the data may be physically stored in many separate computers, typically the nodes of a computer network. This architecture is specifically designed to provide a unified system that is distributed across several interconnected computer systems. As such, it does not address the integration of preexisting databases except by discarding those databases and pacing their data into the new system. Thus the distributed database architecture does not address the same issues as the federated architecture.

Additional architectures have been proposed for databases, and these are more directly comparable to federated databases. The common feature of these *composite database systems* is that they attempt to combine a number of existing databases into a single, logically centralized entity. This is achieved by defining a global schema and then defining translation functions between the global schema and the local schemas of the constituent databases. In order to construct a composite database, two problems must be solved. First, the global schema must be defined in such a way as to integrate all the information in the local schemas and remove as much redundancy as possible. As indicated earlier, such integration is potentially very difficult. Second the translations must be defined so that operations on the global schema may be translated into equivalent operations on some set of the local schemas.

Composite database systems may be further classified as *homogeneous composite databases* and *heterogeneous composite databases*. The former is a composite database in which all of the local schemas as well as the global schema are defined using the same database model. Although this limits the complexity of

[1] The term "distributed databases" is used here as it has been mainly used in the literature, denoting a logically centralized, physically distributed system.

the translation functions, they are not trivial since two or more local schemas may structure the same information in different ways. Thus, no matter how the global schema structures the information, some transformation will be necessary to access one or the other of the local schemas.

A composite database is said to be *heterogeneous* if one or more of the constitutent databases does not use the same database model as is used for the global schema. A heterogeneous composite database system inherits the difficulties associated with homogeneous composite database systems, and in addition must face the additional problem of translating between different data models. This problem is aggravated by the fact that constructs in one model may not exist in another model.

There are several existing and proposed composite database systems, including R* [20], XNDM [15, 16], Multibase [33], "superviews" [25], and the work of Litwin [21, 22]. A significant limitation of composite architectures is their basic centralized nature, and particularly the existence of a global schema. However, these approaches provide significant concepts and principles to address the problems of heterogeneity, integration, and, to some extent, autonomy (R*). Many of the ideas of these approaches have been incorporated into the federated architecture.

By contrast with composite systems, the federated database uses an organizational model based on equal, autonomous databases, with sharing controlled by explicit interfaces. The effect of modifications may be limited, and no database has authority over another. There is no global schema in a federation. Rather, each component has direct access to the original data provided by other components, and it is free to restructure that information into whatever form is most appropriate to its needs. The control of the sharing rests with the owner of the data, but the negotiation mechanism ensures that changes to the structure of the data proceeds in an orderly fashion.

The federated approach has some commonalities with the approach to "information object sharing" described in [23]; in fact, many of the basic concepts in [23] are derived from initial work on the federated architecture [12, 13]. The focus of [23] is to provide a small set of operations for object definition, manipulation, and retrieval in a distributed environment, modeled as a logical network of office workstations. Relationships among objects can be established across workstation boundaries, objects are relocatable within the distributed environment, and mechanisms are provided for access control. An object-naming convention supports location transparent object references, which means that objects can be referenced by user-defined names rather than by address. By contrast, the federated architecture is focused on a higher level than the object-sharing approach, in that it provides more explicit intercomponent interfaces, specific capabilities to support negotiation, and a "semantic" model of information vis-a-vis sharing.

3. THE FEDERATED DATABASE MODEL

Before discussing the details of the federated architecture, it is necessary to summarize the database model used to describe data in a federation. The architecture presented in this paper assumes that a common database model is used throughout the federation; that is, the federation is homogeneous. It is, of

course, possible to have a heterogeneous federation if the components do not all use the same database model. "Hooks" exist in the architecture to deal with heterogeneity, although this paper does not consider them in detail; principles and techniques devised in research on composite database systems can be employed in this regard [15, 16, 21, 22, 25, 33].

The federated database model used in this paper is based on an object-oriented database model: the *event model* [17]. The event model is a characteristic "semantic database model" [1–3, 8, 11, 18, 26, 32]. Specifically, the federated database model is based on three basic data modeling primitives:

—*Objects.* The basic modeling element is an object, which corresponds to some (real world) entity or concept (e.g., the person Jane Smith or the number 5). Objects are divided into two categories: *descriptor objects* and *abstract objects.* Descriptor objects are atomic strings of characters, integers, or Booleans, and generally serve as symbolic identifiers in the database. Decriptor objects are the only directly displayable objects; thus all external references to objects in the database must ultimately be in terms of descriptor objects. All nondescriptor objects are abstract objects. They are not directly displayable, except in terms of related descriptor objects (such as unique identifiers).

—*Types.* Types are time-varying collections of objects that share common properties; the objects of a given type are called the *instances* of that type. Some types are designated descriptor types in that they may only contain descriptor objects. All other types are designated abstract types. A type may be a *subtype* of another (parent) type if it is defined so that its set of instances is always a subset of the instances of the parent type. Associated with any subtype is a *predicate* that determines which objects that are instances of the parent type are also instances of the subtype. A particular subclass of abstract types required in the federated architecture, termed *message types*, is described in Section 4.7.

—*Maps.* Maps are "functions" that map objects from some domain type to sets of objects in the power set of some range type. A number of simple integrity constraints may be specified with each map; for example, a map may be specified to be *single-valued* (i.e., its value for all objects in the domain type has cardinality of zero or one) or *multivalued*, and a map can be declared to be a *unique identifier* (*key*).

In addition to the data structuring primitives described above, the federated information model provides primitive operators for data retrieval and modification. In the federated model, data manipulation primarily involves traversing the directed graph of types and maps that constitutes a given database. The actions of the data manipulation operators are defined principally in terms of *cursors* [9], which in this model are abstract ordered sets of objects together with a pointer into that set. Each cursor refers to a unique sequence of objects and also contains some state information about that sequence. In particular, the cursor contains a marker that points to some specific element of the associated sequence. The model contains operators to create and destroy cursors, and to sequence through the elements of the cursor.

The database model also assumes the existence of *transactions*, which are procedures expressed in some programming language. The parameters to these

procedures are objects in the database system. A transaction is integrated into the schema by representing its interface as a type and its parameters as maps associated with that type. Each invocation of a transaction creates an instance of the type associated with that transaction.

