An Adaptive Object-oriented Approach to Integration and Access of Heterogeneous Information Sources

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 Uniform access to heterogeneous information sources, Scalability to the growing number of information sources, Evolution and Composability of software and information sources, and Autonomy of participants, both information consumers and information producers. We refer to these set of properties as the USECA properties [30]. To address the research issues presented by such systems in a systematic manner, we introduce the Distributed Interoperable Object Model (DIOM). DIOM promotes an adaptive approach to interoperation via intelligent mediation [46, 47], aimed at enhancing the robustness and scalability of the services provided for integrating and accessing heterogeneous information sources. DIOM's main features include (1) the recursive construction and organization of information access through a network of application-specific mediators, (2) the explicit use of interface composition meta operations (such as specialization, generalization, aggregation, import and hide) to support the incremental design and construction of consumer's domain query model, (3) the deferment of semantic heterogeneity resolution to the query result assembly time instead of before or at the time of query formulation, and (4) the systematic development of the query mediation framework and the procedure of each query processing step from query routing, query decomposition, parallel access planning, query translation to query result assembly. To make DIOM concrete, we outline the DIOM-based information mediation architecture, which includes important auxiliary services such as domain-specific metadata library and catalog functions, object linking databases, and associated query services. Several practical examples and application scenarios illustrate the flavor of DIOM query mediation framework and the usefulness of DIOM in multi-database query processing.

Index Terms: *Interoperability, Distributed data Management, Object-oriented Data Model, Intelligent Integration of Heterogeneous Information Sources, Query Mediation.*

1 Introduction

Computer-based information systems, connected to world-wide high-speed networks, provide increasingly rapid access to a diversity of autonomous and remote information sources. The availability of various Internet information browsing tools further promote the information sharing across departmental, organizational, and national boundaries. A critical challenge presented by this technology advancement is the intelligent interoperation of data, of heterogeneous semantics and representations, among autonomous information sources. Of the most desirable services is how to locate relevant information sources to answer a consumer's query request and how to combine and organize the data gathered from multiple information sources or through different information browsers or GUI tools.

Because of the various need of legacy applications and the growing number of autonomous information sources, heterogeneity in the semantics and representation of data is natural and will not disappear by any types of global data integration enforcement. Therefore, a large-scale interoperable database system requires the capability to interconnect information consumers and information producers in a more flexible way, and to provide transparent and customizable information access across multiple heterogeneous information sources (including databases, knowledge bases, flat files, or programs). We have captured the requirements of these systems as the USECA properties [30]: *Uniform access* to heterogeneous information sources, *Scalability* to the growing number of information sources, *Evolution* and *Composability* of software and information sources, and *Autonomy* of participants, both information consumers and information producers.

Our main contribution in this paper is the development of a *Distributed Interoperable Objects Model* (DIOM) to promote the USECA properties in large-scale interoperable database systems. The DIOM proposal presents an extension to the ODMG-93 object model [10]. The main idea is to incorporate system-wide scalability and evolution support as well as component composability into an intelligent interoperation framework. To deal with both the heterogeneity issues and the exponential growth of the available information, scalability and extensibility become very critical. We promote two independent but complementary strategies for achieving better scalability and higher extensibility in advanced cooperative database systems.

- An incremental approach to construction and organization of information access through a network of domain-specific application mediators; and support the dynamic linking of mediators to heterogeneous information sources via repository-specific wrappers (see Figure 1).
- The support for facilities to allow consumers to specify their queries in terms of how they would like their query results be received and represented, rather than relying on a global integrated view of all the participating information sources.

The first strategy guarantees a seamless incorporation of new information sources into the cooperative database system. The second strategy allows the distributed query services to be developed as source-independent middleware services which establish the interconnection between consumers and a variety

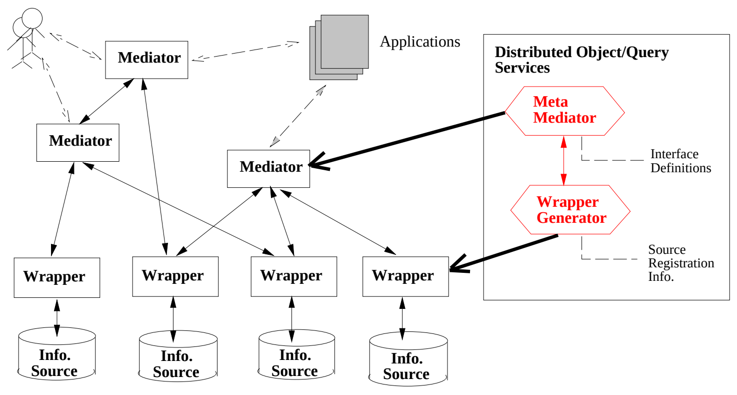


Figure 1: The cooperation architecture of network of mediators

of information producers' sources at the query processing time. As a result, the addition of any new sources into the system only requires each new source to have a wrapper installed. The main issues to be addressed in this paper include:

- the recursive construction and organization of information access through a network of application-specific mediators;
- the important component services of a distributed query processing manager, such as query routing, query decomposition, parallel access planning, query translation, query result packaging and assembly;
- the explicit use of interface composition meta operations (such as specialization, generalization, aggregation, import and hide) and the application of intelligent compilation to support the incremental design and construction of consumer's domain query model;
- the deferment of heterogeneity resolution to the query result assembly stage rather than before or at the time of query formulation.

As a result, new applications may define their interoperation interfaces by refinement of existing interfaces through interface aggregation, interface generalization, or import and hide meta operations. New information sources can be seamlessly incorporated into the existing consumer's query model.

Thanks for the fact that distributed query processing and optimization [12, 25], and extended transaction management [14] have so far been explored quite extensively in the context of classic distributed databases [37], federated databases [15, 42], and distributed object management [34, 38], in this paper we focus only on how the USECA properties should be understood and strengthened in the DIOM environment and what are the important adaptive mechanisms in the design and development of the DIOM interface description language (IDL) and interface query language (IQL), so that a multidatabase system can scale well and be robust in the presence of (i) the growing number of information sources, (ii) the evolution of the information producers' source schemas, and (iii) the evolution of the information consumers' query model. We use a medical insurance application scenario as the running example to illustrate the flavor of the DIOM query services and to highlight the role of the DIOM interface composition meta operations in the reformulation and simplification of a consumer's query over multiple heterogeneous information sources.

We would like to state that the DIOM approach presented in this paper proposes an adaptive framework, rather than a new data model, for implementing intelligent information mediation [46, 47]. This framework is targeted towards large scale interoperable database systems operating in a dynamic and evolving environment, which require transparent, scalable,and customizable access of multiple heterogeneous information sources while maintaining the USECA properties.

The rest of the paper proceeds as follows: Section 2 overviews the system architecture of the DIOM system. In order to demonstrate the practical applicability of the DIOM approach, we also present the Diorama implementation framework. 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 meta operations are described in Section 3 through a number of illustrative examples. Section 4 outlines several query mediation services implemented under the Diorama architecture. We compare our approach with related work in Section 6 and conclude the paper in Section 7.

2 The DIOM Information Mediation Architecture

2.1 An Overview

In DIOM environments, the information sources may differ by their organization, such as traditional databases, knowledge bases, and flat files; by their content, such as HTML hypertext, relational tables, and objects of complex class structures; and by the many browsers and graphical user interfaces (GUIs) as well as query languages used for information access. Figure 2 shows an example network of information mediators collaborating through the DIOM mediator network architecture. This network includes simple wrapper-based mediators such as a wrapper to BookStore data repository which only provides information about and access to the BookStore repository. This wrapper-based mediator is in turn used to construct a BookSale mediator. The BookSale mediator is again used to build both a document inquiry mediator and a travel plan mediator. This recursive construction and organization of information access have the following important features. First, the individual mediators can be independently built and maintained. Each specialized mediator represents a customized personal view of an information consumer over the large amount of information accessible from the growing number of information sources. Second, these mediators can be generated automatically or semi-automatically by using the DIOM IDL/IQL interface specification language and the associated incremental compilation techniques. These features also make the DIOM architecture to scale well to the large and growing num-

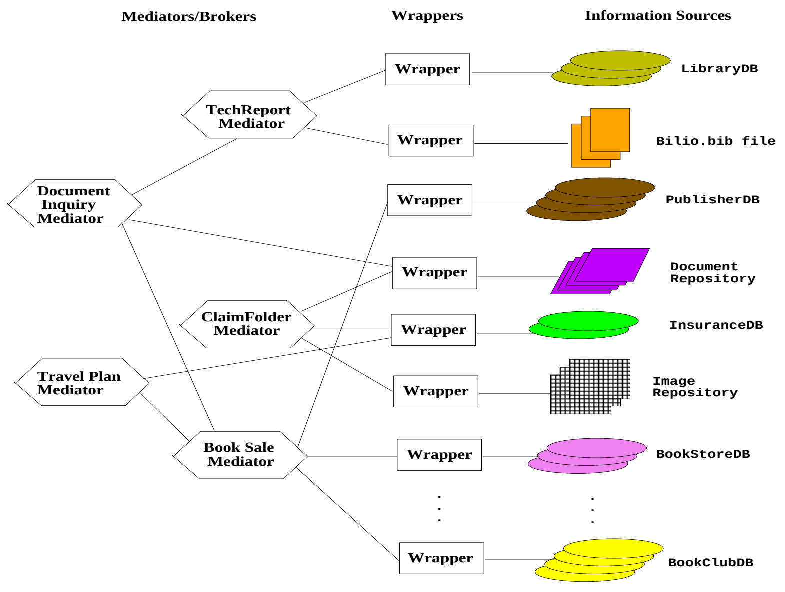


Figure 2: An example network of collaborating information mediators

ber of information sources and to the varying information customization needs from diverse information consumers.

