The DIOM Approach to Large-scale Interoperable Database Systems

Ling Liu
Department of Computing Science
University of Alberta
GSB 615, Edmonton, Alberta
T6G 2H1 Canada
email: lingliu@cs.ualberta.ca

Calton Pu
Dept. of Computer Science and Engineering
Oregon Graduate Institute
P.O.Box 91000 Portland, Oregon
97291-1000 USA
email: calton@cse.ogi.edu

Abstract

A large-scale interoperable database system operating in a dynamic environment should provide a uniform access user interface to its components, scalability to larger networks, evolution of database schema and applications, composability of client and server components, and preserve component autonomy. To address the research issues presented by such systems, we introduce the Distributed Interoperable Object Model (DIOM). DIOM promotes an adaptive approach to interoperation and mediation [34, 35], aimed at improving the robustness and scalability of the interoperation services used for integrating and accessing heterogeneous information resources. DIOM's main features include the explicit representation of semantics in data sources through the DIOM base interfaces, the use of interface composition mechanisms to support the incremental design and construction of compound interoperation interfaces, the decoupling of semantic heterogeneity from the representational mismatch, the deferment of semantic conflict resolution to the query submission time instead of at the time of information gathering and integration, and a clean interface between distributed interoperable component objects that supports the independent evolution and management of such objects. To make DIOM concrete, we outline the Diorama architecture, which includes important auxiliary services such as domain-specific library functions, object linking databases, and query decomposition and packaging strategies. Several practical examples and application scenarios illustrate the usefulness of DIOM.

Index Terms: *Large scale interoperable database systems, Distributed object management, Interoperation, Mediation, Scalability, Composability, System evolution, Object-oriented Data Model.*

1 Introduction

Many papers on multidatabase systems or federated database systems [3, 10, 25, 28], focused on the resolution of schematic and semantic incompatibilities among autonomous and heterogeneous component databases in relatively static system configurations. While this assumption was reasonable for the past, recent developments in the Internet show that in the present environment a useful solution can no longer ignore the issues of scalability and evolution. For the upcoming National Information Infrastructure, these issues will become paramount. For example, in an effort to make US government databases easily accessible to the general public, many thousands of federal, state, and local government databases will be interconnected. Even if each participating agency changes slowly, the whole government interoperable database will be a highly dynamic growing and evolving system.

In a modern networking environment as such, we consider *interoperable database systems*, software systems that provide facilities and techniques to interoperate with several individual databases (called *component databases* in this paper), *and* at the same time offer convenient uniform access to a number of remote data sources. We believe that as the number of participating databases in the interoperation increases rapidly, and the interoperable environment evolves dynamically, the issues of system-wide scalability and evolution, as well as component composability and autonomy become necessarily an integral part of large-scale interoperable database systems. We refer to the properties of Uniform access, Scalability, Evolution, Composability and Autonomy as USECA properties. In large-scale dynamic interoperable database systems, USECA properties are critical requirements, without which the system becomes much less useful.

1.1 USECA Properties as Research Challenges

With rapid advances in computer network technology, information systems connected by world-wide high-speed networks provide great opportunities for up-to-date and rapid access to remote information resources. It is highly desirable for users and applications to be able to work with remote data sources and request services across networks as if they were working with a single database system. We refer to this requirement as **uniform access** property. Besides uniform access, several challenges arise in the context of large-scale interoperable database systems, including scalability and evolution of the system as a whole and its components, as well as the flexibility in composing components and the autonomy of individual components. Each of the USECA properties (Uniform access, Scalability, Evolution, Composability, and Autonomy) is important for the usefulness of a large-scale interoperable database system and each presents a serious research challenge.

Consider **Scalability**. A large-scale interoperable database environment (e.g., three hundreds databases as opposed to three databases) presents challenging questions to the viability of loose-coupling approach to system development (cf. [16, 17]) and the tight-coupling integration framework that concentrates primarily on circumventing schematic and semantic heterogeneity (cf. [14, 29, 30, 31, 32]). As the number of databases participating the interoperable database system increases, the design of an integrated schema involving n different systems requires to reconcile an order of n^2 possibly conflicting representations(i.e., heterogeneity in semantics or in data formats). Such activities are time consuming and can be aggravated when incorporating the system evolution issue with the integration strategies.

Similarly, **system evolution** presents an equally important problem. While the number of data bases available via networks are increasing swiftly, some component databases may also become obsolete or temporarily inaccessible. Furthermore, even when the changes in each individual component database schema are not frequent, the large number of component databases may add up to surprisingly frequent schema change events at the interoperable database system level. Assuming there is only one change in two years for any component database schema, an interoperable system with two hundred databases as such will have to contend with one hundred changes every year, which corresponds to, on average, two changes every week!

Another important challenge is **system composability**, and more specifically, the need for incremental design and construction of interoperation interfaces. This is simply because an interoperable database system like any other software system is not constructed in "one shot". Besides, multiple data integration efforts to produce multiple kinds of integrated systems, which involve overlapping subsets of their component databases, become increasingly desirable for those organizations that need to perform multiple forms of integration with overlapping sets of data. Therefore, the interface definition language of the interoperable database systems should provide interaction abstraction facilities to allow users or applications to define new or complex interfaces in terms of existing or simpler interfaces, for instance, through interface refinement (specialization) or interface composition (aggregation and generalization) mechanisms.

Autonomy and Flexibility. In addition, not only should component systems and their customer set not pay high price on autonomy in order to join the interoperation (or integration), but the remote users should also not have to compromise the way they prefer to understand and represent their application semantics and be forced to surrender to a system-supplied, canonical integration schema. This implies two interesting points. First, to preserve the autonomy of individual systems, it is essential to allow an individual systems to treat the external users and applications and thus the interoperable systems in a similar way as its ordinary clients unless specified otherwise. Thus, the evolution of individual system can be carried out without being interfered by the other parties in the interoperation. Second, to respect the freedom and the flexibility in the use of the interoperable system, it is important to provide data model independent language constructs and facilities in the interoperable object model, which enable users and applications to request services and remote data sources either directly or through the construction of new interfaces that present their own interest and desired representation for accessing interoperable objects.

Of course, our list of critical issues is not exhaustive. For example, efficient query processing is also one of the key concerns in large-scale interoperable database systems, especially when queries involve partial matching, ranking or fault tolerance in terms of quality or relevance of the data from diverse sources. The query optimization strategies that are possible in a small and controlled environment, such as those for "canned" queries to be optimized may not be practical any more, when the environment evolves dynamically, the number of databases increases rapidly, and the autonomy of component systems needs to be fully respected. A discussion of other interesting issues on interoperation and mediation may refer to a recent paper by Gio Wiederhold [35]. We believe that these research challenges are non-trivial in large-scale interoperable systems and deserve special attention in order to provide the cost-effective and robust support for the longer life span of interoperable systems.

1.2 Contribution and Organization of the Paper

Our main contribution in this paper is an approach called *Distributed Interoperable Objects Model* (DIOM), to support the USECA properties of large-scale interoperable database systems. DIOM's main idea is to incorporate system-wide scalability and evolution support as well as component composability into an interoperation framework. Significant features of the DIOM approach include:

- the explicit representation of and access to semantics in data sources through the DIOM base interfaces,
- the use of interface composition mechanisms to support incremental design and construction of compound interoperation interfaces,
- the decoupling of semantic heterogeneity from the representational mismatch, and
- the deferment of conflict resolution to the query submission time rather than at the time of schema integration.

- the facilities that allow applications to set up their own interoperation interface schema at will, rather than being constrained to take the system-supplied monolithic global view of all the information handled by the system.

As a result, when the number of databases that participate in the interoperation increases, the existing applications developed earlier may continue to work without source code changes or recompilation. The new applications may define their interoperation interfaces by aggregation or generalization of existing interfaces, or by refinement of pre-defined interfaces, rather than starting from scratch. Autonomy of the component systems is also fully respected such that component systems can evolve without consultation with the interoperable system. Moreover, principled and disciplined evolution of an individual component schema will cause no or minimal impact on its remote customers. Thus, the rewriting of the application programs developed through interoperable object interfaces can be avoided or minimized in the presence of system evolution.

The rest of the paper proceeds as follows: In order to demonstrate the practical applicability of the DIOM approach, we start with an overview of the Diorama architecture in Section 2. Diorama is an ongoing project that aims at implementing a prototype of the DIOM approach to large scale interoperable database systems with USECA properties. The distributed interoperable object model (DIOM) and the interface composition mechanisms are described in Section 3 through a number of illustrative examples. Section 4 explains query compilation in DIOM databases under the Diorama architecture. We compare our approach with related work in Section 5 and conclude the paper in Section 6.

2 The Diorama Architecture

In terms of object-oriented terminology, *interoperability* refers to the ability to exchange requests between objects and to enable objects to request services of other objects, regardless of the language in which the objects are defined and their physical location (e.g., hardware platforms, operating systems, DBMS's).

Distributed interoperable objects are objects that support a level of interoperability beyond the traditional object computing boundaries imposed by programming languages, data models, process address space, and network interface [1]. The abstractions of distributed interoperable objects are captured in the Distributed Interoperable Object Model (DIOM), described in Section 3.

Diorama is an interoperable software architecture based on DIOM to support USECA properties. In several ways, Diorama facilitates the construction of applications and database systems from data sources and services provided by different software vendors or application systems. First, it uses DIOM's interface definition language and interface query language to support uniform and transparent access to data sources across Internet. Second, Diorama provides facilities and convenient language constructs to support the explicit representation of and access to disparate data semantics, and encourages to defer the conflict resolution to the time when query is submitted. Third, it offers a number of adaptive software development techniques to support the autonomous evolution of individual components, such as specialization mechanism for interface refinement, aggregation mechanism for interface composition, and adaptive transformation of DIOM interfaces into the skeleton bindings in the intended implementation language. Fourth, Diorama supports composability with dynamic linking and binding of interoperable objects. Last but not least, it preserves autonomy through a clean interface between distributed interoperable objects that supports independent evolution and management of such objects.