In a federated database, objects always reside in the component (database) in which they are created, but references to them may be passed to other components so that the objects can be manipulated remotely through a set of exported operators. Using this facility, a copy of any object can be created by any component, but the copy is a different object. This scheme requires that objects be given names that are unique with respect to an entire federation. Such unique names must themselves contain some tag indicating the component that contains the specified object. These unique names are generated by concatenating the component name with a local object name. The component name is guaranteed to be unique within the federation, and this is enforced by the federation system. A unique local object name (unique with respect to a component) can be generated by using a simple counter that is incremented whenever a new object is created. Alternatively, a clock of sufficient resolution can be used to generate unique names. Somewhat more general naming schemes can also be used, for example, as described in [23] and [28].

4. THE FEDERATED DATABASE ARCHITECTURE

The basic elements of the federated architecture are components, of which there may be any number; components represent individual information systems that wish to share and exchange information. Each federation has a single *federal dictionary*, which is a distinguished component whose information province is the federation itself. The federal dictionary supports the establishment, maintenance, and termination of a federation. The only difference between the federal dictionary and any other component is the database it contains. It has no direct control over other components, and it does not mediate communications among other components.

A component may be viewed as an autonomous database. A component has associated with it three schemas, each of which describes some class of information important to the proper functioning of the component. Each of the three schemas of a component is a collection of types and maps. The three component schemas are the private schema, export schema, and import schema; these are described immediately below.

4.1 Private Schema

The *private schema* describes that portion of a component's data that is local to (stored at) the component. The bulk of the private schema is devoted to describing the application data available in the database of a component. This portion of the schema, as well as the data it describes, corresponds to a normal database in a nonfederated environment. Although some of this information will remain local to the component, a portion of the application data and transactions will be exported to other components.

In addition to the application-specific data, the private schema contains a small collection of information and transactions relevant to the component's participation in the federation. This information is exported by the component

for use by other components, particularly the federal dictionary. The federation-specific information falls into three categories:

(1) descriptive information about the component, such as the component name and network address;
(2) primitive operations for data manipulation, such as accessing a type and traversing a map;
(3) the import and export schemas.

4.2 Export Schema

The export schema portion of the component specifies the information that the component is willing to share with other components of the federation. The export schema consists of a collection of types and maps denoting the information to be exported to other components. As in the private schema, the exported information is divided among federation-specific information and application-specific information. The federation-specific information is much the same as the federation-specific information of the private schema and is explicitly derived from it. The application-specific information is analogously derived from the information in the application-specific portion of the private schema.

The export schema is actually a metaschema consisting of a set of types and maps in the component schema that contain the definitions of the types and maps that are to be exported. Other components import this export schema and peruse it like any other information. Not all exported types and maps are represented in the export schema. Certain primitive types and maps are always assumed to be exported, and it is not necessary for them to be explicitly included in the export schema.

Each type and map in an export schema must have certain properties associated with it. There are five properties for types: *category, definition, derivation, access list,* and *connection list.* For exported maps, there are six properties: the same five as for types plus a list of *constraints.* The category property specifies the kind of type or map: descriptor or abstract. The definition property indicates whether the type or map is derived; its actual derivation expression is specified by the derivation property. The constraint list for maps specifies whether the map is single valued, a unique identifier (key), etc. The access list and connection list are used to control access to exported types. The access list property specifies which other components may access this type. The connection list specifies which other components have imported this type and hence are potentially accessing it.

In a given federation each component will have certain types and maps that it is willing to share with every other component, but it will also have other elements that it is willing to share only with some specified subset of the components in the federation. In the federated architecture this is supported by placing access controls on types and maps in the export schema. Thus the first line of control over data access is the export/import mechanism, and the access controls provide a finer grain of control on top of that mechanism.

Access controls must be specified in terms of components rather than individual users of a component, because enforcement of user-level access controls is dependent on the proper operation of the component. If a component has errors

that allow one user of the component to masquerade as another user, then a user may circumvent the access protections. It is possible for one component to enforce component-level access controls since it mediates every access to its data by another component, assuming proper operation of the underlying network.

An access list is a set of ordered pairs. The first element of each pair is a component identifier; the second element of the pair is an access right assigned to that component, specifying some type or map. Any component not contained in the access list has no right to access the type or map. There are two kinds of access rights: "read" and "write" (which implicitly allows read also). If a component has "read" access to a type, then it is allowed to "open" the type to sequence through the objects in that type. A component may only have "read" access to a map if it also has "read" access to the domain and range types of the map. If a component has "read" access to a map, then it can traverse ("apply") the map from a domain object to a set of range objects. A component with "write" access to a map allows a component to change the mappings for a given object, as well as perform "read" operations.

4.3 Import Schema

The import schema of a component specifies the information that component desires to use from other components. As for the other two schemas, the import schema deals both with federation-specific and application-specific information. Both the application-specific information and the federation-specific information are specified by a schema derived from the corresponding (accessible) portions of the export schemas of other components.

An imported type or map has the same properties as an exported type, except that it has no access list and no connection list. In addition, each imported element (type or map) has a derived definition property, specifying how the imported element is derived from the underlying exported element(s).

4.4 Schema Importation

Schema importation is the fundamental information-sharing operation in a federation. The term "importation" refers to the process of modifying a component's import schema as well as gaining access to some element of exported information. Before a component enters a federation, it imports nothing. As soon as it enters, it imports sufficient built-in information to function within the federation. This level of importation is essentially automatic. Beyond this, all importation of information is at the discretion of the component itself and must be explicitly negotiated with other components.

In order to import information, each component must know or discover what information is available in the federation. This is accomplished in two steps. First, through the federal dictionary each component may discover the names and network addresses of the other components. Second, each component contacts those components, using a predefined protocol. At this point a component is in a position to peruse the export schemas of the other components and engage in the schema importation process.

To illustrate the importation process, suppose that component c1 exports a type t1. Further suppose that component c2 wishes to import t1 for reading as

its (c2's) type t2. C1 arranges the importation through the following negotiation:

(1) C2 requests c1 to give it read access to t1.
(2) C1 grants c2 the access, modifying the connection list.
(3) When c2 receives the affirmative reponse from c1, it adds a new type t2 to its import schema. The type t2 is defined to be derived with an initial derivation expression of "c1 > t1"; that is, it is a derivation of type t1 of component c1.

It is important to note that the importation of a type or map is separate from subsequent data access. The type is imported once, and then all subsequent accesses to the contents of that type are carried out directly, without the overhead of negotiation. The actual data transfers occur when the importer attempts to scan the contents of the type. This is completely analogous to access to a local type except that the data is transferred over a network.