To build a network of specialized information mediators, we need an architecture for a meta mediator that can be instantiated to build multiple specialized mediators (see Figure 1 and Figure 3). The ultimate goal of developing such a mediator architecture is to provide a uniform interface description language, a suite of interface composition meta operations, and the associated query processing services that are customizable and can be utilized by a number of application-specific mediators to facilitate the access to multiple heterogeneous information sources.

A typical meta mediator has a two-tier architecture offering services at both the mediator level and the wrapper level. The mediator object server (Figure 3) is responsible for (i) the coordination and information exchange between information consumer's domain usage model and the relevant information source models, (ii) the distributed metadata library of mediated metadata and the correspondence to the metadata of information source models, and (iii) the maintenance of the metadata catalog in the presence of changes or upon the arrival of new information sources.

Mediators in DIOM are application-specific. Each mediator consists of a consumer's domain model and many information producer's source models and are described in terms of the DIOM interface definition language (DIOM IDL) [30]. The consumer's domain model specifies the querying interests of the consumer and the preferred query result representation. The producer's source models describe the information sources in terms of DIOM internal object representation generated by the DIOM interface manager. The consumer's domain model and the information producer's source models constitute the general knowledge of a mediator and are used to determine how a consumer's information request is

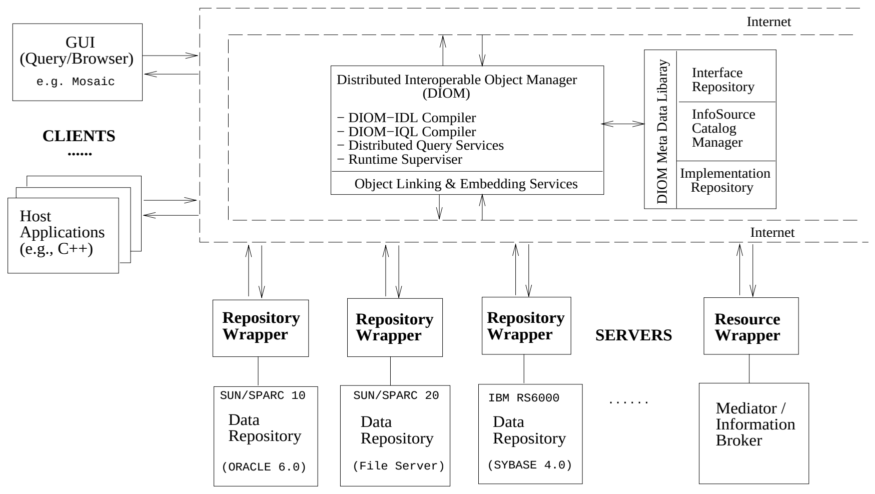


Figure 3: The DIOM meta mediator architecture

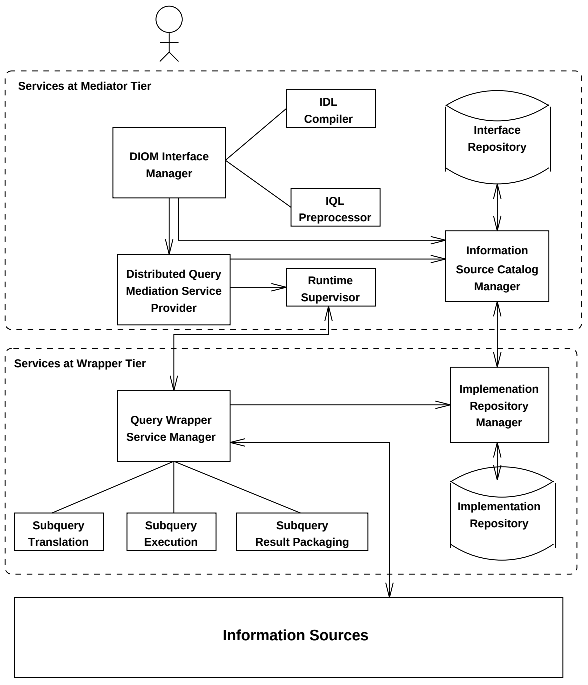
processed. The main task of the mediator sub-system is to utilize the metadata provided by both information consumers and information producers for efficient processing of distributed queries.

Wrappers are software modules, each serving for one component data repository. In order to make an existing information source available to the network of mediators, building a wrapper around the existing system is needed to turn the system into a DIOM local agent, which is responsible for accessing that information source and obtaining requisite data for answering the query. The main task of a wrapper is to control and facilitate external access to the information repositories by using the local metadata maintained in the implementation repository and the wrapper functions. The main services provided by a wrapper include translating a subquery in consumer's query expression into an information producer's query language expression, submitting the translated query to the target information source, and packaging the subquery result into a mediator object. By building a wrapper around the existing system we turn the local system into a cooperative database agent, which is responsible for accessing that information source and obtaining requisite data from the local information source. Note that only one such wrapper would need to be built for any given type of information source (e.g., relational, object-oriented, or flat ASCII or HTML files).

Information sources at the bottom of the diagram are either *well structured* (e.g., RDBMS, OODBMS), or *semi-structured* (e.g., HTML files, text based records), or *unstructured* (e.g., technical reports). Each information source is treated as an autonomous unit and has its own customer set. Component information sources may make changes without requiring consent from mediators. However, an information source must notify the DIOM object server any change made to its export schema (e.g., logical structure, naming, and semantic constraints).

2.2 The Diorama Implementation Architecture

Figure 4 presents a sketch of the Diorama architecture, which implements the I^3 intelligent integration of information reference framework [20] using the DIOM approach.



Services at Mediator Tier
IDL Compiler
DIOM Interface Manager
Interface Repository
IQL Preprocessor
Information Source Catalog Manager
Distributed Query Mediation Service Provider
Runtime Supervisor
Services at Wrapper Tier
Implemenation Repository Manager
Query Wrapper Service Manager
Implementation Repository
Subquery Translation
Subquery Execution
Subquery Result Packaging
Information Sources

Figure 4: The Diorama System Architecture

The following provides a brief description of the main components of the Diorama implementation architecture:

DIOM Interface Manager: interacts with the system user by presenting a GUI interface and underlying API to allow users to perform DIOM functions, compiles IDL statements using the IDL compiler and preprocesses IQL statements using the IQL preprocessor.

Distributed Query Mediation Service Provider: provides the distributed query processing services including source selection, query decomposition, parallel access plan generation, and result assembly.

Runtime Supervisor: executes subqueries by communicating with wrappers.

Information Source Catalog Manager: responsible for the management of both information source repository metadata and interface repository metadata, and communicates with the local implementation repository managers to cooperate in the maintenance of wrapper metadata.

Query Wrapper Service Manager: receives query requests from the runtime supervisor, uses data from the implementation repository, utilizes the wrapper query processing modules, and communicates with the local information sources to return a result.

Implementation Repository Manager: manages the implementation repository metadata by coordinating with the Information Source Catalog Manager. The implementation repository maintains the correspondence between the source information and their DIOM internal object representation.

In our view, the decision on how to distribute the set of distributed query services between mediator layer and wrapper layer is not only an implementation decision but also depends on the strategic policy a system takes. For instance, Diorama implementation distributes the subquery translations from the mediator layer to the wrapper layer, which helps to prevent query processing bottleneck at the DIOM mediator server level. Our experience with Diorama also demonstrates that the approach of building wrappers to coordinate between a mediator and its underlying sources can greatly simplify the implementation of individual mediators, making interoperation among networks of mediators more easily scalable to the ever growing number of information sources.

3 The Distributed Interoperable Object Model

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 [6]. The abstractions of distributed interoperable objects are captured in the Distributed Interoperable Object Model (DIOM).

Rather than inventing yet another object-oriented model, DIOM extends the ODMG ODL [10] in a significant way using reflection concepts [23, 33]. DIOM introduces base interface and compound interface concepts to separate the interface descriptions for information producers' source models from the interface descriptions for information consumers' domain query models (personal views). Such a separation serves dual purposes: (1)It enables the system to automate the specification of base interfaces. (2)It allows information consumers focusing on their domain-specific business interests and building dynamic and personal views, rather than monolithic and integrated snapshots, over the growing number of information sources. We call the ODMG ODL standard interfaces the *base* interfaces, and define *meta operations*, such as *aggregation, generalization, specialization*, and *import/hide*, to manipulate base interface and construct compound interfaces. We believe that this conceptual distinction supports and encourages incremental interface construction, which is essential in an evolving distributed application. In fact, we believe that most of distributed applications will be evolving throughout most of their lifetime and much of the software maintenance problems reside in effectively and efficiently supporting such evolution.