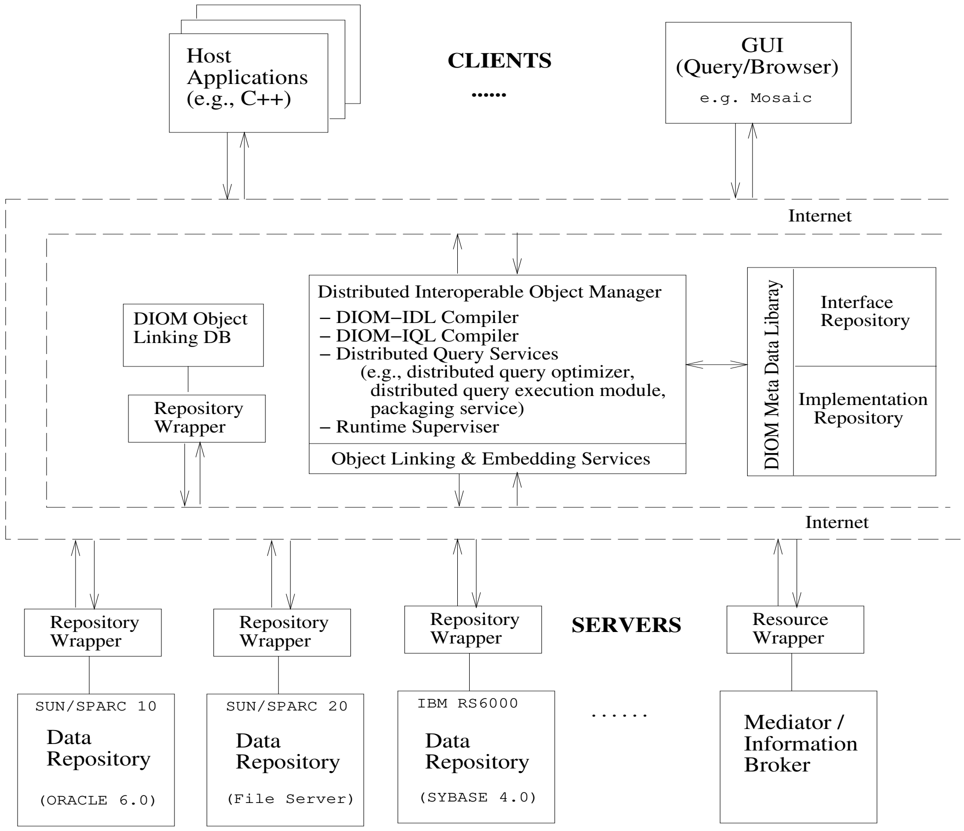


Figure 1: The Diorama system architecture

Figure 1 shows a simplified picture of the Diorama architecture. At the leaf level of Figure 1 we have the data sources that Diorama integrates. Examples of data sources include data repositories managed by relational database systems, object-oriented database systems, file systems, document managers, image managers, video servers, and information resources gathered by other mediators (e.g., TSIMMIS [11]) or information brokers (e.g., Harvest [2]). Above each repository or mediator is an information resource wrapper. The main task of a *resource wrapper* is to control and facilitate the external access to the data source, based on the export schema provided by the data source manager at its integration into the interoperable system. Diorama approach allows new sources to become useful to the DIOM users as soon as a wrapper is supplied. This approach allows a flexible integration of information resources gathered by other mediators or resource discovery systems [2, 11, 33]. For a further discussion, refer to Section 5.

In the middle right of Figure 1, the DIOM meta object library consists of DIOM interface repository and DIOM implementation repository. The interface repository supports dynamic linking of object interfaces (e.g., which data sources provide relevant information requested by a given application). It could be based on a commercial product, such as OLE or OpenDoc, for distributed dynamic object linking or embedding purpose. The DIOM implementation repository contains information that allows the object request linking subsystem to actually locate and activate objects to fulfill the requests through dynamic loading.

In the middle left of Figure 1, the DIOM object linking database manages the complex relationship objects that most DIOM applications require to glue together the data from multiple repositories in some new or useful ways. For example, in a medical insurance application, the DIOM object linking database could be used as a persistent storage to hold a `ClaimFolder` object for some special medical treatment of a patient, which an insurance agent needs to handle. A `ClaimFolder` object could be formed, for example, by linking the following data that are stored in three separate data repositories:

- a patient's X-ray images stored in an image-specific file repository of a radiological lab,
- a doctor's diagnosis report stored in a document management system at doctor's office, and
- a patient's claim and the corresponding insurance agreement records stored in a relational data base of the insurance company.

A DIOM database is a USECA view of clients on the interoperable database system. Diorama is responsible for presenting the applications with a unified object-oriented view of the contents of DIOM databases, and to process users' and applications' requests (e.g., queries, updates on object links, and method invocations) on the content of a DIOM database. The content of a DIOM database is defined using the DIOM interface definition language (IDL). A DIOM database consists of a collection of DIOM objects. DIOM objects are composed through dynamically binding data from multiple data repositories provided by the underlying component systems. All queries and request for data and services may be formulated using DIOM interface query language (IQL). The DIOM-IDL compiler first checks the validity and the type-closure of each import specification in the DIOM interfaces, generates the complete import schema, and then performs mapping of IDL syntax into the DIOM repository implementation language. The DIOM-IQL compiler generates the stubs and skeleton bindings in the repository implementation language and performs the actual mapping of DIOM-IQL syntax into the implementation skeletons that can be understood by the repository wrappers of the target data repositories. We will illustrate the functionality and the use of DIOM IDL and IQL in the later sections.

Diorama's components are designed in the client/server style and dynamically bound to support independent component evolution. On the server side, IDL syntax is mapped into the repository implementation language and generate the implementation skeletons, which will be sent to the corresponding target data repositories. These implementation skeletons can be understood by the data repository wrappers and easily be mapped into a collection of data access functions applicable to the corresponding data repositories. On the client side, DIOM generates the interface stubs that specify the method signatures to invoke methods of target data repositories. For instance, DIOM interfaces to C++ applications may be a set of C++ classes that act as "surrogates" for the corresponding definition of the DIOM interface schema. When a component data repository evolves into a new version, and the changes at the component level need to be reflected at the interoperation interface level, a new DIOM interface library or an interface schema is released as a new interface version that co-exists with the existing versions. Thus, the existing DIOM applications may continue to work without source code changes or recompilation.

End users (top of Figure 1) may access multiple information sources either by writing application programs that request DIOM objects or by using the Diorama browsing toolkit, which includes interactive browsing, navigation, and querying of the DIOM objects. The first implementation of the DIOM browser is currently developed using Netscape, a HTML-based toolkit. It is a platform-independent tool for displaying and requesting the DIOM objects. By entering DIOM, the Internet users can use a uniform interface language to search and integrate heterogeneous information from multiple, disparate data sources, rather than obliged to learn many different languages. DIOM meta library provides available index information of all data sources gathered either directly by the Diorama mediator or linked to the Diorama mediator via the other mediators or information brokers. DIOM users may use DIOM/IQL to issue join queries across multiple information sources managed by several mediators.

Generally speaking, the DIOM database interface schemas are *virtual* schemas, since the data contents are actually stored and managed by individual data repository manager (such as the individual DBMSs).

The DIOM object linking database is created and maintained only when the DIOM applications need to have frequent access to some linking information that are not provided or cannot be derived automatically through access to the individual data repositories. The DIOM linking database should be considered just as yet another persistent data repository in the DIOM architecture. In the first phase, we intend to implement the link database using an object-oriented database management system such as ObjectStore [15].

Note that in Diorama, there is neither a single global database containing the integrated information nor even a global database schema that describes all the information accessible via Diorama. Each application may set up its own interoperable interface schema using DIOM/IDL. One of the key points of DIOM application to information gathering is the fact that no person or software component needs to have a global view of all the information handled by the system.

We now explain in more detail the basic concepts of DIOM and its mechanisms to further address questions like:

- How does DIOM enable cross-database interoperability?
- What makes DIOM such a useful and unifying model?
- How does DIOM enhance the adaptiveness and robustness of the interoperable systems and their applications in the presence of underlying data repository-based evolution or interoperation-based evolution (e.g., the evolution of the interoperation interface definitions)?

3 The Distributed Interoperable Object Model

The purpose of DIOM development is to support the construction of a USECA view of data from disparate data sources, rather than inventing yet another object-oriented data model. Recently, a consortium of major OODBMS vendors proposed a candidate for such a model, called ODMG-93 object database standard [8, 13], which was developed by adding database features to the OMG object model standard. Therefore, we adopt the ODMG-93 proposal as a starting point, and add a number of extensions (such as the *import* mechanism and the aggregation mechanism) for interface composition, in order to build integrated views of data across multiple data sources and to support USECA properties.

3.1 A Quick Look at the Model

As in the ODMG-93 standard, DIOM's basic entity is the *object*. Every object has a unique identity, the object identifier (oid), that can uniquely distinguish the object, thus enabling the sharing of objects by reference. Objects are strongly typed. All objects of a given type exhibit similar behavior and a common range of states. The behavior of objects is defined by a set of *operations* that can be executed on an object of the type. The state of an object is defined by a set of *properties*, which are either *attributes* of the object itself or *relationships* between the object and one or more other objects. Changing the attribute values of an object, or the relationships in which it participates, does not change the identity of the object. It remains the same object.

An object type is described by an interface and one or more implementations. An *interface* can be seen as a strongly typed contract between objects of similar behavior or between objects that need to corporate with each other in order to accomplish a task. It defines properties of a type, properties of the instances of the type, and operations that can be invoked on them. An implementation defines internal data structures, and operations that are applied to those data structures to support the externally visible state and behavior defined in the interface. The combination of the type interface specification and one of the implementations defined for the type is referred to as a *class*. The use of the term class allows a subset of the DIOM model to be consistent with C^{++}.[1] Multiple implementations for a type interface is useful to support databases that span networks which include machines of different architectures, mixed languages and mixed compilers environments.

Consider the `Claim Folder` example mentioned in Section 2. Figure 2 shows the contents of the three data repositories in this example:

1. a patient's insurance agreement and claims for special medical treatments in a relational database,
2. a collection of X-ray images associated with each claim stored in a file-based image repository,
3. a doctor's diagnosis report stored in a C^{++} based document repository.