When component c2 imports a schema element that is exported by another component c1, an implicit contract is established between c1 and c2. In this contract c1 guarantees that it will not modify the definition (structure or semantics) of the exported element unless it notifies the importer, c2. By this contract c2 also agrees to notify c1 when it no longer requires access to the element. This process of negotiated change is a key element of the federated architecture.

Three kinds of modification caused by evolving information sharing patterns require notification: giving the importer no access to the element, changing the importer's access right from "write" to "read," and changing the semantics of the element. In the first case, where the importer is denied any access to the element, the importer is obligated to relinquish the connection to the element. Of course, the importer cannot be forced to do this, but the implicit contract has been broken and further access would be denied. In the other two cases the importer has the option of either relinquishing access to the element or notifying the exporter that the modified element is an acceptable replacement for the original element, and so continue to use the element.

4.5 Type and Map Derivation Operators

Once some set of types and maps have been imported, a component can proceed to restructure that information to suit its purposes. To this end, the architecture provides a set of *derivation operators* for manipulating type and map definitions to produce new ones.

Before discussing the various type and map derivation operators, it is necessary to describe the concept of an *object equality function.* Such functions are essential for combining information across component boundaries. In the federated database model it is assumed that two objects from different components a priori refer to different objects. Often this is acceptable, but sometimes it is necessary to indicate that two objects owned by different components in fact do represent the same entity. Object equality functions are used to define this equivalence of objects.

An object equality function is a string (descriptor object) manipulation operation defined in a programming language.[2] The argument to the function is a string that is assumed to be an assigned name for an object. This string may be derived from the object via a series of key map traversals until a desriptor value is reached. The equality function takes that string and computes another string as a value. This string is assumed to represent some other object, which can be found via an additional series of traversals of key maps.

As an example, suppose that component c1 contains a type "employee" with a key map "employee-number" that is an encodng of the employee's social security number of the form "123456789." Also, suppose that component c2 contains a type "manager" with a map "social-security-number" of the form "123-45-6789." A possible equality function, denote it by ~, is a string funtion that takes a number of the form 123456789 and transforms it to 123-45-6789. Thus, given an employee, one constructively finds the equivalent manager by (1) obtaining the employee-number of the employee, (2) converting it to a social-security-number using ~, and then (3) finding the manager with that social-security-number.

The *type derivation operators* are used to construct new types as combination of existing types, which in turn may be derived types. These operators treat types as multisets of objects. There are four principal type derivation operators:

—*Concatenate* combines the instances of two types to create a new type. As a typical example, a unified type for airplanes might be constructed by concatenating the types for various makes of airplanes.[3]

—*Subtraction* subtracts the instances of one type from the instances of another type. Subtraction is most often used to obtain the complement of a type. For example, given types "airplanes" and "military-airplanes," one may obtain "commercial-airplanes" via subtraction.

—*Cross product* creates a type with one instance for every n-tuple of objects from some set of n types. This operation might be used for example to create a type "date" as the cross product of types "month," "day," and "year."

—*Subtype* allows a new type to be created via some predicate on another type. For example, given the type "airplanes" with a map "kind" specifying whether the airplane is commercial or military, the subtype "commercial-airplanes" is a subtype of airplanes where the map "kind" has the value "commercial."

As for types, it is possible to derive new maps from existing maps. Object equality functions are considered derived maps, although the derivation is by means of an arbitrary host-language procedure. Some type derivation operations (concatenate, cross product, and subtype) automatically induce new maps on the derived type.

In addition to object equality functions, there are the following eight *map derivation operators.*

---

[2] In the case of the prototype federated system, described below, this language is LISP.

[3] Since these operations generally involve types of two different components, it is typically necessary to specify an object equality function for defining common objects in the two types.

—*Composition* defines a new map as the composition of two other maps (e.g., *A* and *B*). If the map *A* and/or the map *B* are multivalued, then the composed map consists of all objects in the range of map *B* that are obtained by following an *A* map from the domain of *A* to the range of *A*, and then following a *B* map; if the result of applying either map is undefined, then so is the composition. For example, composing the map "manufacturer" of type "airplanes" with the map "name" of "manufacturer," the result is a new map specifying the name of the manufacturer of an airplane; if "manufacturer" and "name" are multivalued, then the composition consists of the set of all names of all manufacturers of airplanes.

—*Inversion* defines a new map as the inverse of another map. For example, inverting the map specifying the manufacturer of an airplane gives a map specifying the airplanes of a manufacturer.

—*Extension* extends the definition of a map from a type to its supertype. Even if the original map is total, its extension will be partial since it is undefined for objects in the supertype but not in the type. Extension is most commonly used with composition to allow a map on a subtype to be attached to the supertype. Thus if the type "commercial" airplanes had a map specifying the number of cabin attendants, composing this map with the extension map to type "airplanes" could provide the number of cabin attendants for all airplanes. Note, however, that the value of the map would be undefined for military airplanes.

—*Restriction* restricts the definition of a map from a type to its subtype. If the original function is total, then so is the restricted function. This derivation allows a map on a type to be attached to a subtype.

—*Cross product* creates a map whose value is the two-tuple of values produced by applying two other maps. It is most commonly used with the cross-product type derivation. Thus if the type "airplane" has maps specifying the year, month, and day of manufacture, these maps could be combined via cross product to create a map representing the date of manufacture.

—*Discrimination* maps are automatically defined for each type derived by concatenation. If *n* types are concatenated, then *n* discimination maps are defined. The *i*th discrimination map is defined only on elements from the *i*th type, so it may be used with selection to test whether an object originated from a particular type. This derivation is usually used with the selection derivation.

—*Projection* maps, similarly to discrimination maps, are automatically defined for cross-product types. The *i*th projection map selects the *i*th element of any *n*-tuple of the cross product. This derivation can be used, for example, to choose the "month," "day," and "year" maps of the "date" type created via cross product.

—*Selection* allows a map to be one of a set of map expressions based upon a series of condition tests (viz., a "case statement"). The conditions are also map expressions. Each condition is evaluated, and if it results in a defined value, then the corresponding map expression is evaluated and returned as the result of the selection. When used in conjunction with discrimination maps, selection can be used to convert an operation on a concatenated type into an operation on one of types from which it was created.

This collection of type and map derivation operators is quite low-level (compared for example to [25] and [27]). In particular, it is important to realize that the derivation of a type is generally independent of the derivation of maps. For example, suppose two types, X and Y, are concatenated into a new type Z. Further, suppose that X and Y both have associated maps called "name." This does not mean that Z automatically is given a derived map called "name." Instead, that derivation, if it is desired, must be explicitly constructed. A higher level interface supporting such functionality, which, for example, automatically derives maps when new types are derived, can be constructed using the primitives provided here.