3.1 A Quick Look at the Model

As in the ODMG-93 standard, DIOM's basic entity is *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 (see Section 3.3 for the generation of OIDs).

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.

In the rest of the paper, we assume that the readers are familiar with the ODMG-93 ODL and OQL standard [10]. Thus we only discuss the distinct features of DIOM design and development, which are currently not provided in the ODMG object model according to the ODMG-93 proposal [10].

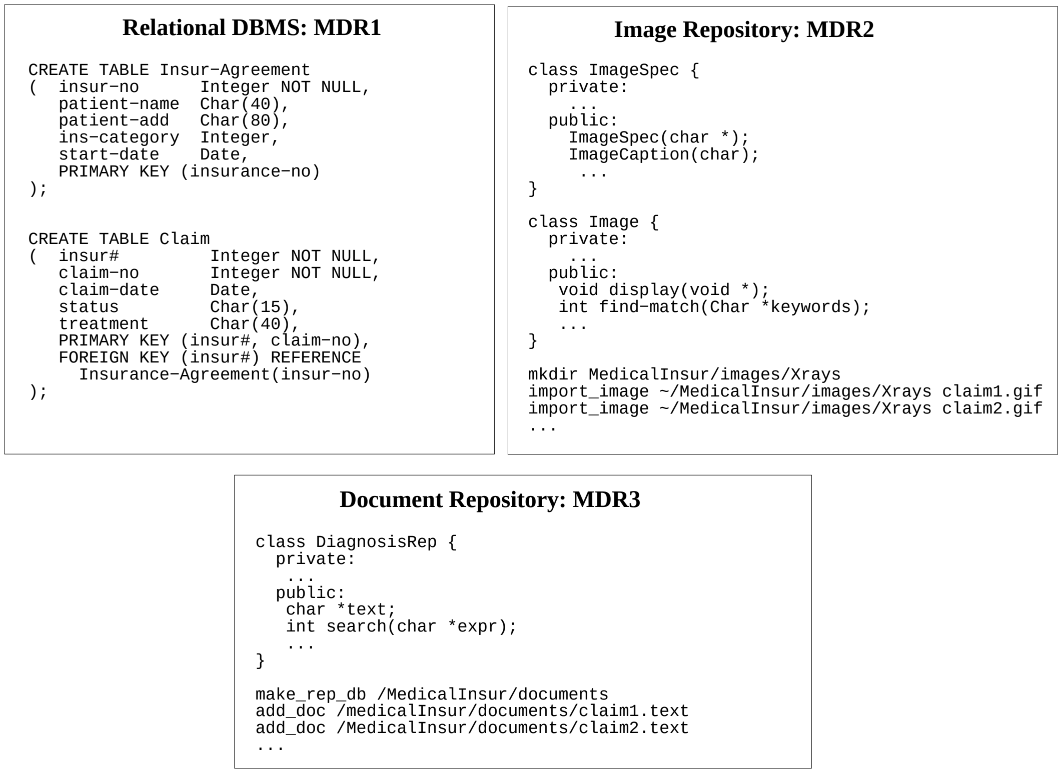
Consider a medical insurance application. Suppose that an insurance agent needs to construct `ClaimFolder` objects by combining data from the following three disparate data repositories:

- a patient's insurance agreement and claims for special medical treatments stored in a relational data base of the relevant insurance company,
- a collection of X-ray images associated with each claim of a patient, stored in an image-specific file system of a radiological lab, and
- a doctor's diagnosis report stored in a C^{++} based document repository at doctor's office.

Figure 5 shows the contents of these three data repositories:

Assume that each claim may contain more than one X-ray image but involve only one doctor's diagnosis report. The information consumer may specify the entity objects to be imported from the given data sources using the *import/hide* meta operation. An example DIOM IDL specification is provided below:

[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: MDR1

```
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: MDR2

```
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: MDR3

```
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 5: Fragments of the information sources relevant to the medical claim folder scenario

```
CREATE INTERFACE Claim                CREATE INTERFACE Insur-Agreement
(   EXTENT Claims )                   (   EXTENT Insur-Agreements  )
{                                     {
    FROM MDR1                             FROM MDR1
    IMPORT Claim;                         IMPORT Insur-Agreement;
};                                    };

CREATE INTERFACE Image                CREATE INTERFACE DiagnosisRep
(   EXTENT Images )                   (   EXTENT DiagnosisReps  )
{                                     {
    FROM MDR2                             FROM MDR3
    IMPORT Image;                         IMPORT DiagnosisRep;
};                                    }.
```

The IDL pre-processor is responsible for checking the type closure property and the referential integrity of the imported entities, and for generating the concrete IDL specification for each imported entity type. Figure 6 shows the concrete IDL description of the base interfaces, generated by the IDL pre-processor.

To create `ClaimFolder` objects, we need to link the patients' claims in the relational database with their associated X-ray images and the corresponding doctors' diagnosis reports. The classical way is to let application developers be responsible for implementing the `ClaimFolder` objects within their

```
CREATE INTERFACE MDR1:Insur-Agreement
(   EXTENT  Insur-Agreement
    KEY     Insur-no)
{
   ATTRIBUTES
     Integer insur-no;
     String  patient-name;
     String  patient-add;
     Integer ins-category;
     Date    start-date;
);

CREATE INTERFACE MDR1:Claim
(   EXTENT  Claims
    KEY     insur#, claim-no)
{
    ATTRIBUTES
      Integer insur#;
      Integer claim-no;
      Date    claim-date;
      String  status;
      String  treatment;
);

CREATE INTERFACE MDR3:DiagnosisRep
(   EXTENT DiagnosisReps )
{
    ATTRIBUTES
      String insur-no;
      String text;
      ...