Figure 3 shows a sample interface schema for the `Claim Folder` application. This schema contains an interface definitions for each type of data in the underlying data repositories. We assume that each claim may contain more than one X-ray image but involve only one doctor's diagnosis report. There are a number of ways to create `ClaimFolder` objects by linking the patients' claims in the relational database

[1] A C^{++} class has a single *public part* and a single *private part*. The private part corresponds to the implementation of a class in DIOM.

Relational DBMS

```
CREATE TABLE Insur-Agreement
(  insur-no      Integer NOT NULL,
   patient-name  Char(40),
   patient-add   Char(80),
   ins-category  Integer,
   start-date    Date,
   PRIMARY KEY (insurance-no)
);

CREATE TABLE Claim
(  insur#         Integer NOT NULL,
   claim-no       Integer NOT NULL,
   claim-date     Date,
   status         Char(15),
   treatment      Char(40),
   PRIMARY KEY (insur#, claim-no),
   FOREIGN KEY (insur#) REFERENCE
     Insurance-Agreement(insur-no)
);
```

Image Repository

```
class ImageSpec {
  private:
    ...
  public:
    ImageSpec(char *);
    ImageCaption(char);
     ...
}

class Image {
  private:
    ...
  public:
   void display(void *);
   int find-match(Char *keywords);
   ...
}

mkdir MedicalInsur/images/Xrays
import_image ~/MedicalInsur/images/Xrays claim1.gif
import_image ~/MedicalInsur/images/Xrays claim2.gif
...
```

Document Repository

```
class DiagnosisRep {
  private:
    ...
  public:
   char *text;
   int search(char *expr);
   ...
}

make_rep_db /MedicalInsur/documents
add_doc /medicalInsur/documents/claim1.text
add_doc /MedicalInsur/documents/claim2.text
...
```

Figure 2: The data repository contents relevant to the medical claim folder scenario

with their associated X-ray images and the corresponding doctors' diagnosis reports. For example, application developers may use the base interface definitions in Figure 3 to design the `ClaimFolder` objects within their C++ programs. It is also possible to use the DIOM interface aggregation abstraction mechanism to establish the links among the three underlying data repositories. Section 3.3 provides more details.

3.2 Basic Concepts

3.2.1 Types and Type Properties

Types are themselves DIOM objects and may have properties themselves. For instance, each type may have the following three properties:

- *Supertypes.* Object types can be defined by stepwise refinement, which forms a subtype/supertype graph. All attributes, relationships, and methods defined on a supertype are inherited by its subtypes. A subtype may add additional properties and operations to introduce behavior or state

```
CREATE INTERFACE Insur-Agreement
(   EXTENT  Insur-Agreement
    KEY     Insur-no): persistent
{
   ATTRIBUTES
     Integer insur-no;
     String  patient-name;
     String  patient-add;
     Integer ins-category;
     Date    start-date;
);

CREATE INTERFACE Claim
(   EXTENT  Claims
    KEY     insur#, claim-no): persistent
{
    ATTRIBUTES
      Integer insur#;
      Integer claim-no;
      Date    claim-date;
      String  status;
      String  treatment;
);

CREATE INTERFACE DiagnosisRep
(   EXTENT DiagnosisReps ): persistent
{
    ATTRIBUTES
      String insur-no;
      String text;
      ...

    OPERATIONS
      int *search(char *expr);
   ...
};
```

```
CREATE INTERFACE ImageSpec
(   EXTENT ImageSpecs ): transient
{
    ATTRIBUTES
     ...

    OPERATIONS
      void *ImageSpec(Char *);
      Char *ImageCaption(Char);
        ...
}

CREATE INTERFACE Image
(   EXTENT Images ): persistent
{
    ATTRIBUTES
      Integer InsurNum;
      ...

    OPERATIONS
      void *display(void *);
      int *find-match(Char *keywords);
       ...
}
```

Figure 3: A sample interface definition in DIOM-IDL

unique to instances of the subtype. It may also "override" the properties and operations it inherits to make them specialized to the behavior and range of state values appropriate for instances of the subtype. The formal semantics of inheritance along with subtype/supertype graph is based on the usual notion of subtyping [4]. Figure 4 shows the *subtyping* relationships between types, and the *instance-of* relationship between types and their instances.

- *Extents.* The extent of a type denotes the set of all instances of the type. By including an extent declaration in the type definition, the type definer instructs the DIOM system to automatically maintain a current index to the members of this set. The actual maintenance of the extent depends on the object binding options specified for the extent: *persistent* or *transient.* The `persistent` option will have the objects stored and managed by the link database, whereas the `transient` option implies dynamic binding, loading, and packaging the objects requested to the target applications. If an object is an instance of type `T`, then it is a member of the extent of `T`. Similarly, if a type `T` is a *subtype* of type `S`, then the extent of `T` is a *subset* of the extent of `S`.

- *keys.* A key of an object type is a property or a set of properties that together uniquely identify the individual instances of the type. An object type may have more than one key. The concept

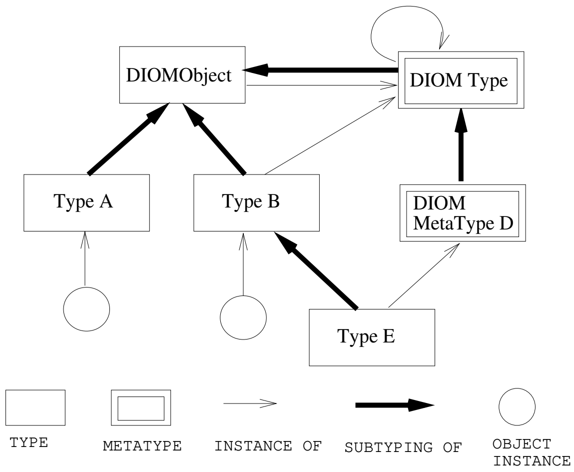


Figure 4: The subtyping relationships between DIOM types

of keys is similar to the concept of candidate keys in the relational data model, and is useful for capturing the key constraints in the underlying relational databases.

3.2.2 Attributes, Relationships and Operations

The interface definition for each DIOM type also contains instance property and instance operation declarations. The instance properties are the properties for which objects of the type carry values. Each property of an object instance is either attribute property or relationship property.

- *Attributes.* The attributes of a given type are specified as a set of *attribute signatures.* Each signature defines the name of the attribute and the type of its legal values (e.g., strings, numbers, etc.). Accessing an attribute returns a value. *Values* can be *atomic values* (values of atomic primitive types), such as integers, characters, floating-point numbers, Boolean values, or values of *structured primitive types*, such as Bit-String, Character-String, Enumeration, Date, Time, Timestamp, Interval. Unlike objects, values cannot be referred by means of references and therefore cannot be shared other than copied.

- *Relationships.* The relationships in which objects of a give type participate are specified as a set of *relationship signatures.* Each relationship signature defines the name of the *traversal function* used to refer to the related object(s), the cardinality of the relationship, and the type of the related object(s). In contrast to attributes that are defined between an object and a value, relationships are binary and are defined between objects. Since the model supports a set of built-in collection types such as sets, lists, bags, arrays, and so on, the cardinality of a relationship (e.g., one-to-one, one-to-many, and many-to-many) can be described by means of the built-in collection types.

- *Operations.* The instance operations of a given type are the operations which objects of the type support. An operation may take zero or more parameters with specified interfaces as arguments, and return a value or an object to the specified interface. Operations are specified as a set of *operation signatures.* Each signature defines the name and the type of any arguments, the type of returned values or objects, and the names of any *exceptions* (or error conditions) the operation may incur. Many of the operations declared in DIOM are actually remote operations. Executions of a remote operation automatically begins a nested transaction so that the calling operation is insulated from network failures.

Due to the space limitation, we omit the discussion on the DIOM built-in type hierarchy and the BNF syntax of the IDL in this paper. Readers who are interested in this part may refer to our technical report [20].

3.3 Advanced Concepts for Interface Definitions

3.3.1 Base Interface and Compound Interfaces

As mentioned earlier, a type of distributed interoperable objects is defined by specifying its interface in the DIOM Interface Definition Language (DIOM-IDL). The DIOM interfaces are classified into two categories: *base* interfaces and *compound* interfaces.

A **base interface** refer to the interface in which all the types involved in the definition are from a single data repository. In order to build links and abstract relationships between data from different data repositories, each data type defined in the underlying data repositories and visible to the DIOM system should be declared in terms of a base interface. For example, in the `Claim Folders` application scenario, in order to provide users and applications with both direct and uniform access to the source data from the three repositories, and in the meanwhile to create the `ClaimFolder` objects for linking the data from the three disparate source repositories (see Figure 2 and Figure 3), we may define one or more base interfaces for each of the underlying data repositories. The set of base interfaces that correspond to a given source repository can be seen as a *view* over the underlying data repository. The scope of a base interface is the corresponding data repository.

The **compound interfaces** are constructed through repetitive applications of the interface composition mechanisms to the existing (either base or compound) interfaces. A compound interface can be seen as a strongly typed contract between interfaces to provide a small but useful collection of semantically related data and operations. Each interface defines certain expected behavior and expected responsibilities of a group of objects. The scope of a compound interface is the number of data repositories to which the interface definition references. Thus, a compound interface can be an external database schema of a given data repository, or an integrated view schema over multiple data repositories. The interface composition mechanisms provided by DIOM include aggregation abstraction, generalization abstraction, specialization abstraction and the import mechanism. We will illustrate each of them through examples in the subsequent sections.

There are a number of advantages for distinguishing between base and compound interfaces. For example, by defining a base interface for each type of data in the underlying data repositories, applications of the interoperable systems may have direct access to the data sources residing in any of the underlying repositories through the DIOM interfaces. More importantly, it helps to establish a semantically clean and consistent reference framework for incremental definition of DIOM interfaces. It also facilitates the building of an adaptive and robust implementation architecture that allows independent evolution and management of distributed interoperable objects. For example, queries against compound interfaces can be easily dispatched into a package of subqueries, each against a collections of base interfaces whose interface scope is the same repository. Finally, the distinction allows a better control to the addition of new data types that are not presented in the underlying data repositories and that are unique to the interface schema, because in a distributed interoperable object system, it is important to make sure that the new types will be only those types whose instances can be unambiguously derived from instances of the source types that are defined in the underlying repositories.