4.6 Data Update

In addition to providing read access to imported and derived data, the federated architecture must provide the capability for components to update such data. The update problem is complicated by the existence of derived types whose update requires updating the types from which it is derived (called "base types"). The problem of updating derived data is essentially the same as the view update problem [4, 7, 14], which has been principally studied in the context of the relational database model. Briefly, the problem is that the derived type is obtained by a mapping from a set of base types to the derived type. To update a derived type, it is necessary to invert the derivation function so that updates to the base types can be determined from the update to the derived type. In the most general case this inversion is impossible (it may be undefined or ambiguous), which means the derived type cannot be updated correctly.

The following approach to the update problem is adopted in the federated architecture. If the derivation is direct (i.e., renaming only), then update is allowed. Otherwise the data abstraction approach of Rigel [31] is used. In this method, all updates to derived types are funneled through an associated set of user-defined operations. Thus, the definer of the derived type also specifies all possible operations on that derived type, and specifically the update operations for the type. In practice, this method just transfers the problem to the definer of the derived type, who must choose operations and their parameters so that enough information is available to do the inversion. In the federation these operations are specified by means of a set of message types implementing transactions that perform the meaningful updates. The exporter is free to define the semantics of these transactions as desired.

4.7 Message Types

The decentralized nature of a federation dictates the need for many forms of communication among components. The capability for importing base types and maps is one form of communication, but other means are needed to support the exchange of higher levels of information. Specifically, the federation must allow components to import and invoke transactions defined by other components. Shared transactions are useful for two purposes. First, they can be used to control updates to shared data, much as in abstract data types. Second, they are needed to implement the negotiation subsystem (described in the next section).

The federated architecture allows components to share transactions through message passing. This facility is embodied in the *message type* construct, which

serves as an interface specification for some transaction. The message type, like any type, can be exported by one component and imported by other components; it is this process that constitutes the sharing of a transaction among components.

The message type describes a class of intercomponent messages. Associated with the exporting component is a procedure that defines the transaction associated with that message type. The normal message-sending paradigm is transformed to the database paradigm of creating a new object: creating an object of the message type is equivalent to sending the object as a message. At the receiving component, the message is queued as an instance of the type. The message objects are scanned in arrival order, and the earliest one is passed to the transaction for processing. When the transaction is finished, the object is returned to the sender to signal the completion of one message passing cycle. If the object sender attempts to access the object before it is returned, the sender is delayed. Thus, to the sender the process is reasonably transparent and appears more or less as a normal object creation and access activity.

The maps associated with the message type allow parameters to be passed to the receiver and results returned to the sender. The maps associated with the type are partitioned into two kinds: *input maps* and *output maps*. The input maps define attributes of the messages that are intended to be inputs to the transaction associated with the message type. Similarly, the output maps represent results returned after the transaction processes the message.

A message actually consists of two objects. The importer of the message type creates a surrogate object locally and assigns values to all of the input maps. Each message type has two predefined input maps, "msg-surrogate" and "msg-source," whose values are assigned by the database system. The "msg-surrogate" map specifies a unique object name of the surrogate, while the "msg-source" map provides the name of the component sending the message.

For example, if component c1 exports the message type "order" with maps "part" and "quantity" as follows:[4]

order:
  part → partname
  quantity → integer

then c2 may import the type and maps as

c1/order:
  part → partname
  quantity → integer

The identifier "c1/order" is c2's local name for the imported version of c1's "order" type. An order message can be sent (by c2) using the following sequence:

1. let m = new (c1/order)
2. insert-map (m, c1/order.part, "wrench")
3. insert-map (m, c1/order.quantity, 100)

Here step 1 creates a new object of type "order," and steps 2 and 3 establish the input maps.

[4] Here the arrow separates a map name from its range type.

It is at the point when all the input maps have been assigned that the object is actually sent to the receiver. The originating component, c2 in this case, collects the name of the surrogate object, its own name (c2), and the input map values ("wrench" and 100), and converts them to a linear message suitable for shipping over the network. It should be noted that when the map value is an object, the value shipped is the unique name of that object (i.e., objects are transmitted by reference).

When the message is received by c1 (the exporter), c1 creates an instance of its message type ("order" in this case) and assigns the input map values taken from the message. The transaction of the message type is then invoked with this object as its argument. The transaction performs whatever action it desires, assigns values to the output maps, and returns. In the case where map values are objects (object references), accesses to these remote objects are converted to appropriate intercomponent messages. After the transaction is complete, the output maps are collected, linearized, and returned to the sender. The sender assigns the output values to its surrogate object and continues operation.

4.8 Negotiation

With the absence of a central authority, the federated architecture provides a mechanism to coordinate the sharing of information among components: the negotiation subsystem. A negotiation is a multistep, distributed dialogue among two components. For example, there is a built-in negotiation that sequences through the steps for importing a type that was exported by some component. Other negotiations control the entry and exit of components with respect to the federation. It is important to note that negotiation is distinct from the process of data access. Negotiation establishes the right to access some general kinds of data elements. Once this is established, the primitives of Section 3 are used to manipulate that data.

It is also possible for users to define new application-specific negotiations as well. As a corollary, the structure of negotiations must be accessible to the user, and hence they must be at least partially embedded in the database itself. The actual negotiation subsystem has two main parts: an interpreter and a negotiation language for writing negotiation procedures. The negotiation procedures (written in the negotiation language) are stored in the database of each component. Each negotiation procedure contains three elements: a set of *participant schemas*, a *negotiation schema*, and a *negotiation graph*.

Any particular negotiation is conducted between two participants, which are abstractions for components in the federation. Several of the same kind of negotiation may be in operation simultaneously but with different bindings of participant to component. Each participant has a participant schema, which when instantiated provides local memory during the negotiation. The schema may be parameterized, and the participant's state is initialized with actual values of these parameters when the negotiation is invoked.

To support the negotiation process, each participating component has a *negotiation database*. A negotiation database is described by a negotiation schema, which specifies a collection of types and maps; the database is in turn a collection of objects and map instances matching the format of the negotiation schema. It

contains the actual state for any single instance of a negotiation. One of the types in the negotiation schema is called the *token type*; it represents the "root" type of the negotiation schema in that it serves as a handle to reach all other portions of the negotiation schema. The instance of the token type in the negotiation database is designated the *token*. During the steps of a negotiation, the token is exchanged between the participants until some final result is determined. The negotiation database is modified during the steps of the negotiation to reflect changes in the state of the subject of the negotiation.