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

```
CREATE INTERFACE MDR2:ImageSpec
(   EXTENT ImageSpecs )
{
    ATTRIBUTES
     ...

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

CREATE INTERFACE MDR2:Image
(   EXTENT Images )
{
    ATTRIBUTES
      Integer InsurNum;
      ...

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

Figure 6: Base interface descriptions for the *Medical Claim Folder* application

application programs (say in C or C++). An obvious disadvantage of this approach is the lack of support for reusability. A truly adaptive approach is to encourage consumers to use the DIOM interface aggregation abstraction meta operation to establish the connections among the three underlying data repositories (see Section 3.4.1 for details).

3.2 Base Interface and Compound Interfaces

As mentioned earlier, objects in DIOM are described by their interface specifications in the DIOM Interface Definition Language (DIOM-IDL). To facilitate the interoperation between multiple heterogeneous information sources, we classify the interoperation interfaces into two categories: *base* interfaces and *compound* interfaces.

A **base interface** refers to the interface in which all the types involved in the definition are from a single data source. In order to build links and abstract relationships between data from different data repositories, each entity type defined in the underlying data repositories and visible to the DIOM system may be declared in terms of a base interface. Consider the `Claim Folders` application scenario. In order to provide information consumers (and applications) with transparent access to the source data from the three repositories, and create the `ClaimFolder` objects by linking the related data from the three disparate source repositories (see Figure 5), we may define one or more base interfaces for each of the underlying data repositories (see Figure 6). The set of base interfaces that correspond to a given source can be seen as a *view* over the underlying data source. The scope of a base interface is the corresponding data source.

The **compound interfaces** are constructed through repetitive applications of the interface composition meta operations 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 which the interface definition is based upon.

In DIOM, interfaces defined by means of the interface composition meta operations, such as aggregation abstraction, generalization abstraction, and specialization abstraction, are compound interfaces. The *import/hide* meta operation is designed to serve for information consumers to specify the entity objects of interest from a single information source (see Section3.4.4 for detail).

There are a number of advantages for distinguishing between base and compound interfaces. First, it helps to establish a semantically clean and consistent reference framework for incremental design and construction of interoperation interfaces. The concept of compound interfaces enables information consumers to concentrate more on their query requirements rather than the heterogeneity of the underlying information sources. The concept of base interfaces promotes the separation of information producers' source data models from the information consumer's domain query model. The connection between information consumers and information producers are maintained transparently through the flexible construction of compound interfaces and the automated generation of relevant base interfaces. In addition, a separation of base interfaces from compound interfaces also facilitates the building of a scalable and robust query mediation framework that allows new information sources to be seamlessly incorporated into the existing query mediator.

3.3 Object Identification

Similar to the object-oriented models, the *object identity* (OID) in DIOM is a property of an object that provides a means to uniquely denote the object regardless of its state or behavior [21, 22]. The object identity has two dimensions which influence the degree of support of identity. The scope dimension refers to the scope within which uniqueness can be guaranteed. The temporal dimension expresses the duration in which the OID of an object remains immutable.

In an information mediation framework, for any given application-specific mediator which connects its information consumers with the relevant information sources, the scope of the OID is represented by the specific mediator. The duration is represented by the availability of the mediator. The data object available within an interoperation interface schema are physically stored in the local information sources and can be accessed by means of local identifiers.

Objects (tuples) in relational databases are uniquely distinguished by the user-defined primary key within the scope of their relation. Any local application may change the key value of a tuple if required. Its visibility is necessary as it serves to carry additional user-defined application-specific information.

Objects in object-oriented databases are identified by their unique object identities within the scope of the database. OIDs are generated by the system itself and unchangeable throughout the lifetime of the objects. The visibility of OIDs is not necessary as it serves only for identification purpose through references and object equality comparison.

In file-based data sources, statements about object identifiers are in general not predictable. In this paper we assume that the file-based data sources that we consider all have local identifiers defined either by key field or sequence number of the records.

Thus, in DIOM we provide an OID module to handle object identifiers, so that it maintains the property of uniqueness within its scope and duration; enables access to the objects of component information sources via local object identifiers (LOIDs); and allows for an efficient navigation. The object identification module needs to deal with system generated *global object identifiers* (GOIDs), local object identifiers (LOIDs) if any, and a dynamic binding assignment between a GOID and its related LOIDs.

The initial design of our OID module includes the operations *top, next, assign, connect, compose, rename, and delete* and a data structure that stores the current binding assignments. The assignment of a GOID to a DIOM object proceeds by two consecutive steps:

- use the operation *assign(top)* to assign the the first possible GOID and the operation *assign(next)* to produce a unique GOID in succession and
- call the *connect* operation to bind the LOID of a local object to the generated GOID.

If the DIOM object is of base interface type, this GOID assignment is performed at the source-specific wrapper to reduce the workload at the mediator side. When the object is of compound interface type (e.g., a composite object), the GOID assignment is performed at mediator level, and, instead of *connect*, the *compose* operation is called to relate a set of existing GOIDs to this generated GOID. To ensure the uniqueness of the identification within a mediator scope, each GOID can only be assigned to one DIOM mediated object at most. The GOID of an composite object is different from the GOID of its component object. As local applications are capable of changing the LOIDs of objects in the local source, any change of this kind must be announced to the OID module residing at the source-specific wrapper through the *rebind* operation. The *rebind* operation then changes the LOID of an object accordingly in the current GOID binding assignment table. Thus, the OID module meets the requirement for stable GOIDs. Similarly, the deletion of an object in a local source must be notified to the OID module at the source-specific wrapper through the *unbound* operation. The operation *unbound* then deletes the LOID of an object.

3.4 Interface Composition Meta Operations

One of the main contributions of the DIOM approach is to extend ODMG93 ODL with four meta operations to compose existing base or compound interfaces into more sophisticated compound interfaces. The meta operations *aggregation, generalization*, and *specialization*, are derived from abstractions of the same name [28] that compose and augment interfaces. The meta operation *import/hide* is introduced to facilitate the construction of base interfaces. In the following subsections, we describe each of the meta operations and illustrate their usage by examples.

3.4.1 The Aggregation Meta Operation

Aggregation is a meta operation 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* meta operation is a useful facility for implementing behavioral composition [28] and ad-hoc polymorphism [8] based on coercion of operations.

Recall the `Claim Folder` example given in Section 3.1. To build links that connect a patient's claim 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 6. However, when several applications need to use `ClaimFolder` objects, it would be highly beneficial to create the link objects of type `ClaimFolder` by introducing a compound interface through interface aggregation. An example DIOM IDL definition is given as follows:

```
CREATE INTERFACE ClaimFolder
(   EXTENT ClaimFolders )
{
    AGGREGATION OF Claim, Image, DiagnosisRep;
    RELATIONSHIPS
       Claim        claims;
       Set<IMage>   X-ray-pictures;
       DiagnosisRep report;
    OPERATIONS
       void *get_val(ClaimFolder &cf)
       ...
 }
```

```
CREATE INTERFACE Claim
(   EXTENT Claims )
{
    GENERALIZATION OF
      SELECT interface-name
      FROM InterfaceRepository
      WHERE description CONTAINS 'claim';
    ATTRIBUTES
      Integer insur#;
      String  patient-name;
      ...
}
```

```
CREATE INTERFACE Image
(   EXTENT Images )
{
    GENERALIZATION OF
      SELECT interface-name FROM InterfaceRepository
      WHERE description CONTAINS 'image';
    ATTRIBUTES
       Integer InsurNum;
       ...
    OPERATIONS
```

```
CREATE INTERFACE DiagnosisRep
(   EXTENT DiagnosisReps )
{
    GENERALIZATION OF
      SELECT interface-name
      FROM InterfaceRepository
      WHERE description CONTAINS 'diagnosis';
    ATTRIBUTES
      String insur-no;
      String text;
```

```
        Image *display(void *);                               ...
        ...
    }                                                     }
```

Figure 7 presents a sketch of the relationship between compound interfaces and between a compound interface and the related base interfaces, as well as the correspondence between base interfaces and the source entity types. It also shows how the consumer's application-specific `ClaimFolder` objects is defined in DIOM IDL and connected to the corresponding source data from disparate data repositories.

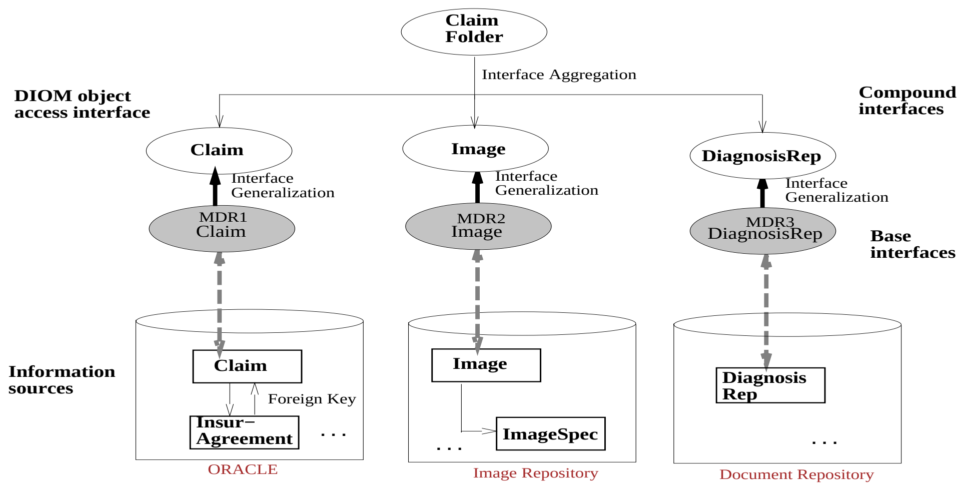


Figure 7: Access to objects from multiple information sources through a compound interface

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 of the `ClaimFolder` mediator. The default is the `transient` option. When a `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.

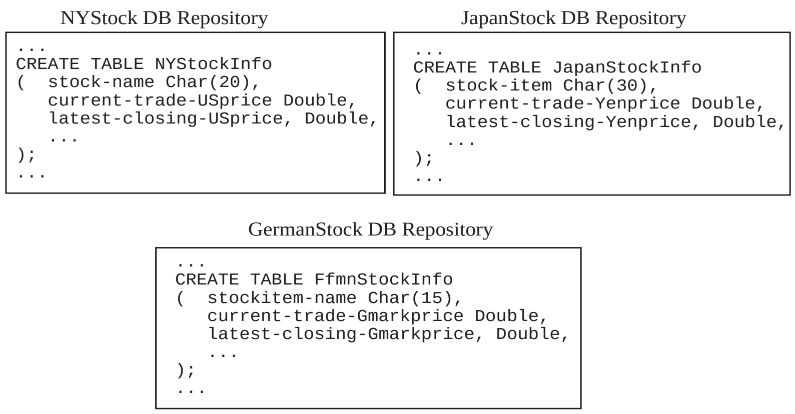
The use of meta operation `generalization` in this example IDL specification will be discussed in the next section.

One of the benefits of using the meta interface operation `aggregation` is to allow consumers to interconnect objects that reside in disparate data sources according to their business objectives and domain interests.

3.4.2 The Generalization Meta Operation

Generalization is a meta operation that provides a convenient facility to merge several semantically similar and yet different interfaces into a more generalized interface. The main idea by interface generalization is to define a new interface by abstracting the common properties and operations of some existing (base or compound) interfaces. The interface generalization meta operation enables objects that reside in disparate data repositories to be accessed and viewed uniformly through a generalized DIOM interface. It also provides a means to assist the delay of resolution of representational and semantic heterogeneities to the query result assembly stage.

Consider the example shown in Figure 8. Suppose we have three stock trade information sources currently available to the system: `NYStockInfo`, `TokyoStockInfo`, and `FfmStockInfo`. For presentation brevity, let us assume that these three repositories are all relational databases and are created and maintained separately. Figure 8 shows the relevant portion of the sample export schemas of these three stock trade data repositories. Assume that an information consumer wants to query the *latest-*


NYStock DB Repository
JapanStock DB Repository
GermanStock DB Repository

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

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

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

Figure 8: Fragments of the source specifications of `NYStocks, TokyoStocks, FfmStocks`

closing price and the *current-trade price* of a stock and receive the query result in some desired currency (say US dollar) no matter whether the stock was traded in New York, Tokyo, Frankfurt, or any other international cities. One convenient way to satisfy this requirement is to create a new compound interface `StockTrade` through interface generalization abstraction as shown in Figure 9:

Note that, first, in order to allow the definition of this compound interface `StockTrade` to scale well to the growing number of information sources becoming available to the system, we use the view-based interface generalization construct to define this compound interface `StockTrade` as shown in the line `(3)` to line `(5)`. Of course, if the information consumer is only interested in the stock information from the market in New York, Tokyo, and Frankfurt, then we may easily replace the view-based interface generalization abstraction by a snapshot-based interface generalization such as "`GENERALIZATION OF NYStocks, TokyoStocks, FfmStocks`". Second, each attribute signature may associate with a `WHERE` clause to specify the special preference that the consumer might have on how the query result should be received. In the above example, line `(9)` to line `(11)` specify that the consumer would like to receive all the price information about stocks in US dollar no matter where the stocks are traded.

DIOM provides facilities to support *domain-specific library functions* which can be used to resolve large