In what follows, we first briefly discuss how DIOM makes use of the three well-known data abstraction mechanisms (i.e., aggregation, generalization, and specialization) introduced in object-oriented models to build useful links among disparate data sources and to resolve certain representational conflicts. Then we describe the fourth interface composition mechanism that DIOM supports, namely the *import* mechanism, and illustrate the major benefits of using the import mechanism in construction of interoperation interfaces.

3.3.2 Interface Composition Mechanism: Aggregation

The *aggregation* abstraction is a mechanism that allows to compose a new interface from a number of existing interfaces such that objects of the container interface may access the objects of component interfaces directly. As a result, the operations defined in the component interfaces can be invoked via the container's interface. The aggregation mechanism is a useful facility for implementing behavioral composition [18, 19] and ad-hoc polymorphism [5] based on coercion of operations.

Recall the `Claim Folder` example given earlier. Assume that each claim may contain more than one X-ray image but involve only one doctor's diagnosis report. In order to build links that connect the claims with the relevant images and the corresponding doctors' report, one approach is to let application developers design the `ClaimFolder` objects within their programs based on the base interface definitions given in Figure 3. However, when several applications need to use `ClaimFolder` objects, it would be highly desirable to construct a compound interface through interface aggregation abstraction mechanism for creating link objects of type `ClaimFolder`. An example IDL definition is shown in Figure 5.

```
CREATE INTERFACE ClaimFolder
(   EXTENT  ClaimFolders ): persistent
{
    AGGREGATION OF Claim, Image, DiagnosisRep;

    RELATIONSHIPS
      Claim        claims;
      Set<Image>   X-ray-pictures;
      DiagnosisRep report;
    OPERATIONS
      void *get_value(ClaimFolder);
      void *set_value(ClaimFolder);
       ...
);
```

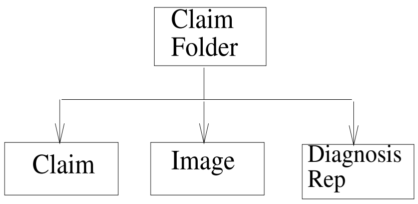


Figure 5: An example of creating a new interface using aggregation abstraction mechanism

If the type definer of `ClaimFolder` specifies that the link objects be maintained as `persistent` objects, then once the `ClaimFolder` objects are formed, they will be stored in the link database. When the `transient` option is used, it notifies the system that each subsequent query over the `ClaimFolder` objects will be carried out through the dynamic binding and loading of the objects from the underlying data repositories.

Another benefit of using interface aggregation abstraction is to minimize the impact of component schema changes over the application programs working with the existing interoperation interfaces. For example, consider the `ClaimFolder` scenario. Suppose a new image repository, called `CAT-Scan` is added into the `Claim Folder` application. After creating a base interface for `CAT-Scan`, we can simply compose a new compound interface `ClaimFolderNew` by aggregation of the original interface schema `ClaimFolder` with the base interface for the `CAT-Scan` image repository, and add the necessary links between `CAT-Scan` and `ClaimFolder`. Thus, the existing programs may continue to work with the original `ClaimFolder` interface schema. No source code modification is necessary. At the same time, the new applications can work with the new interoperation schema.

3.3.3 Interface Composition Mechanism: Generalization

The *generalization* abstraction provides a convenient facility to merge several semantically similar and yet different interfaces into a more generalized interface. The main idea is based on generalization by abstracting the common properties and operations of some existing (base or compound) interfaces. As a result, it enables objects that reside on disparate data repositories to be accessed and viewed uniformly through a generalized DIOM interface. Interface generalization mechanism also provides a helpful means to assist the automatic resolution of representational conflicts.

Consider the following example. Suppose we have three stock trade information bases available in the Internet: `NYStockInfo`, `TokyoStockInfo`, and `FfmStockInfo`. For presentation brevity, let us assume that these three repositories are all relational databases maintained separately. Figure 6 shows the relevant portion of the sample export schemas of these three stock trade data repositories. The corresponding base interface definitions in DIOM-IDL are given in Figure 7.

NYStock DB Repository

```
...
CREATE TABLE NYStockInfo
(  stock-name Char(20),
   current-trade-USprice Double,
   latest-closing-USprice, Double,
   ...
);
...
```

JapanStock DB Repository

```
...
CREATE TABLE JapanStockInfo
(  stock-item Char(30),
   current-trade-Yenprice Double,
   latest-closing-Yenprice, Double,
   ...
);
...
```

GermanStock DB Repository

```
...
CREATE TABLE FfmnStockInfo
(  stockitem-name Char(15),
   current-trade-Gmarkprice Double,
   latest-closing-Gmarkprice, Double,
   ...
);
...
```

Figure 6: The sample source specifications of `NYStocks, TokyoStocks, FfmStocks`

To query the `latest-closing price` of a stock and return the result in some desired currency (e.g., in US dollar) no matter whether the stock was traded in New York or Tokyo or Frankfurt, it is convenient if we create a new interface `StockTrade` through generalization abstraction of the base interfaces `NYStocks, TokyoStocks, FfmStocks` as shown in Figure 7. The operations `YentoUSD` and `GMtoUSD` in Figure 7 are system-supplied default conversion functions, which are generated based on the *domain-specific* knowledge stored in the DIOM metadata library. For example, we may use the currency exchange rates of the day on which the queries are issued as the default criteria for the currency exchange conversions.

```
CREATE INTERFACE NYStocks
(  EXTENT NYStocks ):  persistent
{
   ATTRIBUTES
     String stock-name;
     Double current-trade-USDprice;
     Double latest-closing-USprice;
   ...
);


CREATE INTERFACE TokyoStocks
(  EXTENT TokyoStocks ): persistent
{
   ATTRIBUTES
     String  stock-item;
     Double  current-trade-Yenprice;
     Double  latest-closing-Yenprice;
   ...
);


CREATE INTERFACE FfmnStocks
(  EXTENT FfmStocks ): persistent
{
   ATTRIBUTES
     String  stockitem-name;
     Double  current-trade-GMprice;
     Double  latest-closing-GMprice;
   ...
);
```

```
CREATE INTERFACE StockTrade
(  EXTENT StockTrade ): transient
{
   GENERALIZATION OF NYStocks, FfmStocks, TokyoStocks;

   ATTRIBUTES
     String  stockname;
     Double  current-trade-price;
     Double  Latest-closing-price;
     Enum Currency{USD,Yen,GM,...} currency-preferred;
     ...
   OPERATIONS
     double *get_value(Double current-trade-price,
                       int currency-preferred)
      {
       ...
       switch(x.currency-preferred)
         case x.currency-preferred = 0:
           if (NYStocks(x)
             return x.current-trade-USDprice;
           else
             if (TokyoStccks(x))
                return YentoUSD(x.current-trade-Yenprice);
             else
               if (FfmStocks(x))
                  return GMtoUSD(x.current-trade-GNprice);
               else < exception-handling >
          case x.currency-preferred = 1:
          ...
       };
    ...
};
```

Figure 7: Creating a new interface `StockTrade` based on generalization

DIOM provides facilities to support *domain-specific library functions* which can be used to resolve large amount of representational heterogeneity and several semantic mismatches. In addition, users may override the system-supplied library functions by defining their own conversion routines as part of their interface definitions. When a function is called, the system first check whether the relevant conversion functions exist in users' interface definitions. If not, the system-supplied library functions will be invoked.

For a particular application whose domain-specific knowledge is not yet recorded completely in the form of DIOM *domain-specific library functions*, it is then the DBA's responsibility to define the necessary conversion functions within the generalization interfaces. For example, assume an application requires information about college marks of a collection of students who come from different countries. Suppose we may gather such information from three disparate data repositories available in the Internet: `Students-1, Student-2` and `Student-3`. We want the receiving results to be represented in a chosen making scheme uniformly no matter what marking schemes were used in the source data. For example, `Students-1` uses the marking scheme of A to E, called `grade`, whereas `Students-2` uses the `point` scheme of one to ten. The `score` scheme is used by `Students-3` in the third repository in which the highest mark is 100. Suppose the receiver's preferred marking scheme is based on `score` scale, and there is no domain-specific library functions for such marking scheme conversion. Figure 8 shows an example

```
CREATE INTERFACE Int-Students
(  EXTENT Int-Students ): transient
{
   GENERALIZATION OF Students-1, Students-2, Students-3;

   ATTRIBUTES
     String  sname;
     String  mark;
     Enum Mark {grade, point, score} scheme-preferred;
      ...
   OPERATIONS
     void *get_value(char *mark, int scheme-preferred)
     {
       ...
       if (Students-1(self))
         mark(self) = gradetoscore(grade(self));
       else
         if (Students-2(self))
            mark(self) = pointtoscore(point(self));
         else
           if (Students-3(self))
              mark(self) = score(self);
           else
             < print error message >;
      };
     ...
};
```

Figure 8: An example of user-defined conversion functions in IDL

of how a DBA may specify his/her own conversion functions for converting the marking scheme used in each data repository to the receiver's preferred marking scheme.

Alternatively, users may also choose to specify and store these conversion functions as domain-specific knowledge in the DIOM metadata library. The DIOM system will provide standard header files to allow application developers to use the domain-specific library functions in a similar way as the usage of C++ library functions.

3.3.4 Interface Composition Mechanism: Specialization

The *specialization* abstraction is a useful mechanism for building a new interface in terms of some existing interfaces through type refinement. This mechanism promotes information localization such that changes in an object type or its implementations can automatically be propagated to the subtypes that are specialized versions of it. In the first phase of DIOM implementation, the specialization abstraction is only supported for construction of a new interface based on one or more base interfaces whose scope is the same data repository (see [20] for more detail).