The possible steps for each kind of negotiation are specified by the negotiation graph. If a negotiation is viewed as a program, then the negotiation graph serves as a representation of the major control flow of that program. A negotiation graph consists of a collection of nodes connected by arcs. Each negotiation node stands for a possible state of the negotiation, and each negotiation arc indicates a possible transition between states. A negotiation node has the following structure:

—the *node name* describes the state associated with that node.
—The *class* of the node indicates whether a node is *initial*, which means that it is the (unique) starting node for the negotiation, *terminal*, which means that it is a final node for the negotiation, or *other*. The three classes are mutually exclusive properties of the nodes.
—The *transitions* are the arcs from a given node leading to other nodes.
—Associated with each node is a procedure that defines the *semantics* of the node. It determines which transition is taken from the node on the basis of any criteria it chooses, for example, by interrogating one of the participants to the negotiation, or by some arbitrary computation.
—Each node of the graph is assigned to an *owning participant*, which "owns" that node.

A negotiation arc has the following structure:

—The arc has an *arc name* that is used as an input to the node semantics.
—The *source* and *destination* are the nodes connected by the arc.

One of the components in a federation initiates a negotiation by assigning itself as participant one and choosing another component as participant two. The use of "one" and "two" is arbitrary, but one of the participants must be identifiable as the initiator of the negotiation. Participant two is notified that a particular negotiation is to be initiated by means of an imported message type, specific to that negotiation. Each participant creates its local state and initializes it. In addition, participant two is responsible for creating the token and returning its unique name to participant one.

The negotiation starts with the participant that owns the initial node of the negotiation graph. It "places" the token upon the initial node of the negotiation graph via an arc with no name. Whenever the token is "located" at a given node,

the owner of the node executes the procedure associated with that node. This procedure is given four arguments: its state, the token, the name of the current node, and the name of the last arc traversed. The procedure may examine and modify its local database and the negotiation database, and it may interact with the owner of the node. For nonterminal nodes the output of the procedure is an arc, which the system then traverses to reach a new node. For terminal nodes any output is ignored and the negotiation is terminated.

The success or failure of the negotiation is determined by the semantics of the terminal node. Many negotiations will have several terminal nodes, some representing success and some representing failure. It is important, however, that both participants know the outcome of the negotiation, and so as a special rule of notification, once the result of the negotiation is decided by one participant, the token must travel to a node of the other participant by the time it reaches a terminal state. In this way each participant knows the outcome of the negotiation.

As an example, consider the "access-type" primitive negotiation. In this negotiation, participant one requests access to (i.e., imports) some type exported by participant two. As data the negotiation requires access to an instance of the negotiation schema, a state for component one, and a state for component two. This information is defined by the following types:

```
access-type-info:
  type → string
  category → string
  definition → string
  excuse → string
access-type-state1:
  component → component-name
  type → string
access-type-state2:
  source → component-name
```

The type "access-type-info" is the sole type in the negotiation schema, so it is also the token type for the negotiation. It carries four items of information:

—The "type" is the name of the type to be imported.
—The "category" is either "descriptor," "abstract," or "message."
—The "definition" is either "base" or "derived."
—The "excuse" is an error message in case the access to the type is denied.

The two types "access-type-state1" and "access-type-state2" define the state information for each participant. Since the state and the token are the only items of information passed among nodes, the state must record any information at initiation that is needed by later nodes. In this case, each participant must record its binding to a particular real component, and participant one (the requester) must record the type to which access is requested.

The negotiation interpreter also requires information in order to track the state of the negotiation, and to this end, it uses the following

negotiation-specific types:

```
access-type$participant1-negotiation:
  current-node → negotiation-node
  current-arc → negotiation-arc
  state1 → access-type-state1
  participant2 → component-name
  token → object-name

access-type$participant2-negotiation:
  current-node → negotiation-node
  current-arc → negotiation-arc
  state2 → access-type-state2
  participant1 → component-name
  token → access-type-info
```

The interpreter at each component maintains a collection of information about each instance of negotiation in process, specifically, "access-type$participant1-negotiation" and "access-type$participant2-negotiation." For both participants, the interpreter keeps a record of the node currently containing the token ("current-node") and the last arc traversed ("current-arc"). Additionally, links to the state information are maintained via "state1" or "state2," as appropriate. Further, the "token" is recorded, and hence a link is provided to the entire negotiation database. Finally, the interpeter records the identity of the component bound to the other participant.

A set of three message types is defined for each negotiation. These message types are used by the interpreters on each component to invoke action by the other component. The types are defined as follows:

```
access-type$initialize:
  input maps:
    graph → string
  output maps:
    token → access-type-info
  participant2-negotiation → access-type$participant2-negotiation

access-type$transition:
  input maps:
    graph → string
    participant2-negotiation → access-type$participant2-negotiation
    participant1-arc → string
  output maps:
    participant2-arc → string

access-type$finalize:
  input maps:
    graph → string
    participant2-negotiation → access-type$participant2-negotiation
  output maps:
```

The interpreter initializes the negotiation by first creating an instance of "access-type$participant1-negotiation" to stand for this instance of the negotiation. Next, the interpreter creates an instance of "access-type-state1," passes it to a negotiation-specific initialization procedure specified by the negotiation graph, and finally links that state object to the negotiation instance. The interpreter then sends an "access-type$initialize" message to the second participant to inform it to initialize for the specified negotiation. When the second

participant receives the message, it creates an instance of "access-type-state2," an instance of the token type, and an instance of "access-type$participant2-negotiation." These latter two objects are returned to participant one as the output of the message. Upon receiving the response, the first participant begins to sequence through the graph.

Graph traversal is handled by repeated exchanges of "access-type$transition" messages between the two participants. The essential input from participant one is the arc that it is traversing; the output from participant two is the next arc. As a side effect of the message passing, the appropriate negotiation-specific node action is executed. After the negotiation reaches a terminal state and participant one gets a final response from participant two, the first participant sends an "access-type$finalize" message to the second participant to allow both of them to clean up the residue from the negotiation.