```
CREATE INTERFACE StockTrade
(   EXTENT StockTrades )
{
    GENERALIZATION OF
       SELECT interface-name FROM InterfaceRepository
       WHERE  description CONTAINS 'Stock';
    ATTRIBUTES
       String    stockname;
       Double    current-trade-price
                 WHERE scale = 'USD';
       Double    latest-closing-price
                 WHERE scale = 'USD';
 }
```

Figure 9: Creating a compound interface through interface generalization

amount of representational heterogeneity and certain amount of semantic mismatches. For example, in the mediator for *stock trade* application, the domain-specific library function for the currency exchange conversions may use the currency exchange rates of the day on which the queries are issued as the default criteria. In addition, users may override the system-maintained 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 then be invoked.

The interface generalization operation can also be combined with the interface aggregation operation. Such a combination not only minimize the impact of component schema changes over the consumers' application programs working with the existing interoperation interfaces, but also enhance the scalability of query mediation services with respect to the growing number of information sources. For example, consider the `ClaimFolder` scenario. Suppose a new image source, called `CAT-Scan` needs to be added into the `Claim Folder` application. After creating a base interface `CAT-Scan-Image` for this new image source, the change to the catalog of information sources is notified to the DIOM query mediation manager. By recompiling the generalization interface `Image`, both the base interface `MDR2:Image` and the base interface `CAT-Scan-Image` for the newly added image source `CAT-Scan` will now be used to answer queries over the `Image` objects. There is no need for manually rewriting of any existing interface definitions or application programs working with the original `ClaimFolder` interface schema. Subsequent executions of query applications will seamlessly incorporate this new data source in the interoperation process.

3.4.3 The Specialization Meta Operation

Specialization is a useful meta operation for building a new interface in terms of some existing interfaces through adding new attributes, relationships, or operations. For example, suppose that the information consumer wants to add a new attribute, such as the difference between current-trade price and the last-closing price, to the compound interface `StockTrade` defined in Figure 3.4.2. A truly adaptive and robust way to satisfy this change requirement is to define a new compound interface by means of interface specialization abstraction of the `StockTrade` interface. This is primarily because, by introducing a new interface to capture the change, existing queries or applications defined based on the `StockTrade` interface can remain to be operational without any rewriting effort.

3.4.4 The Import/Hide Meta Operation

The meta operation *import/hide* is designed for importing selected portions of the data from a given information source, instead of importing everything that is available. For a information source that manages complex objects, the import and hide meta operation performs automatic checking of the type closure property and referential integrity of the imported classes. The type closure property refers to the type consistency constraint over subclass/superclass hierarchy such that whenever a class is imported, all the properties and operations it inherits from its superclasses have to be imported together unless being explicitly excluded by `HIDE` statement. 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 classes.

Consider an example given in Figure 10, where the export source database schema `UnivDB1` consists of nine classes `Person, Employee, Department, Professor, Staff, Student, TA, RA` and `Course`. We omit the syntactical definition of the source schema for presentation brevity. Suppose an information consumer wants to create a `University` interface by importing only a portion of the data, which is related to teaching faculty or TAs, from the source `UnivDB1`. We may specify the `University` interface by using the DIOM import facility as shown in Figure 11(a). Thus, three classes `Person`, `TA`, and `Professor` are explicitly imported. According to the referential integrity and type closure property, the class `Course` needs to be imported as well, because the `Course` objects are referenced by the imported classes `Professor` and `TA`. (see Figure 11(b)).

However, there is no need to import the entire type structure of `Address, Department, RA, Staff`. The reasons are the following:

- The `Address` objects are only referenced by `Person` objects through the attribute `address` which is explicitly excluded in the `University` interface description through `HIDE` facility associated with the import meta operation.
- The classes `RA` and `Staff` are neither referenced nor superclasses of any of the three imported classes.
- The class `Department` is explicitly excluded from the importing list through the `HIDE` facility associated with the import meta operation. Thus, all the references to `Department` objects from

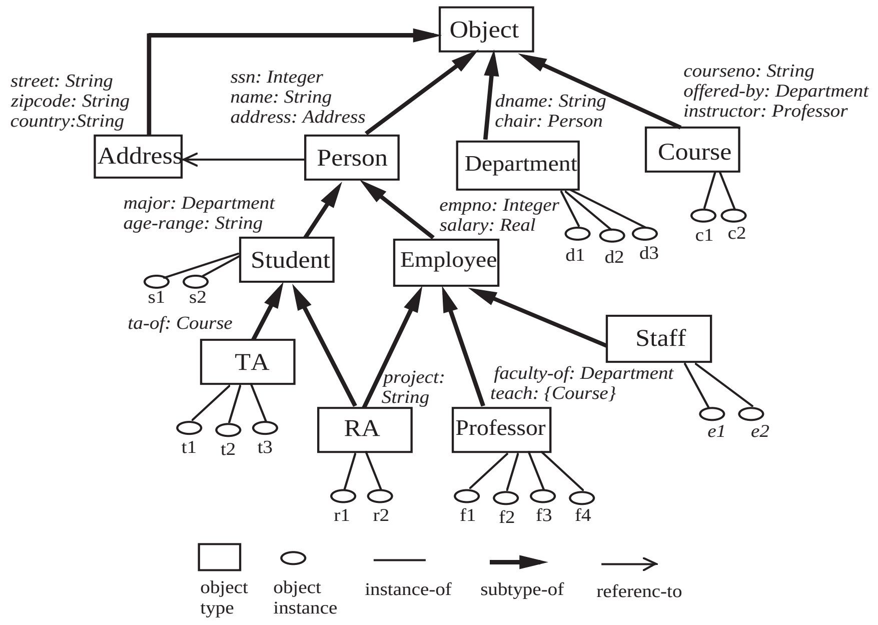


Figure 10: A sample source schema of the source database `UnivDB1`

the imported objects should be modified accordingly. Hence, the attribute `faculty-of` of class `Professor` is modified from domain type `Department` to refer to, for example, the department name of type `String`.

Using the import/hide meta operation, a number of benefits can be obtained. First, by means of the import facility, information consumers need only to specify the key information that are of interest to their intended applications. The system will automatically infer the rest of the classes that need to be imported in order to preserve the referential integrity and type closure property. Second, the use of import facility 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 a large amount of classes from a single data source, using the import/hide meta operation may relieve the consumers from the tedious job of specification of base interfaces for each of the source entity types. Last but not least, the interoperation interfaces constructed through the import facility exhibit higher robustness and adaptiveness in the presence of source schema changes.

3.5 Benefits of DIOM IDL

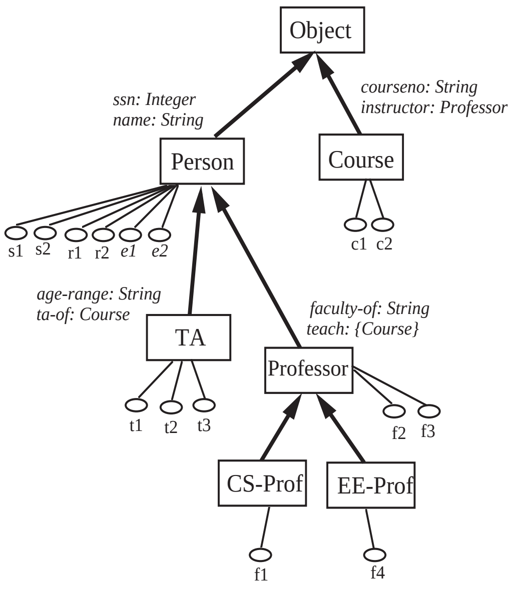
The consumer's interface schema defined in DIOM-IDL differs from the conventional global schema approach to multidatabase integration in at least three ways:

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

1. Unlike in the tightly-coupling approach to schema integration [42] where equivalent attributes are merged into one attribute, the DIOM mediated attributes are only given in a logical form. That is, a mediated attribute is solely a semantic correspondence to, rather than an absolute integration/generalization of, the equivalent attributes in component information sources. Difference in domain and operations do not matter since no attributes are merged in the multidatabase system. Thus, the implementation details about coding and data structure are not necessary.

2. Each DIOM interface schema only represents an information consumer's personal view of how the result of a query should be captured and represented. We call it the domain query model. There is no global schema that is created by the system itself for all the multidatabase users. The interconnection between a consumers' domain query model and the producers' information sources is established dynamically at the query processing stage, rather than at the interface definition phase. For each consumer query model, the DIOM metadata naming convention is used to denote the consumer-defined interface types, attributes and relationships. For example, the attribute naming convention used to denote the mediated attributes is to associate each mediated attribute with its interface type name through arrow notation (e.g., `Insur-Agreement->insur-no`).

3. The interface composition meta operations supported by DIOM-IDL ensure not only the seamless extensions to the consumers' interface schemas, but also the scalability of the consumer's IDL specification. Each consumer-defined IDL interface schema is maintained as a fully dynamic window, rather than a relatively static "view", of the global information. Upon arrival of a new information

source, an incremental recompilation of the relevant consumer IDL interface descriptions will be performed transparently to ensure a seamless incorporation of the new information sources into the global query routing and query planning process. From information consumers' point of view, no manual interface schema modification is required upon the arrival of new information sources.

4 Query Processing in DIOM

Queries to multiple information sources are expressed in the DIOM interface query language (IQL). These queries use the naming convention and terminology defined in the information consumer's domain model. There is no need for the query writers to be aware of the many different naming conventions and terminology used in the underlying information sources. Given a query, an information mediator identifies the appropriate information sources, decompose the query expressed in terms of the consumer's domain model into a collection of subqueries, each expressed in terms of an information producer's source model, and then forwards these reformulated queries to the corresponding sources for subquery translation and execution. The approach of building wrappers to coordinate between a mediator and its component systems greatly simplifies the implementation of individual mediators, since each mediator only needs to handle one underlying language. It also makes the interoperation among the networks of mediators easily scalable to the ever growing number of information sources.

4.1 Informal Overview of DIOM-IQL

The DIOM interface query language (IQL) is designed as an object-oriented extension of SQL. The basic construct in DIOM IQL is a SQL-like `SELECT-FROM-WHERE` expression, with the following syntax:

```
[TARGET <Source-expressions>]
SELECT <Fetch-expressions>
FROM   <Fetch-target-list>
[WHERE  <Fetch-conditions>]
[GROUP BY <Fetch-expression>]
[ORDER BY <Fetch-expression> ]
```

A detailed syntax description of a DIOM IQL expression is provided in Appendix B. The IQL design follows a number of assumptions: First, IQL by itself is not computationally complete. However, queries can invoke methods. Second, IQL is truly declarative. It does not require users to specify navigational access paths or join conditions. Third, IQL also deals with the representational heterogeneity problem. For example, different information sources may have different structural and naming conventions for the same real world entity. These information producers' representations may not fit to the information consumer's preference. Using IQL, the information consumer may specify a query in terms of his/her own IDL specification or express a query without having a pre-defined IDL schema. The system is responsible for mapping a consumer's query expression into a set of information producer's query expressions.

Furthermore, IQL allows the use of object type names of information sources in the FROM clause by prefixing the source name with dot notation. It is also possible to use the information source names directly in the FROM clause of a query. The result of an IQL query is by itself an object of the special interface type `QueryResult`. The main objective of these IQL extensions is to increase accessibility for information consumers and to improve the quality of global information sharing and exchange.

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, DiagnosisRep` as defined in Figure 6, and the compound interface `ClaimFolder` as specified in Section 3.4.1. 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 paid the insurance premium for less than three months (90 days). The desired query answer form is a collection of patients, each patient is presented by the name and the insurance number of the patients and the set of corresponding X-ray images. Assume, this query requires to perform an inter-source join over three disparate data repositories: Insurance database, Document repository, and the image repository.

Without using DIOM interface schema, the information consumer has to learn to use different languages in order to write three separate queries correctly, one for each source repository. Besides, the consumer also needs to provide a program that implement a join over the three sets of query results returned and package them into the desired output format. Furthermore, whenever a new data source becomes available and is relevant to this query request, the consumer's program has to be rewritten accordingly.

With DIOM IDL/IQL services, the user may easily express this query in IQL against the `Claim Folder` interface schema as shown in Figure 12:

```
Q1:   SELECT I->patient-name, I->insur-no, F->X-ray-pictures
      FROM   ClaimFolder F, Insur-Agreement I
      WHERE  F->report->search("dental" && "bridge")
      AND    (F->claims->claim-date - I->start-date) < 90;
```

Figure 12: An example query

First, in representing this query, the information consumer does not need to be concerned about the different naming conventions and different structural design. Instead, she can write a query using the terminology that is preferable in her own application domain. Second, the query writer need not specify any "join" conditions in her query expression. The IQL preprocessor is able to automatically insert the necessary connection paths to relate different object types involved in the query with each other, since such connection semantics should be easily derived from the metadata maintained in the DIOM interface repository and implementation repository. Note that, unlike the conventional SQLs, in IQL the `FROM` clause may allow zero or more target objects, and the "joins" between target objects may not be specified at the top level `WHERE` clause of the query.

Another interesting feature of the DIOM query model is the automatic creation of a result type for each consumer query. It means that, for every consumer query expressed in IQL, a compound interface will be generated as the result type of the query, and maintained in the DIOM interface repository. This functionality is particularly important for preserving the closure of the DIOM object model and for assembly of multidatabase query results in terms of consumer's preferred representation. Consider the query "*find the name, insurance number and the X-ray pictures of all the patients who started their insurance in 1995*". We may express this query in IQL as follows:

```
Q2:  SELECT I->patient-name, I->insur-no, F->X-ray-pictures
     FROM   ClaimFolder F, Insur-Agreement I
     WHERE  I->start-date > 1995;
```

The following compound interface description will be generated as the result type of this query:

```
CREATE INTERFACE QueryResult<Q2>
( extent ResultObjects<Q2> )
{
   AGGREGATION OF ClaimFolder, Insur-Agreement;
   ATTRIBUTES
       String     Insur-Agreement->patient-name,
       String     Insur-Agreement->insur-no,
       Date       Insur-Agreement->start-date,
       Set<Image> ClaimFolder->X-ray-pictures;
 }
```

We use the basic techniques described in [28] as the formal foundation for automatic generation of a query result type for each DIOM IQL query expression.

4.2 Query Mediation Process: An Overview

The main task of a distributed query mediation manager is to coordinate the communication and distribution of the processing of information consumer's query requests among the root mediator and its component mediators or wrappers (recall Figure 1). Figure 13 presents the general procedure for distributed query processing in Diorama. The main component services include:

- **Query routing**
  The main task of query routing is to select relevant information sources from available ones for answering the query. This is done by mapping the domain model terminology to the source model terminology, by eliminating null queries, which return empty results, and by transforming ambiguous queries into semantic-clean queries.

- **Query decomposition**
  This is done by decomposing a query expressed in terms of the domain model into a collection of

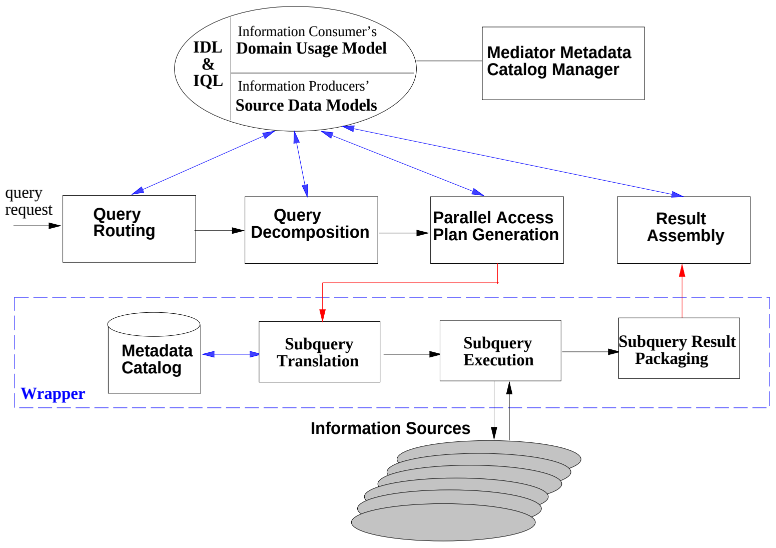


Figure 13: The distributed query scheduling framework in DIOM

queries, each expressed in terms of a single source model. Useful execution dependencies between subqueries are captured during query decomposition phase and will be used in parallel access planning and query result assembly.

- **Parallel access plan generation**
  The goal of generating a parallel access plan for the groups of subqueries is to find a relatively optimal schedule that makes use of the parallel processing method and the useful execution dependencies between subqueries, resulting from built-in heuristics, to minimize the overall response time and reduce the total query processing cost. A optimial schedule is composed of single-source queries (each posed against exactly one local export schema), move operations that ship results of the single-source queries between sites, and postprocessing operations that assemble the results of the single-source queries in terms of the information consumer's query request expressions.

- **Subquery translation and execution**
  The translation process basically converts each subquery expressed in terms of an IDL source model into the corresponding information producer's query language expression, and adds the necessary join conditions required by the information source system.

- **Query result packaging and assembly**
  The result assembly process involves two phases for resolving the semantic variations among the subquery results: (1) packaging each individual subquery result into a DIOM object (done at wrapper level) and (2) assembling results of the subqueries in terms of the consumers' original query statement (done at mediator level). The semantic attachment operations and the consumers' query profiles are the main techniques that we use for resolving semantics heterogeneity

implied in the query results.

Since this paper is targeted at the design of IDL interface composition meta operations and their applications to the access of heterogeneous information sources through DIOM IQL queries. We omit the details on query processing strategies and techniques used for each step [32, 29]. In what follows, we first overview the strategies and techniques used in the query decomposition step, and then use an example to highlight the role of IDL interface descriptions in the DIOM query processing.

4.3 Query Decomposition

The goal of query decomposition is to break down a multi-information source query into a collection of subqueries, each targeted at a single source. The query decomposition module takes as input the IDL query expression modified (reformulated) by the dynamic query source selection module, and performs the query decomposition in two phases:

1. **target split**. This is done by decomposing the query into a collection of subqueries, each targeting at a single source. These subqueries can be either independent of or dependent upon each other. The procedure of target split consists of the following steps:

   (a) For each compound interface involved in the query,

   - if it is defined in terms of aggregation abstraction, then an aggregation-based join [29] over the component interfaces is used to replace this compound interface;
   - if the compound interface is defined in terms of generalization abstraction, then a distributed-union (D-union) [29] over the specialized interfaces is used to replace this compound interface.

   (b) Move the information sources specified in the target clause downward and closer to the leaves of the query tree. Metadata in the interface repository are used to lead the moving of each information source to an appropriate leaf node of the query tree.

   (c) Group the leaf branches of the query tree by target source.

   (d) Use conventional query processing tactics, such as moving selection down to the leaf nodes, perform selection and projection at individual sources, and attach appropriate subset of the projection list to each subquery before executing any inter-source joins and inter-source unions.

   The process of target splitting results in a set of subqueries and a modified query expression that is semantically equivalent to the original query, and that presents the operations of how the subqueries should be combined to produce the answer for the original multi-information source query.

2. **Useful subquery dependency recording**. This is done by recording all the meaningful subquery execution dependencies with respect to the efficiency of the global query processing. Two synchronization operations are supported: *parallel* "$||$" and *sequential* "$\mapsto$". Any combinations of these two operations are also allowed.

   For each consumer's query, if the number of subqueries resulting from the query decomposition phase is k, then it is easy to see that the number of combinatorial ordering alternatives increases rapidly when the number of subqueries (i.e., k) becomes large. Not surprisingly, many of the possible synchronization alternatives will never be selected as the "optimal" parallel access plan for the given query. For example, assume that, for any two subqueries, the independent evaluation of a subquery subQ_1 is more expensive than the independent evaluation of subquery subQ_2. Thus, the synchronization scheme that executing subQ_1 first and then ship the result of subQ_1 to another source to join with the subquery subQ_2 is, relatively speaking, an undesirable access plan because of the poor performance it provides. We develop a number of heuristic rules that may be used to rule out those obviously undesirable synchronization alternatives. A detailed discussion will be reported in a forthcoming paper. Here are some examples of such heuristics applicable to the subqueries whose results may not be union-compatible:

   - The selectivity of a *Fetch* condition with one of the following operators "$<=, <, >, >=$" is lower than the selectivity of a *Fetch* condition with the equality operator "$=$", but higher than the selectivity of a *Fetch* condition with "$<>$" operator.
   - Subqueries of the highest selectivity are executed first. If there is more than one subquery of highest selectivity, then those subqueries should be executed in parallel.
   - Subqueries of the lowest selectivity are executed last.

   The idea of applying these simple heuristics is to use different selectivity factors of different query predicates (*fetch conditions*) to estimate the cost difference among various subqueries. When the selectivity of a subquery is higher, the size of the subquery result tends to be relatively small. Thus it is always a recommended tactic to execute those subqueries of higher selectivity earlier.

It is well known that the quality of a query processor relies on the efficiency of its query processing strategies and the performance of its query execution plans. Furthermore, the performance of a distributed query processing plan is not only determined by the response time of the local subqueries but also affected by the synchronization scheme chosen for synchronizing the execution of subqueries. A good synchronization scheme may utilize the results of some subqueries to reduce the processing cost of the other subqueries whenever it is beneficial.

4.4 An Example

Suppose that after passing through the query routing phase, the necessary connection paths (e.g., `I->insur-no == F->claims->insur#`) and the minimal set of candidate information sources (e.g., `MDR1, MDR2, MDR3`) are added into the query expression. The original query `Q1` is reformulated as follows:

```
TARGET MDR1, MDR2, MDR3
SELECT I->patient-name, I->insur-no, F->X-ray-pictures
FROM   ClaimFolder F, Insur-Agreement I
WHERE  F->report->search("dental" && "bridge")
AND    (F->claims->claim-date - I->start-date) < 90
AND    I->insur-no == F->claims->insur#;
```

Figure 14(a) presents the query graph for this query, where

- X_1 denotes the fetch-expression "`I->patient-name`,
- X_2 denotes the fetch-expression "`I->insur-no`, and
- X_3 denotes the fetch-expression "`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`".

The query decomposition for this example is performed in the following steps:

Step 1: Target Split

Based on the domain usage model for the `Claim Folder` application, the interface `ClaimFolder` is a compound interface defined based on an aggregation of the three compound interfaces: `Claim, Image, DiagnosisRep`. The interface `Insur-Agreement` is a base interface.

- We first replace the compound interface `ClaimFolder` by an aggregation-based join over the three interfaces `Claim, Image`, and `DiagnosisRep`. An aggregation-based join refers to an object join which maintains the nested aggregation structure while traversing through the objects following the aggregation path [28]. The query tree in Figure 14(a) is then transformed into the query tree in Figure 14(b).
- By moving the query target downward and close to the leaf nodes, the query tree is modified as shown in Figure 14(c).
- After grouping the query leaf nodes by target source, the query tree is modified again and is shown in Figure 14(d).
- Now by applying the high-level query optimization heuristics, such as performing selections and projections before inter-source join and inter-source union, finding common subexpressions, Figure 14(e)) presents the modified query tree as the result of target split.

As a result of the target split, the original IQL query is decomposed into the following three IQL subqueries:

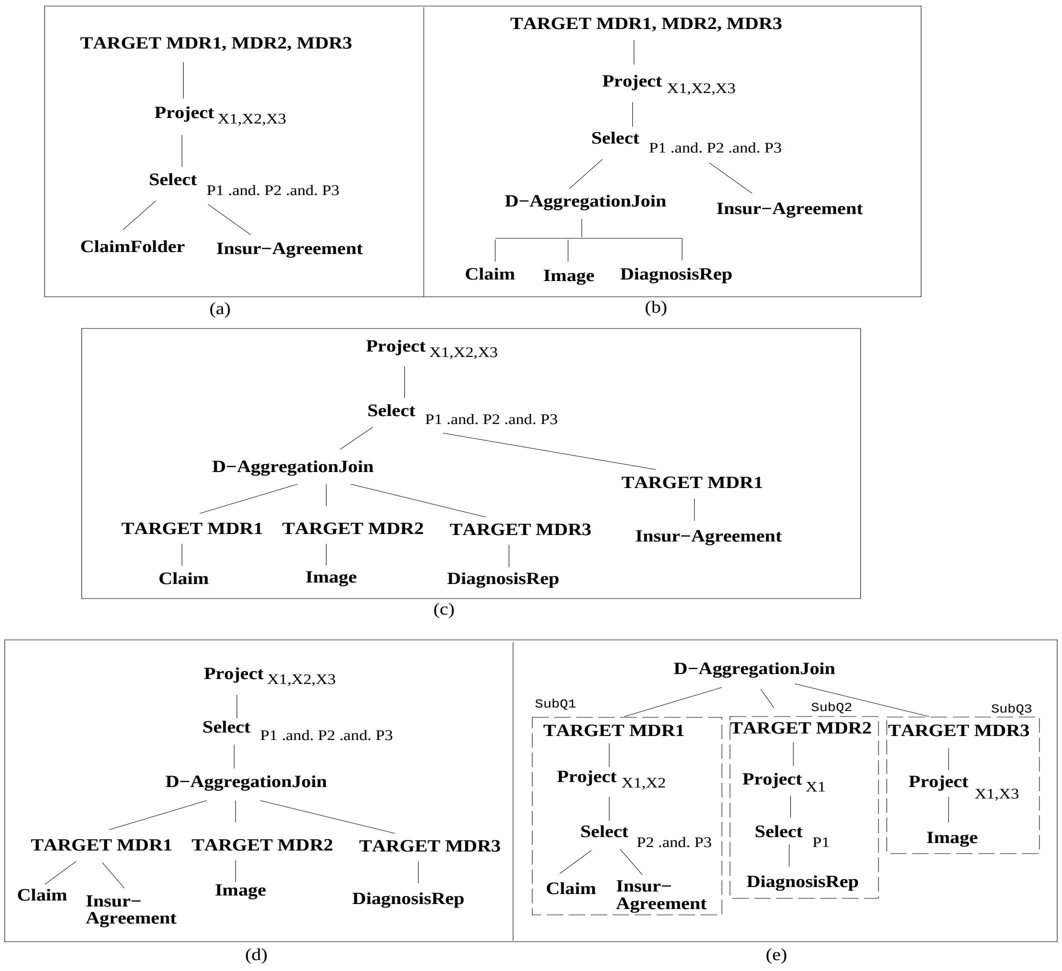


Figure 14: Query decomposition in DIOM: an example