3.3.5 Interface Composition Mechanism: Import

The *import mechanism* is designed for importing selected portions of the data from a given export schema, instead of importing everything that is available. For a data repository that manages complex objects, the import mechanism performs the automatic checking of type closure property and referential integrity of the imported types/classes. The type closure property refers to the type consistency constraint over subtype/supertype hierarchy such that whenever a type/class is imported, all the properties and operations it inherits from its supertypes have to be imported together. The referential integrity property refers to the type "completeness" rule on object reference relationships, and is used to guarantee that there is no dangling reference within the imported schema.

Using the import mechanism, a number of benefits can be obtained. First, by means of the import mechanism, users may simply specify the key information that are of interest to their intended applications. The system will automatically infer the rest of the types/classes that need to be imported in order to preserve the referential integrity and type closure property. Second, the use of import mechanism allows users to customize the source data during the importing process by excluding irrelevant portions of the source data to be imported. Similarly, users may add derived data as well. Third, when an application is interested in many types of data from a single data repository, using the import mechanism may also relieve the database administrators from the tedious job of specification of base interfaces for each of the source data types. Last but not least, the interoperation interfaces constructed through the import mechanism exhibit higher robustness and adaptiveness in the presence of component schema changes.

Consider the example shown in Figure 9, where the export source database schema `UnivDB1` consists of `Person, Employee, Department, Professor, Staff, Student, TA, RA` and `Course`. We omit the syntactical definition of the source schema in Figure 9 for presentation brevity. Suppose an application want to create a `University` interface by importing only a portion of the data from the source schema `UnivDB1`, which is related to teaching faculty or TAs. We may specify the `University` interface by using the DIOM import facility as shown in Figure 10(a). Three types are explicitly imported from `UnivDB1`. They are `Person`, `TA`, and `Professor`. For the BNF syntax of the IDL import specification, readers may refer to our technical report [20]. According to the referential integrity and type closure property, the type `Course` needs to be imported as well, because the `Course` objects are referenced by the imported types such as `Professor` and `TA` (see Figure 10(b)).

However, there is no need to import the entire type structure of `Address, Department, RA, Staff`, even though they are not explicitly excluded (hidden) from the import list in the `University` specification. The reasons are the following:

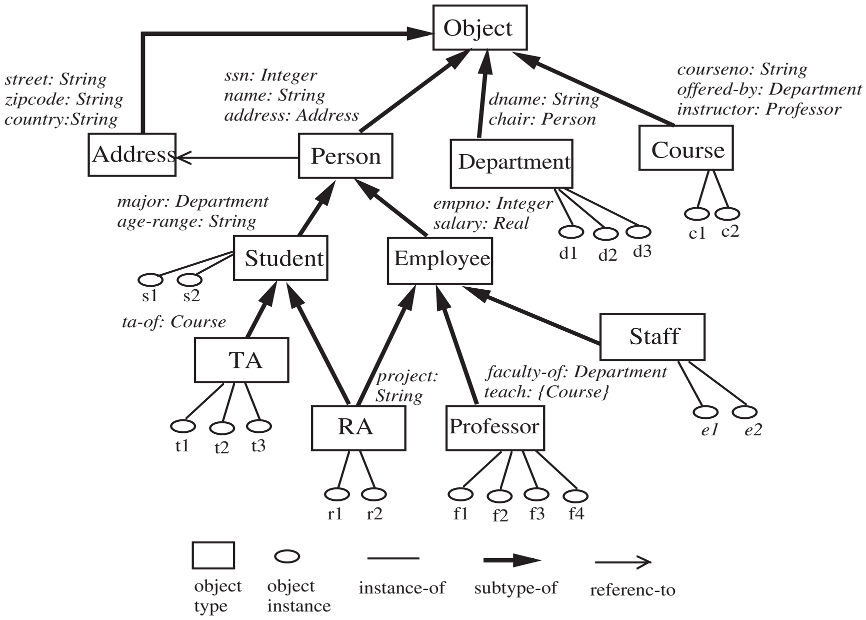


Figure 9: A sample source schema of the source database `UnivDB1`

- The reference to `Address` objects from `Person` is the only reference, and is explicitly excluded in the `University` interface specification through `HIDE` facility associated with the import mechanism. Thus there is no need for importing `Address` into the `University` schema. Similarly, there is no need to import `RA` and `Staff` either.

- In principle, if two types `T` and `S` that have indirect subtyping relationship are imported explicitly, and there are types `T1, T2, ..., Tk` in between `T` and `S` in the subtype/supertype type hierarchy, then all the properties and operations that are locally defined in `T1, ..., Tk` are imported implicitly as the properties and operations of type `S` unless being hidden explicitly. Therefore, although the type `Student` is not in the *import* list, properties of `Student` which are not explicitly hidden, such as `age-range`, are imported as the properties of `TA` objects.

- Since type `Department` is explicitly excluded from the importing list through the `HIDE TYPE` facility associated with the import mechanism, all the references to `Department` objects from the imported objects should be modified accordingly. Hence, attribute `faculty-of` in the `Professor` objects is modified from domain type `Department` to refer to, for example, the department name of type `String`.

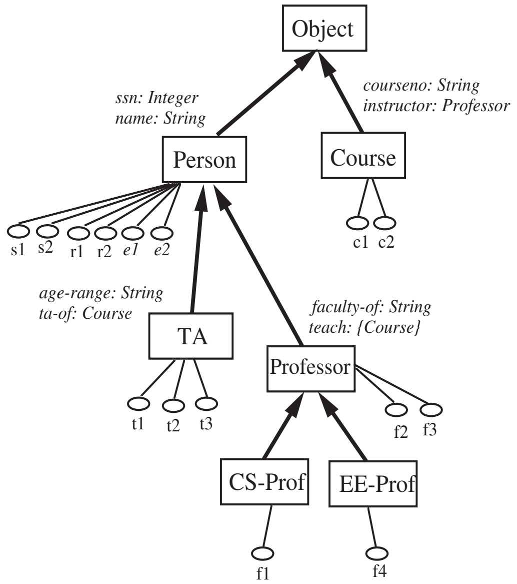
Now let us discuss how the import mechanism may help to minimize the impact of component schema changes on the applications at interoperable system level. Suppose after the `University` interface is

```
CREATE INTERFACE University
(  EXTENT University ): dynamic
{
   FROM UnivDB1
     IMPORT TYPES Person, TA, Professor;
     HIDE TYPES Deparrtment;

     TYPE Person
       HIDE address;
     TYPE Professor ISA Person
       HIDE empno, salary;
     TYPE TA ISA Person
       HIDE major, empno, salary;

     ADD TYPE CS-Prof ISA Professor
        select * from Professor
        where faculty-of = "CS";
     ADD TYPE EE-Prof ISA Professor
        select * from Professor
        where faculty-of = "EE";
     ...
};
```

(a) Creating interface `University` using the importing mechansim



(b) The resulting interface schema inferred by the facilities that support the importing mechanism

Figure 10: Creating a DIOM interface by importing a portion of the source database `UnivDB1`

created, the component database `UnivDB1` is changed by the following schema update transactions:

`T1: adding new attributes such as dept-location, total-num-emp to the Department objects;`

`T2: modifying the attribute of age-range to a larger scope;`

`T3: adding a new attribute weekly-wk-hrs to the Employee objects.`

Not surprisingly, the updates T1 and T2 require no changes to the source code of the applications which work with the `University` interface schema. Consider T1, since the `University` interface specification in Figure 10 has explicitly excluded `Department` through `HIDE TYPE` facility, any later change to the source data type `Department` will be transparent to the users of the `University` interface schema. Thus the execution of T1 has no change effect on the applications that use the interoperation interface schema `University`. For the schema update T2, since it modifies the domain of attribute `age-range` by augmenting it, thus there is no conflict or illegal data resulted from the programs developed earlier that access the `TA` objects in the `University` schema.

In the case of the component schema update T3, there is no need to rewrite the interoperation interface `University` or to modify the source code of the existing programs that work with the `University` schema. However, a new version of the `University` interface will be generated in order to provide the new applications with up-to-date information. This is simply because in the specification of the compound interface `University`, only two properties `empno, salary` are explicitly hidden from the source type `Employee`. According to the type closure property, this implies that any later extension to type `Employee` in the component repository should be visible through the `Professor` objects in the `University` interface, because of the inheritance relationship between `Employee` and `Professor` in the source repository `UnivDB1`. In order to add the `Employee`'s new attribute `weekly-wk-hrs` to the `Professor` objects, the only action the DIOM system needs to take, after notified of the component update T3, is to recompile the `University` schema, and save the recompilation result into a new version of the `University` interface, called `UniversityNew`. This new interface is seen as a specialized version of the original `University` interface. As a result, the existing programs which use the `Professor` objects via the original `University` interface may continue to work as before, while the new applications can work with the `UniversityNew` interface.

Generally speaking, by using the four DIOM interface composition mechanisms, as long as the changes in the source data repositories is information-preserving or information-augmenting, the rewriting or recompilation of the source code of the existing programs that have been developed at the interoperable system level can be either avoided or minimized.

4 Queries in DIOM databases

Even though DIOM is object-oriented, SQL-style query access to DIOM databases should be supported. The DIOM interface query language (IQL) is designed as an object-oriented extension of SQL. The design follows a number of assumptions: First, IQL by itself is not computationally complete. However, queries can invoke methods, and conversely methods supported in any host language can include queries. Second, IQL should provide declarative access to objects and thus can be optimized with known optimization techniques. In this section, we outline the DIOM query model and illustrate the IDL/IQL compilation process under the Diorama framework through an example.

Recall the `Claim Folder` application given in Section 3. Suppose the interface schema of `Claim Folder` consists of the base interfaces `Claim, Insur-Agreement, Image, ImageSpec`, `DiagnosisRep` as defined in Figure 3, and the compound interface `ClaimFolder` as specified in Figure 5. Assume that we have a query that asks for the patients who have a claim for insurance of the special "dental bridge" treatment and who have received the insurance premium within three months (90 days). The desired query answer form is the name and the insurance number of the patients and the corresponding set of X-ray images. This query requires to perform a join over three disparate data sources: Insurance database, Document repository, and the image repository. Without using DIOM interface schema, the user has to learn to use different languages in order to write three separate queries correctly, one for each repository. Furthermore, the user also needs to provide a program that implement join over the three sets of query results returned and package them into the desired output format. Whenever a new data source is involved in the query, this program has to be rewritten accordingly.