Textually, the negotiation graph for "access-type" is as follows:

```
access-type
  (token = access-type-info,
  state1 = access-type-state1,
  state2 = access-type-state2,
  init1 = save-atype, init2 = savesource,
  final1 = nofin1, final2 = nofin2)

request-access
  (type = start,
  semantics = get-atype,
  owner = 1):
  ready → receive-request

receive-request
  (type = other,
  semantics = test-type-access,
  owner = 2):
  ok → access-granted
  notok = access-denied

access-granted
  (type = terminal,
  semantics = finish-type-access,
  owner = 1):

access-denied
  (type = terminal,
  semantics = explain-type-access-failure,
  owner = 1):
```

The first part, labeled "access-type," is the header of the negotiation. It specifies the type which is the token, the two state types, and the names of the procedures that will handle initialization and finalization for each participant. Following the header is a series of node definitions. Each node specifies the node type (start, other, terminate), the procedure defining the semantics of the node, and the participant that owns the node. After that it specifies the names of the arcs from that node and the destination of each arc. For example, the start state has only one arc, named "ready," and it leads to the node names "receive-request."

The negotiation subsystem provides enough functionality so that it is relatively easy to add new negotiations to the system. It is only necessary to define the negotiation schema, the participant states, the negotiation graph, and the node

semantics procedures. The interpreter then performs remaining functions automatically.

4.9 System-Level Issues

Underlying any instance of the federated architecture must be a real system of hardware and software, and certain system-level mechanisms are needed to support the proper operation of the federation. In [12] three particular system issues are addressed in detail: concurrency control, nested transactions, and object passing. The solutions proposed are straightforward modifications of known or obvious solutions (which is not to say that better solutions are not possible). Concurrency control is a variation on two-phase locking. Nested transactions are handled by associating locks of the inner transactions with the parent transaction. This solution is similar to that described in [24]. Finally, objects are only passed by reference, and all objects are given systemwide unique names derived from the unique name of the component in the federation.

4.10 The Initial Structure of a Federation

In order to function, the federated architecture assumes that each component shares a common set of descriptor types, base types, message types, and negotiations. There are seven primitive descriptor types: *string*, *boolean*, *integer*, *real*, *object name*, *component name*, and *path*. The first four are needed to represent descriptor objects. The type "object name" is a subset of strings representing a unique object name. "Component name" is a subset of strings representing the component names. "Path" is a subset of strings representing a unique type or map name; this unique name is formed from the concatenation of the component name and the type or map name.

Base types and message types may be grouped into three classes: (1) data manipulation, (2) negotiation support, and (3) import and export schemas. Data manipulation types allow a component to perform the data operations on remote objects analogously to operations upon local objects. As described previously, the basic access structure is the cursor. The base type "cursor" represents a set of primitive objects corresponding to cursors. It has no associated maps and can only be manipulated with primitive message types. The type "cursor" is used to access objects of other components.

The message types for data operations implement the operations of the federated database model. They are provided as message types so that they may be exported to other components. These components may in turn use them to navigate through their imported data. Cursors may be manipulated via the following operations: "cursor-create," "cursor-destroy," "cursor-reset," "cursor-next," and "cursor-more." Type-related operators are: "create-object" and "delete-object"; maps are manipulated via "apply," "apply-inverse," "insert-map," and "delete-map."

Most of the types and all of the operations associated with negotiations have been described in the context of the example negotiation "access-type." The only types not mentioned are those that store the negotiation graphs for the interpreter. The type "negotiation-graph" contains one object for every kind of negotiation in the system. Each such object refers to (via string names) the

associated base and message types for each kind of negotiation. In addition, each graph refers to a set of nodes in the primitive type "negotiation-node," which in turn refers to the set of arcs in "negotiation-arc."

The import schema for each component is represented by a collection of types and maps defining the imported types and maps. In effect, there is a metaschema for defining the import schema. In this metaschema, the type "import-schema" has one object for each connected component. Each of these objects refers in turn to a set of "import-types," which in turn refer to a set of "import-maps." Associated with these imported types and maps is defining information (definition, category, and constraints).

The export schema is similar to but slightly simpler than the import schema. Since there is only one exporter (per component), there is no need for a type "export-schema." Thus only the types "export-type" and "export-map" are needed. These types differ from the import types by the addition of a connection list map specifying the names of the other components currently importing a given type or map.

Manipulation of the import schema and the export schema is carried out by a specific set of negotiations. One set allows the exporter to augment the export schema or to reduce it by withdrawing previously exported types and maps. Another set allows the importer to augment the import schema and to reduce it. The complete set is as follows:

—*Bootstrap* supports the initial negotiation between the federal dictionary and a component entering the federation.

—*Connect* links two components in the federation; it causes them to exchange import and export interface information so that each may determine what data is provided by the other.

—*Disconnect* unlinks two components; typically this is done as part of a sequence of actions when a component plans to leave the federation.

—*Withdraw-type* notifies others that a component plans to withdraw a type from its export schema.

—*Withdraw-map* notifies others that a component plans to withdraw a map from its export schema.

—*Access-type* requests access to a type exported by some component.

—*Access-map* requests access to a map exported by some component.

—*Release-type* notifies an exporter of a type that some component no longer wishes to import it.

—*Release-map* notifies an exporter of a map that some component no longer wishes to import it.

5. A PROTOTYPE IMPLEMENTATION

Completely implementing the federated architecture requires the use of a network of computers with each computer supporting a semantic database system. Neither the database nor the network was originally available when the federated architecture was designed. In consequence, a modest experimental prototype was produced. This existing prototype is a large program written in Franz-LISP under

the Berkeley UNIX[5] operating system. The prototype supports a simple database implementing the federated information model, export and import schemas, message passing, and negotiations. In [12] an exhaustive, annotated transcript is provided of the execution of the prototype. The prototype has successfully executed access to imported data, simple derivations, message types, all of the built-in negotiations, and a sample negotiation based on a shared database of parts, suppliers, and consumers.

As a further test of its utility, the federated architecture is now being used as the basis for a distributed software engineering database [6]. A typical software project may involve a number of programmers each working on his or her own part of the software but also using certain pieces of code and data provided by other programmers. The federated architecture, with its emphasis on autonomy and partial sharing, is a natural structure for such programming environments.

This software engineering system prototype is a combination of the federated architecture with Odin [5], which is an extension of the UNIX "make" facility. The prototype runs on a network of Sun workstations running Berkeley UNIX 4.2, which provides both the hardware and software necessary to support distributed programs. In this prototype, there are three processes per machine: a user process (running Odin), a local database server process, and a federation server process. The user process provides the interface between the user and the local database server on one hand, and the user and the rest of the federation on the other hand. The local server performs requests generated by the user. The federation server handles requests from other components for exported information and negotiations. The local server and the federation server are designed to access the local database in parallel to provide better response for local requests. In consequence, requests from the local user are handled immediately. Requests from other components are multiplexed by the federation server. This means that intercomponent requests (e.g., for data transfer or negotiation) may not execute immediately. If a greater degree of concurrency is desired, then additional federation server processes can be added. In the extreme there may be one federation server for each known external component database.

6. CONCLUSIONS

While there is no production version of a federated architecture in use by a large body of users, it is nevertheless possible to assess how well the original goals are met by the architecture presented in this paper. It is also possible to see the parts of the architecture that are not completely successful and should be changed in some future version. Recall that in the absence of a central authority, the federated architecture has to resolve two conflicting requirements: (1) the components must maintain as much autonomy as possible, but (2) the components must be able to achieve a reasonable degree of information sharing. Autonomy specifically refers to four capabilities: control of data sharing, control of data viewing, cooperative activity, and support for structural evolution.

The principal architectural features in support of autonomy are the export and import interfaces of components. The export interface directly supports the

[5] UNIX is a trademark of AT&T Bell Laboratories.

requirement for control of data sharing. All accesses to some item of data must ultimately reference the component containing that item and hence are under the control of that component. The export interface also supports cooperation by providing a barrier between the private data of the component and all the other components. As long as a component maintains its interface contract with the federation, it is free to change the structure of its private data.

The role of the import interface is not as direct as that of the export schema. Its primary function is to support directly the requirement that each component be able to define its own view of the directly available data (i.e., without the use of a global schema). As a secondary function, the import schema focuses the attention of each component upon that exported data of immediate interest to it. If every component indiscriminately imported all the available information, which is equivalent to having no import schema, then it would be difficult for a component to deal with all of the information. It would also be difficult for other components to know who was a potential user of its data, thus inhibiting the evolution of a federation.

Once the concept of export and import interfaces is integrated with the database, the idea of importing simple data items follows immediately. But extending importation to include transactions is not quite so obvious, at least from the database point of view. Traditionally, databases have kept the transactions quite separate from the database structure. This is a result of the traditional emphasis on long-term data independence of the operations that manipulate it. The federated architecture, along with other work in office database systems (e.g., [35]), integrates communication facilities with the database.

Any discussion of sharing must also consider the exchange of metadata, namely, data representing the structure of the data, as opposed to the actual data. The federated architecture allows metadata to be exported so that other components may peruse the structure of exported data. Metadata are supported by a set of types in the built-in structures of the architecture. In this way, they may be shared using the normal export–import mechanisms.

Negotiation is another key feature of the federated architecture. Initially, the negotiation subsystem was to be a monolithic program that had all possible negotiations embedded within it. This approach, although feasible, does not allow the users easily to discern the structure of negotiations, and makes it difficult to add new kinds of negotiations. The negotiation subsystem has thus been divided into two parts: an interpreter, and a collection of procedures written in the negotiation language (negotiation graphs). A problem with the approach is that it does not go far enough. Currently, the semantics of a node is described as a string representing a host-language procedure to be executed. This means that a user must know that language in order to understand existing negotiations and to write new ones. Further, much information about the meaning of a negotiation state is hidden in those procedures. The system would be more uniform if the semantics was specified using the structures provided by the database model. In effect this would build in a programming language into the database model, and its programs would be manipulated in the same way as any other database structure.

The lack of multiparticipant negotiations is another problem with the current architecture. There are cases in which negotiations need to be carried out

simultaneously by three or more participants. Modifying the system to handle n participants, for n fixed, is straightforward; the problem is handling negotiations that require varying numbers of participants or that need to be "quantified" over all the components currently in the federation.

An additional limitation of the negotiation system is its handling of exceptional conditions. Since it is operating in a distributed environment, many kinds of failures can occur: lost or duplicated messages, component failures, and network partitioning. At the moment, the only way to deal with these during a negotiation is to introduce explicit failure arcs to all the nodes in the graph. This seriously complicates the structure of such graphs. One alternative currently being explored is to introduce special nodes to the graph that can handle such errors, but need not have explicit arcs leading into them. Thus a typical negotiation graph would have the main graph plus a separate collection of graphs to handle various kinds of failures.

Finally, it is possible to compare the federated architecture with the list of the benefits of the logically centralized architectures (such as composite systems). A comparison shows two benefits that have been partially lost: removal of redundancy and providing a global resource. Since global data is directly counter to the goals of a federation, the latter loss seems inevitable. The redundancy problem has two essential aspects. First, two components may export the same information, kept separately. It may be desirable to relate these two versions by electing one of them to export the data and have the others keep their versions local. Second, for efficiency, it may be desirable to allow an importing component to keep a local copy of the shared data. It is clear from some work on federated software environments that this form of redundancy is desirable. To this end, new mechanisms (principally negotiations) are currently being added to support duplicated data.

ACKNOWLEDGMENTS

The authors would like to gratefully acknowledge the comments and suggestions of several individuals on earlier versions of this paper: Hamideh Afsarmanesh and Amihai Motro (USC); Robert Balzer, Neil Goldman, and David Wile (USC Information Sciences Institute); Roger King (University of Colorado, Boulder); Withold Litwin (INRIA); and Peter Lyngbaek (Hewlett-Packard Research Laboratories). The very useful recommendations of the *ACM Transactions on Office Information Systems* referees and editors are also most appreciated.

REFERENCES

1. ARITEBOUL, S., AND HULL, R. IFO: A formal semantic database model. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Apr. 1984). ACM, New York, pp. 119–132.
2. BRODIE, M. L., MYLOPOULOS, J., AND SCHMIDT, J. W. (ED.). *On Conceptual Modelling.* Springer-Verlag, 1984.
3. BUNEMAN, P., AND FRANKEL, R. E. A functional query language. In *Proceedings of the International Conference on Management of Data* (Boston, Mass., May 30–June 1, 1979). ACM, New York, pp. 52–57.
4. CHAMBERLIN, D. D., GRAY, J. N., AND TRAIGER, I. L. Views, authorization, and locking in a

relational database system. In *Proceedings of the National Computer Conference* (June 1975). AFIPS Press, Reston, Va., pp. 425–430.