With DIOM IDL/IQL services, the user may easily express this query in DIOM-IQL against the `Claim Folder` interface schema.

```
SELECT I.patient-name, I.insur-no, Set(F.X-ray-pictures)
FROM   ClaimFolder F, Insur-Agreement I
WHERE  F.report.search("dental" && "bridge")
AND    I.insur-no == F.claims.insur#
AND    (F.claims.claim-date - I.start-date) < 90;
```

This query contains a number of path expressions (such as `F.claims.claim-date`) for traversal of the relationships defined by `ClaimFolder` link objects. It also includes the method invocation of `search()` associated with `DiagnosisRep`. DIOM query processor will automatically locate the number of data sources involved in the query each time when the query is processed. In the following we use the above example to outline the two-layered DIOM query processing: mediator level and wrapper level[2].

4.1 Query Optimization at Mediator level

Figure 11 illustrates the procedure sketch of how a DIOM query is processed under the Diorama architecture. Suppose this example query is embedded in a host program language as shown in the upper left box of Figure 11. First, the program containing the query will be passed to the IDL/IQL preprocessor. After syntax analysis, query optimization and decomposition, the IDL/IQL preprocessor generates a data access path module for each query embedded in the host program, and returns a modified source file in which all the IQL queries are replaced by DIOM system calls to the corresponding access path modules. Each subquery contained in the access path module ranges over a particular data source and will be passed by the DIOM mediator to the corresponding resource wrapper for subquery processing (see Section 4.2 for detail).

The IDL/IQL preprocessor itself consists of four basic modules as shown in Figure 11. They are syntax analyzer, optimization module, query dispatch module and query assembler module.

[2]According to I3 glossary [36], a mediator is a software component at middleware layer with services that facilitate integration and access of heterogeneous information. A wrapper is a tool that, on the one hand, convert the mediator query specification into the known repository query language, and on the other hand, translate the objects returned from the local repository to the objects that are understandable by the mediator interface.

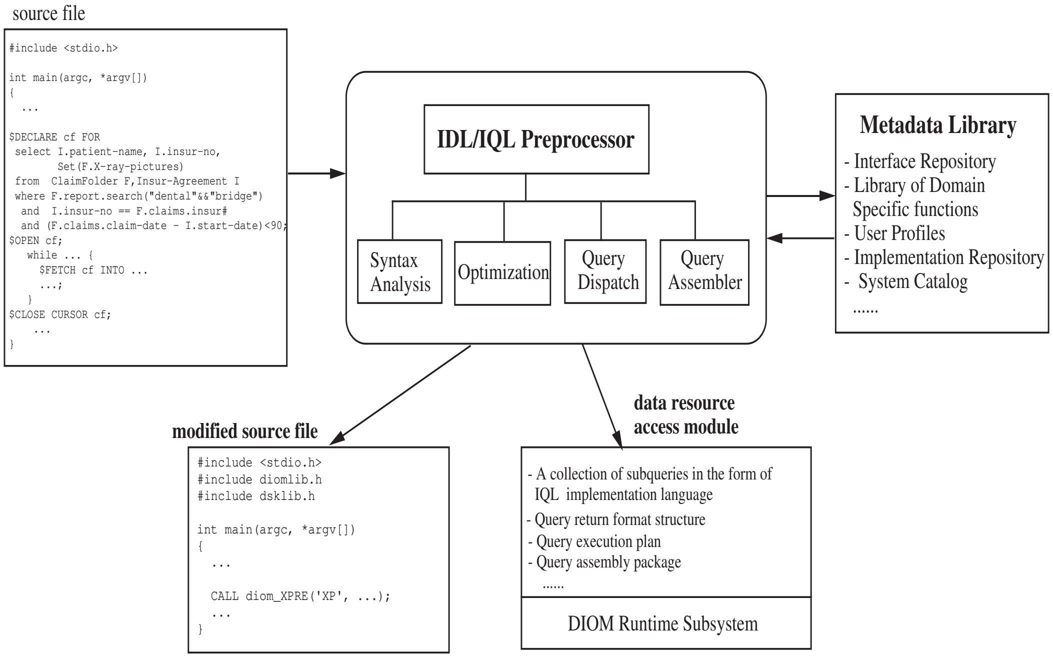


Figure 11: Accessing data from the DIOM database

- **Syntax analysis** module takes an IQL query as input, checks its syntax and produce a syntax tree expression of the query.

- **Optimization** module takes the query syntax tree as input, and performs a number of tasks, including:

  – Replace the names used in the query with internal implementation object names according to the DIOM metadata library.

  – Decompose the query into a number of smaller queries, each runnable on an individual repository. Then group those which run on the same repository into one skeleton binding on the server side, and generate a stub for each skeleton binding, including the stub in the header file of the client program.

  – Perform authorization and security checking to verify whether the users may access the collection of objects or the corresponding source data repositories, based on the records in the system catalog. For instance, in the `Claim Folder` query above, the application should have access permission to all the three source data repositories.

  – Use the optimization strategies and the statistics available to the DIOM distributed query processor to define the synchronization sequence for remote access among all the subqueries.

Prepare the plan for combining results to be returned by the number of smaller queries into the desired query answering form. And then generate an access path module in the repository implementation language. However, if the statistics required for optimization of a set of subqueries over a particular data repository is not available, which is a common case, then the optimization of the subqueries will be carried out by the individual source repository independently.

- **Query dispatch** module will be responsible for invoking the object linking and embedding services to submit each group of subqueries to the corresponding source repository for execution and collect the subquery result.

- **Query Assembler** module collects the results of the subqueries from the wrappers and assemble or filter them in terms of the user's desired representation format and then returns the final answer of the query to the user either by DIOM browser interface or through the DIOM application programs written in a host programming language (e.g., C++).

Consider the `Cliam Folder` query example given earlier. Suppose this example query is recorded in an algebraic tree structure as shown in Figure 12(a), where X denotes the projection list "`I.patient-name, I.insur-no, F.X-ray-pictures`", the predicate P1 denotes the condition "`F.report.search("dental" and "bridge")`", the predicate P2 denotes "`I.insur-no == F.claims.insurno`", and the predicate P3 denote "`(F.claims.claim-date - I.start-date) <= 90`". Recall Figure 5, the compound interface `ClaimFolder` is an aggregation of three base interfaces: `Claim, Image, DiagnosisRep`. Thus we may further decompose the compound interface `ClaimFolder` into an aggregation-based join[3] over the three base interfaces, which results in a tree structure as shown in Figure 12(b). By applying the high-level heuristics, such as performing selections before join and union, finding common subexpressions, the "optimal" algebraic expression is generated (see Figure 12(c)). It also implies that this query can be decomposed into three subqueries `subQ1, subQ2`, and `subQ3`, with the execution sequence (`subQ1` || `subQ2`) `->` `subQ3`, namely, the execution of `subQ1` and `subQ2` can be submitted to the corresponding repositories in parallel. The result of `subQ2` will be used in the selection condition of `subQ3` against the image repository.

Once a DIOM query is compiled through DIOM query pre-processor, it will be decomposed into a group of subqueries, each corresponding to one data source. These subqueries will be then passed to the corresponding wrappers for subquery processing. The DIOM mediator will collect the subquery results returned from the wrappers, and perform some filtering and assembly jobs to package the final

[3] Aggregation-based join refers to an object join [18] which follows the aggregation path specified by object identities.

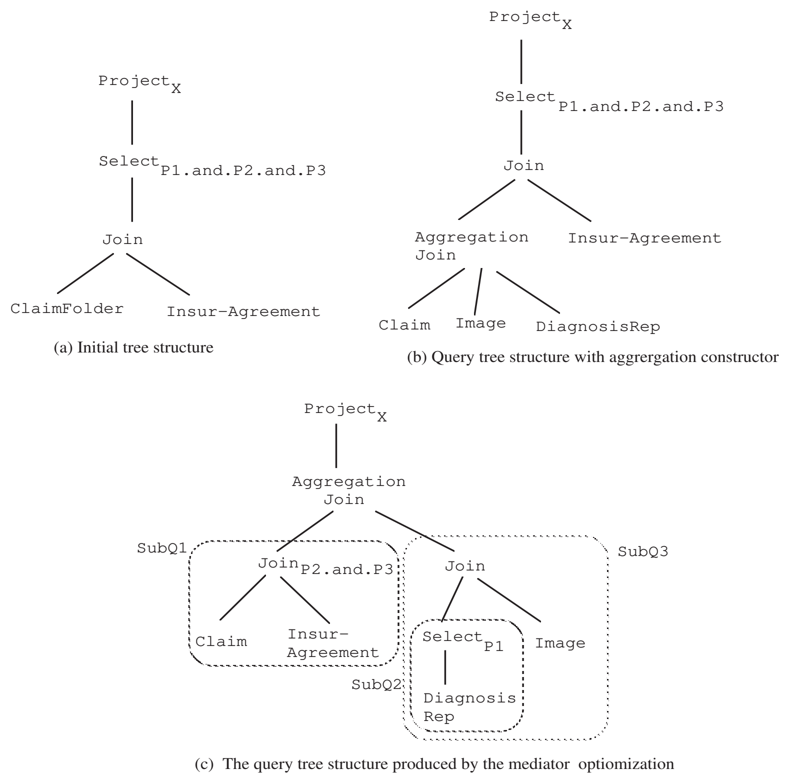


Figure 12: Query optimization at mediator level

answer of the query according to the user's desired format before delivery of the query answer to the user. Figure 13 shows a sketch of the coordination between a DIOM mediator and its associated wrappers.

We have illustrated by example the flavor of DIOM query processing at mediator level and will discuss the processing of decomposed subqueries at wrapper level in the next section. We are currently working on the formal development of the DIOM heuristic-based query optimization, which incorporates the four compound interface composition constructors, and the set of equivalence-preserving rewriting rules. We will report the result in a forthcoming technical report.