5. CLEMM, G. M. ODIN—An extensible software environment: Report and user's manual. CU-CS-262-84, Computer Science Dept., Univ. of Colorado, Boulder, Colo., March, 1984.
6. CLEMM, G., HEIMBIGNER, D., OSTERWEIL, L., AND WILLIAMS, L. Keystone: A federated software environment. In *ACM SIGPLAN Symposium on Programming Languages and Programming Environments* (Seattle, Wash., May 1985). ACM, New York.
7. DAYAL, U. AND BERNSTEIN, P. A. On the updatability of relational views. In *Proceedings of the 4th International Conference on Very Large Databases* (West Berlin, Sept. 1978). ACM, New York, pp. 368–377.
8. GIBBS, S., AND TSICHRITIZIS, D. A data modelling approach for office information systems. *ACM Trans. Office Inf. Syst. 1*, 4 (Oct. 1983), 299–319.
9. GRAY, J. N. Notes on data base operating systems. In *Operating Systems: An Advanced Course*, Lecture Notes in Computer Science, vol. 60. Springer Verlag, 1978, pp. 393–481.
10. HAMMER, M., AND MCLEOD, D. On database management system architecture. In *Infotech State of the Art Report: Data Design*, Infotech State of the Art Reports, vol. 8. Pergamon Infotech Limited, Maidenhead, United Kingdon, 1980, pp. 177–202.
11. HAMMER, M., AND MCLEOD, D. Database description with SDM: A semantic database model. *ACM Trans. Database Syst. 6*, 3 (Sept. 1981), 351–386.
12. HEIMBIGNER, D. M. A federated architecture for database systems. Ph.D. dissertation, Univ. of Southern California, Los Angeles, Calif., Aug. 1982.
13. HEIMBIGNER, D., AND MCLEOD, D. Federated information bases—A preliminary report. In *Infotech State of the Art Report: Database.* Infotech State of the Art Reports, vol. 9. Pergamon Infotech Limited, Maidenhead, United Kingdom, 1981, pp. 383–410.
14. KATZ, R., AND GOODMAN, N. View processing in multibase—A heterogeneous database system. In *An Entity–Relationship Approach to Information Modelling and Analysis*, ER Institute, 1981, pp. 259–280.
15. KIMBLETON, S. R., WANG, P. S. C., AND FONG, E. XNDM: An experimental network data manager. In *Proceedings of the Berkeley Workshop on Distributed Data Management and Computer Networks* (Berkeley, Calif., Aug. 1979). Pp. 3–17.
16. KIMBLETON, S. R., WOOD, H. M., AND FITZGERALD, M. L. Network operating systems—An implementation approach. In *Proceedings of the National Computer Conference* (June 1978), AFIPS Press, Arlington, Va., pp. 773–782.
17. KING, R., AND MCLEOD, D. A database design methodology and tool for information systems. *ACM Trans. Office Inf. Syst.1*, 1 (Jan. 1985), pp. 2–21.
18. KING., R., AND MCLEOD, D. Semantic database models. In *Database Design*, S. B. Yao, Ed. Prentice Hall, Englewood Cliffs, N.J., 1985.
19. LIEN, Y. E., AND YING, J. H. Design of a distributed entity–relationship database system. In *Proceedings of the International Computer Software and Applications Conference* (Chicago, Nov. 1978). IEEE, New York, pp. 277–282.
20. LINDSAY, B., AND SELINGER, P. G. Site autonomy issues in R*: A distributed database management system. Res. Rep. RJ2927, IBM Research Lab, San Jose, Calif., Sept. 1980.
21. LITWIN, W. A model for distributed data bases. In *Proceedings of the ACM 2nd Annual Louisiana Computer Exposition* (Feb. 1980). ACM, New York, pp. 1–36.
22. LITWIN, W. Logical design of distributed data bases. MOD-1-043, INRIA, Paris, France, July 1981.
23. LYNGBAEK, P., AND MCLEOD, D. Object sharing in distributed information systems. *ACM Trans. Office Inf. Syst. 2*, 2 (Apr. 1984), 96–122.
24. MOSS, E. B. Nested transactions: An approach to reliable distributed computing. Ph.D. Dissertation, Massachusetts Institute of Technology, Cambridge, Mass., Apr. 1981.
25. MOTRO, A., AND BUNEMAN, P. Constructing superviews. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data* (Ann Arbor, Mich., Apr. 1981), ACM, New York, pp. 56–64.
26. MYLOPOULOS, J., BERNSTEIN, P. A., AND WONG, H. K. T. A language facility for designing database-intensive applications. *ACM Trans. Database Syst. 5*, 2 (June 1980), 185–207.
27. NAVATHE, S. B. Schema analysis for database restructuring. *ACM Trans. Database Syst. 5*, 2

(June 1980), 157–184.

28. OPPEN, D. C., AND YOGEN, Y. K. The clearinghouse: A decentralized agent for locating named objects in a distributed environment. *ACM Trans. Office Inf. Syst. 1*, 3 (July 1983), 230–253.
29. ROTHNIE, J. B., JR., BERNSTEIN, P. A., FOX, S., GOODMAN, N., HAMMER, M., LANDERS, T. A., REEVE, C., SHIPMAN, D. W., AND WONG, E. Introduction to a system for distributed databases (SDD-1). *ACM Trans. Database Syst. 5*, 1 (Mar. 1980), 1–17.
30. ROTHNIE, J. B., JR., AND GOODMAN, N. A survey of research and development in distributed database management. In *Proceedings of the 3rd International Conference on Very Large Databases* (Tokyo, Japan, Oct. 1977). IEEE, New York, pp. 48–62.
31. ROWE, L. A., AND SHOENS, K. A. Data abstraction, views, and updates in Rigel. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data* (Boston, May 1979). ACM, New York, pp. 71–81.
32. SHIPMAN, D. The functional data model and the the data language DAPLEX. *ACM Trans. Database Syst. 2*, 3 (Mar. 1981), 140–173.
33. SMITH, J. M., BERNSTEIN, P. A., DAYAL, U., GOODMAN, N., LANDERS, T., LIN, K. W. T., AND WONG, E. Multibase: Integrating heterogeneous distributed database systems. In *Proceedings of the National Computer Conference* (June 1981). AFIPS Press, Reston, Va., pp. 487–499.
34. STONEBRAKER, M. R., AND NEUHOLD, E. A distributed database version of INGRES. In *Proceedings of the Berkeley Workshop on Distributed Data Management and Computer Networks*, (Berkeley, Calif., May 1977). University of California, Berkeley, pp. 19–36.
35. TSICHRITZIS, D. C. Integrating data base and message systems. In *Proceedings of the International Conference on Very Large Databases* (Cannes, France, Sept. 1981). IEEE, New York, pp. 356–362.

Received December 1984; revised June 1985; accepted June 1985