4.2 Processing of Decomposed Queries at Wrapper Level

After optimization at the mediator level, the subqueries in IQL expressions will be passed to the corresponding wrappers (see the subQ1, subQ2, and subQ3 in Figure 13). The wrapper will first convert the IQL query into a query expression that is understandable to the data source which the wrapper serves to. When the repository is a RDBMS, say Oracle, the wrapper will map an IQL expression into Oracle/SQL statements. Also some further query optimization will be performed locally by the Oracle

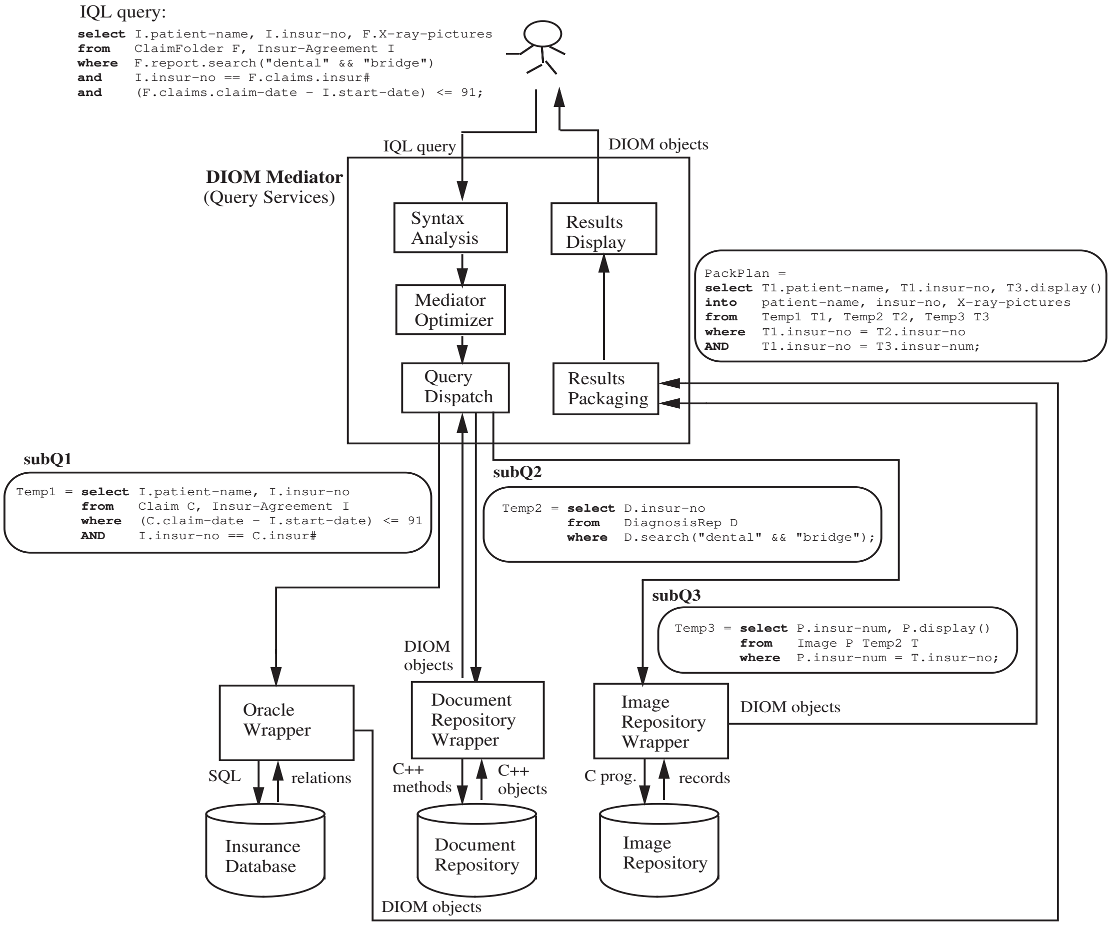


Figure 13: The DIOM query processing strategies

query processor. Furthermore, if there are more than one Oracle databases that should respond to the query, the wrapper will be in charge of further decomposing this query into a number of smaller queries, each running over a single database, if no distribution is supported. It is also the wrapper's responsibility to collect the results and package them in the form of the DIOM objects before sending the results of the subqueries back to the mediator (see Finger 13). If the data source is an OODBMS, say ObjectStore, the wrapper will need to map each IQL expression into a query that is bounded to a class hierarchy. When the data source is stored and managed solely by a file server, say in form of Unix files, then the wrapper will map the IQL expression into, for example, a C module that scans the source data and returns the matching records.

We are currently investigating the generic wrapper architecture and the generic conversion functions for transforming DIOM IQL queries to different types of data repositories. Our first implementation is done through a Netscape based DIOM browser, which runs over two types of wrappers: an Oracle wrapper for structured data processing and a Harvest wrapper for unstructured or semi-structured data processing (e.g., WWW pages, NetNews articles, bibtex files).

5 Related Work

Over the last decade, many published results on database interoperability have primarily focused on resolving schematic and semantic incompatibilities arising from autonomy of the underlying databases. They use two integration strategies: *tight-coupling* and *loose-coupling* [30]. Tight-coupling insists on total schema integration. Since the creation and maintenance of such an integrated schema has been recognized as a fundamental roadblock towards system scalability and evolution, tight-coupling is generally considered a small-system strategy. In contrast, loose-coupling goes to another extreme by favoring zero schema integration, for example, by promoting integration and interoperability via a multidatabase query language [16, 17]. It does not require the existence of an integrated schema, leaving many responsibilities, such as resolving semantic mismatch, dealing with multiple representations of data, and coping with dynamic system evolution, to the users.

Recent research in database interoperability has started to pay more attention to interoperation architectures that may support both tight-coupling and loose-coupling (cf. [11, 9, 24, 27, 34]). A few prototype systems are under development. Examples include the Garlic project at IBM Almaden Research Center [6], which targets at developing a system and tools for the management of large quantities of heterogeneous multimedia information; the TSIMMIS project at Stanford [11, 23], which proposes an object exchange model (OEM) and a declarative language (LOREL) to intelligent information mediation; and the Context Interchange approach [26, 12].

The DIOM development is motivated mainly by the mediator architecture [33, 34, 35]. For example, the idea of using a repository/resource wrapper to bridge between the interoperable database system and the individual component repositories is to some extent encouraged by the mediator approach. The intelligent integration of information through query compilation is also supported in the DIOM framework. Comparing with the Garlic project, both DIOM and Garlic extend ODMG model to some extent. Garlic focuses more on extensions that may facilitate multimedia application design and development, while DIOM's extension to the ODMG is targeted more at the flexible and adaptive composition of interoperation interfaces, such as the import mechanism and the aggregation abstraction of interfaces. DIOM has also adopted the same approach as TSIMMIS to mediation and to allow different mediators to interoperate in the similar way as the interoperation among multiple data repositories. However, TSIMMIS uses a very simple object model which does not provide explicit support for specialization, generalization, and aggregation abstraction as the object-oriented model does. In contrast, DIOM supports a rich set of interface composition constructors and allows new interfaces to be constructed from existing ones rather than from scratch. We believe that each of the two approaches has its pros and cons to different application domains. Comparing with the Context Interchange proposal, both DIOM approach and Context Interchange approach support explicit representation of semantics underlying disparate data sources and heterogeneous receivers. However, in contrast to the shared ontology model [35] used in the Context Interchange proposal, DIOM provides an interface definition language and a semantically rich and yet clean framework that allows independent evolution and management of distributed interoperable objects. DIOM also supports four interface composition mechanisms for incremental design and construction of the interoperation interfaces, and provides strategies for distributed query decomposition and packaging, for generation of optimal data access plan, and for dynamic object linking and embedding.

DIOM approach to interoperation allows flexible integration and and access of heterogeneous information gathered by other mediators or information brokers, such as the TSIMMIS system [11] and the HARVEST resource discovery toolkit [2], as soon as a wrapper/adaptor is provided, for example between Diorama and TSIMMIS or between Diorama and HARVEST (see Figure 14).

In addition, a number of proposals have competed as the basic enabling technologies for implementing interoperable objects in distributed and dynamic object computing environments. Examples include Microsoft's Object Linking and Embedding (OLE), IBM's System Object Model (SOM) and its distributed version (DSOM), OMG's Common Object Request Broker Architecture (CORBA), and CI labs OpenDoc. Many of these are available as deployed software packages. The emergence of these technologies also demonstrates that the continued evolution of object-oriented programming and object-oriented database management systems are converging into language-independent and distributed object computing. Although these proposals are clearly practical and important, they focus primarily on the software interface problem, not the USECA properties in large scale interoperable database systems. DIOM can be seen as a glue that spans and integrates these interface models at a higher level. For example, unlike in CORBA, ODMG, and DSOM where only specialization/generalization are supported in interface construction, DIOM also supports import and aggregation mechanism for interface abstraction, which not only provide more flexibility and convenience in interface composition but also increase the robustness, the scalability, and the adaptiveness of compound interfaces in the presence of component schema changes and system evolution.

Another related field is the area of Distributed Object Management [22]. A representative effort is the

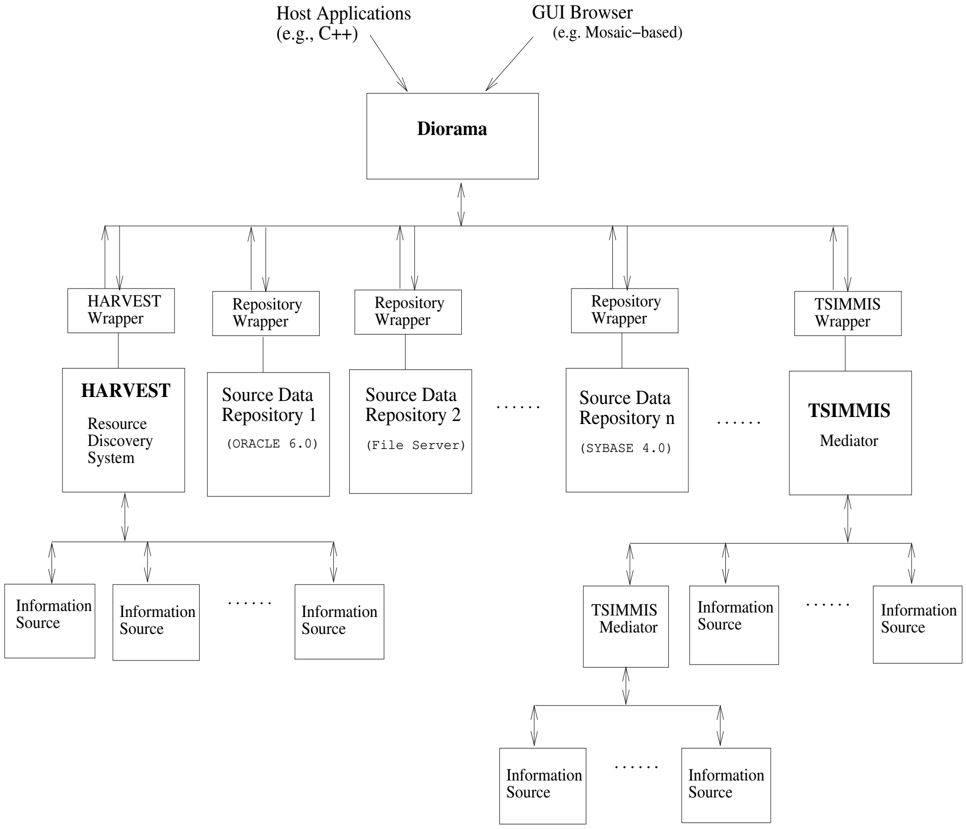


Figure 14: The Diorama system in a mediator-based information integration architecture

ongoing research at GTE Labs [21] to divide object models into components and then map them into each other. Another example is the BLOOM model [7], which is a semantic data model that extends the relational model with data abstractions such as generalization and aggregation. Compared to these more formal models, DIOM is designed to support the USECA properties and it combines abstractions with a practical implementation path.

6 Conclusion

We have introduced the Distributed Interoperable Object Model (DIOM) and described a concrete instance of DIOM in the Diorama project. Diorama and DIOM have been designed to address the research challenges in large scale interoperable database systems: Uniform access, Scalability of network, Evolution of system components, Composability of component databases, and preservation of Autonomy (USECA properties).

Our main contributions in this paper are the following. First, we propose the use of base interfaces to model data from individual repositories and compound interfaces to model the desired links among remote data sources. Our design not only presents a useful method for construction of interoperable databases from disparate data sources, but also provides a semantically clean framework for allowing independent evolution and management of distributed interoperable objects. Second, the interface composition mechanisms provided in DIOM, such as *import* and *hide*, *aggregation, generalization* and *specialization*, allow incremental design and construction of new interfaces in terms of the existing ones. For instance, by combining the import facility with the aggregation and generalization abstraction mechanisms, one can easily build a new interface by adding additional properties and operations to the existing interfaces. The composability as such is very important for the organizations which perform multiple interoperable projects with overlapping sets of data. Thirdly, the design of both the DIOM interfaces and the Diorama architecture have given the full respect to the autonomy of component systems. There is no need for component systems to request an agreement from the interoperable system in the presence of component evolution. Furthermore, all the information-preserving and information-augmenting changes in the component systems will have no impact on the use of the interoperable interfaces. To our knowledge, the DIOM interface composition mechanisms are original with respect to the support for USECA properties.

In addition, conflict resolution is considered secondary to the explicit representation of disparate data semantics in DIOM. Thus, the DIOM approach does not require any commitment to a canonical, pre-defined integration representation of component databases. Conflicts do not have to be resolved once for all in a static integrated schema. Instead, we encourage decoupling the resolution of semantic heterogeneity from representational heterogeneity. Much of conflict resolutions are deferred to the query submission time. More importantly, the conflict resolution is not hard-wired in the interoperation interface schema. Users may define their preferred query answering formats or update the existing conversion functions whenever necessary. Thus, automatic recognition and resolution of semantic conflicts can be supported. We have also informally described the DIOM query processing strategies at both mediator level and wrapper level. We believe that the framework developed in this paper presents an interesting step towards supporting the USECA properties in large scale interoperable database systems.

Finally, we would like to state that the DIOM approach presented in this paper proposes an adaptive framework and a collection of techniques for incremental design and construction of interoperation interfaces, rather than a new object model. This framework is targeted towards large scale interoperable database systems operating in today's dynamic, growing, and evolving environment, which require the support of USECA properties.

References

[1] M. Betz. Interoperable objects: laying the foundation for distributed object computing. *Dr. Dobb's Journal: Software Tools for Professional Programmer*, (220), October 1994.

[2] C. Bowman, P. Danzig, D. Hardy, U. Manber, and M. F. Schwartz. The harvest information discovery and access system. In *Proceedings of the Second International World Wide Web Conference*, pages 763–771, Chicago, Illinois, October 1994.

[3] M. Bright, A. Hurson, and S. H. Pakzad. A taxonomy and current issues in multidatabase systems. *IEEE Computer Magazine*, March 1992.

[4] L. Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, pages 51–67. Springer Verlag, 1984.

[5] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471, December 1985.

[6] M. Carey, L. Haas, and P. S. et al. Towards heterogeneous multimedia information systems: the garlic approach. In *Technical Report, IBM Almaden Research Center*, 1994.

[7] M. Castellanos, F. Saltor, and M. Garcia-Solaco. A canonical model for interoperability among object-oriented and relational databases. In *Proceedings of the 1992 International Workshop on Distributed Object Management*, Edmonton, Canada, August 1992.

[8] R. Cattell and et al. *The Object Database Standard: ODMG-93 (Release 1.1).* Morgan Kaufmann, 1994.

[9] D. D.Fang, S.Ghandeharizadeh and A. Si. The design, implementation, and evaluation of an object-based sharing mechanism for federated database systems. In *Proceedings of International Conference on Data Engineering*, Vienna Austria, 1993.

[10] A. K. Elmagarmid and C. Pu. *Special Issue on Heterogeneous Databases.* ACM Computing Surveys, Vol. 22, No. 3, September 1990.

[11] H. Garcia-Molina and et al. The tsimmis approach to mediation: data models and languages (extended abstract). In *Technical Report, Stanford University*, 1994.

[12] C. Goh, S. Madnick, and M. Siegel. Context interchange: overcoming the challenges of large-scale interoperable database systems in a dynamic environment. In *Proceedings of International Conference on Information and Knowledge Management*, pages 337–346, 1994.

[13] W. Kim. Observations on the odmg-93 proposal. *ACM SIGMOD RECORD on Management of Data*, 23(1), March 1994.

[14] W. Kim and J. Seo. Classifying schematic and data heterogeneity in multidatabase systems. *IEEE Computer Magazine*, 24(2):12–18, 1991.

[15] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The objectstore database system. *Communications of the ACM*, 34(10), 1991.

[16] T. Landers, W. Litwin, and W. Kent. Language features for interoperability of databases with schematic discrepancies. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, pages 40–49, 1991.

[17] W. Litwin and A. Abdellatif. An overview of the multidatabase manipulation language mdsl. In *Proceedings of the IEEE 75, 5*, pages 621–632, 1987.

[18] L. Liu. A recursive object algebra based on aggregation abstraction for complex objects. *Journal of Data and Knowledge Engineering*, 11(1):21–60, 1993.

[19] L. Liu and R. Meersman. Activity model: a declarative approach for capturing communication behavior in object-oriented databases. In *Proceeding of the 18th International Conference on Very Large Databases*, Vancouver, Canada, 1992. Morgan Kauffman.

[20] L. Liu and C. Pu. The distributed interoperable object model and its application to large scale interoperable database systems. Technical report, University of Alberta, June 1995.

[21] F. Manola and S. Heiler. An approach to interoperable object models. In *Proceedings of the 1992 International Workshop on Distributed Object Management*, Edmonton, Canada, August 1992.

[22] M. Ozsu, U. Dayal, and P. Valduriez, editors. *Distributed Object Management*, Edmonton, Canada, August 1992. Morgan Kaufmann.

[23] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of International Conference on Data Engineering*, Taiwan, March 1995.

[24] C. Pu. Superdatabases for composition of heterogeneous databases. In *Chapter in IEEE Computer Society Tutorial Multidatabase Systems: An Advanced Solutions for Global Information Sharing*, ed. A.R. Hurson, M.W. Bright, and S.Pakzad. An extended abstract was published in the Proceedings of International Conference on Data Engineering (Los Angeles 1988).

[25] S. Ram. *Special Issue on Heterogeneous Distributed Database Systems.* IEEE Computer Magazine, Vol. 24, No. 12, December 1991.

[26] E. Sciore, M. Siegel, and A. Rosenthal. Using semantic values to facilitate interoperability among heterogeneous information systems. *ACM Trans. Database Syst.*, Vol. 19, No. 2 June 1994.

[27] M. Shan. Pegasus architecture and design principles. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, 1993.

[28] A. Sheth. *Special Issue in Multidatabase Systems.* ACM SIGMOD Record, Vol.20, No. 4, December 1991.

[29] A. Sheth and V. Kashyap. So far (schematically) yet so near (semantically). In *Proceeding of the IFIP WG2.6 Database Semantics*, pages 283–312, Victoria, Australia, 1992. North Holland.

[30] A. Sheth and J. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Trans. Database Syst.*, Vol. 22, No.3 1990.

[31] M. Siegel and S. Madnick. A metadata approach to solving semantic conflicts. In *International Conference on Very Large Data Bases*, pages 133–145, 1991.

[32] V. Ventrone and S. Heiler. Semantic heterogeneity as a result of domain evolution. *ACM SIGMOD RECORD on Management of Data*, 20(4), 1991.

[33] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer Magazine*, March 1992.

[34] G. Wiederhold. Intelligent integration of information. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, 1993.

[35] G. Wiederhold. Interoperation, mediation, and ontologies. In *Proc. Int. Symp. on Fifth Generation Comp Systems*, pages 33–48, ICOT, Tokyo, Japan, 1994.

[36] G. Wiederhold. I3 glossary. *Draft 7*, March 16 1995.