```
SubQ1: TARGET MDR1
        SELECT I->patient-name, I->insur-no
        FROM   Claim C, Insur-Agreement I
        WHERE  (C->claim-date - I->start-date) < 90
        AND    I->insur-no == C->insur#;


 SubQ2: TARGET MDR2
        SELECT D->insur-no
        FROM   DiagnosisRep D
        WHERE  D->search("dental" && "bridge");


 SubQ3: TARGET MDR3
        SELECT P->insur-num, P->display()
        FROM   Image P;
```

Step 2: Useful subquery dependency recording

From the resulting query graph given in Figure 14(e), the possible execution dependencies that can be used for synchronization of the executions of subqueries `SubQ1, SubQ2, SubQ3`, in addition to those sequential alternatives, are the following:

1. `SubQ1` $||$ `SubQ2` $||$ `SubQ3`.
2. `SubQ1` $\mapsto$ `SubQ2` $||$ `SubQ3`.
3. `SubQ2` $\mapsto$ `SubQ1` $||$ `SubQ3`.
4. `SubQ3` $\mapsto$ `SubQ1` $||$ `SubQ2`.
5. `SubQ1` $||$ `SubQ2` $\mapsto$ `SubQ3`.
6. `SubQ2` $||$ `SubQ3` $\mapsto$ `SubQ1`.
7. `SubQ1` $||$ `SubQ3` $\mapsto$ `SubQ2`.

Each dependency alternative represents a possible synchronization plan for subquery executions. Based on the selectivity estimation heuristics given at the beginning of Section 4.3, we observe the following facts: First, the result of `SubQ1` and `SubQ2` can be used to restrict the search space of `SubQ3`. Second, `SubQ2` has a lower level of selectivity than `SubQ1` does. Therefore, based on the first observation we may drop those execution dependencies with `SubQ3` at the start of the execution sequence. Namely, the execution dependencies 4, 6, 7 are dropped. Based on the second observation, those execution dependencies which do not include `SubQ2` at the start of the execution sequence are disregarded too. Thus the execution dependency 7 is dropped. As a result, the useful subquery execution dependencies being recorded are the following three:

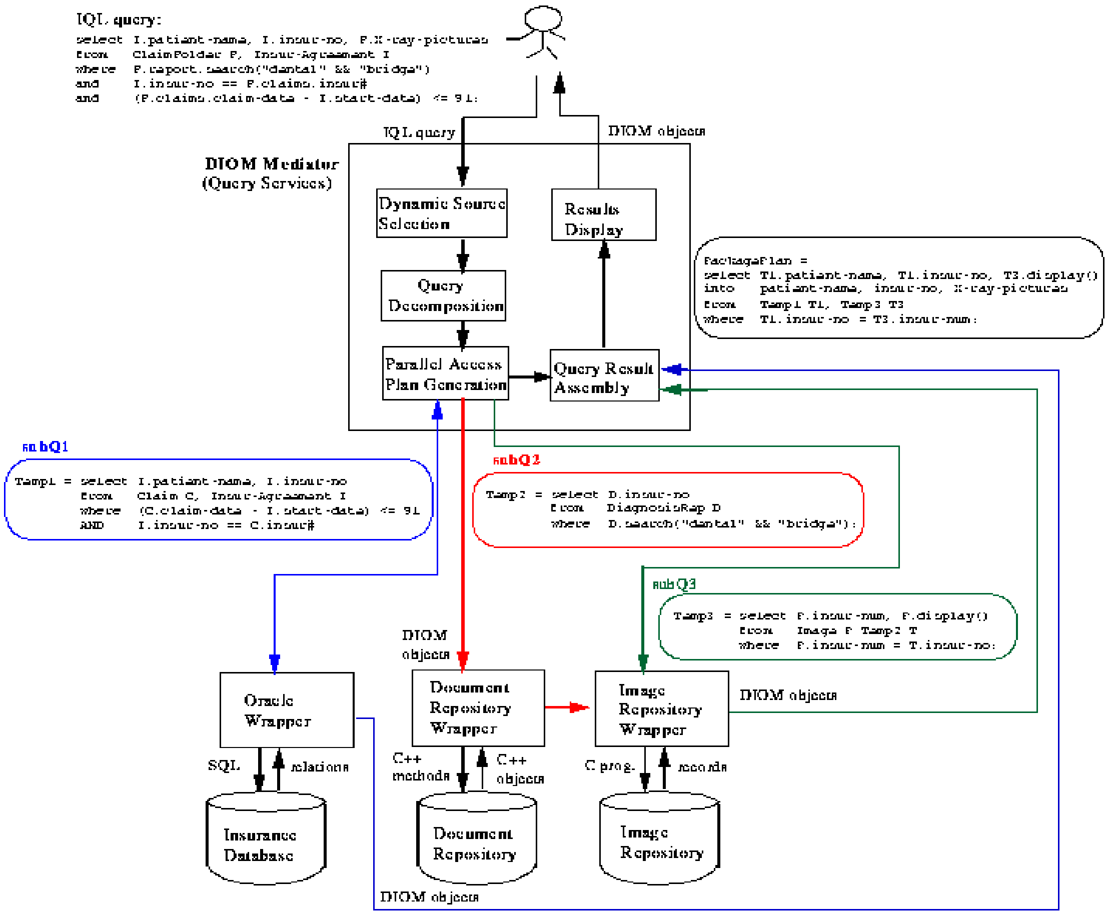
- (1) `SubQ1` $||$ `SubQ2` $||$ `SubQ3`.
- (3) `SubQ2` $\mapsto$ `SubQ1` $||$ `SubQ3`.
- (5) `SubQ1` $||$ `SubQ2` $\mapsto$ `SubQ3`.

A final decision will be made at the phase of the parallel access plan generation, based on the estimation of the total cost of subquery processing and query results assembly.

Assume that `SubQ1` $||$ `SubQ2` $\mapsto$ `SubQ3` is the final access plan generated by parallel access planning step. Let `Temp2` denote the result of subquery `SubQ2`. Thus, only the subquery `SubQ3` needs to be reformulated accordingly, as follows:

```
SubQ3: TARGET MDR3
       SELECT P->insur-num, P->display()
       FROM   Image P, Temp2 T
       WHERE  P->insur-num = T->insur-no;
```

Now, the subqueries SubQ1, SubQ2, SubQ3 in IQL expressions will be passed from the mediator to the corresponding wrappers for subquery translation and execution. (see the subqueries SubQ1, SubQ2, SubQ3 in Figure 15). The wrapper will first convert the IQL query into a query expression that is



IQL query:

```
select  I.patient-name, I.insur-no, 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) <= 91;
```

```
PackagePlan =
select  T1.patient-name, T1.insur-no, T3.display()
into    patient-name, insur-no, X-ray-pictures
from    Temp1 T1, Temp3 T3
where   T1.insur-no = T3.insur-num;
```

subQ1

```
Temp1 = select  I.patient-name, I.insur-no
        from    Claim C, Insur-Agreement I
        where   (C.claim-date - I.start-date) <= 91
        AND     I.insur-no == C.insur#
```

subQ2

```
Temp2 = select  D.insur-no
        from    DiagnosisRep D
        where   D.search("dental" && "bridge");
```

subQ3

```
Temp3  = select  F.insur-num, F.display()
         from    Image F Temp2 T
         where   F.insur-num = T.insur-no;
```

Figure 15: The DIOM query processing strategies

understandable to the data source to which the wrapper serves. When the repository is a RDBMS, say Oracle, the wrapper will map an IQL expression into an Oracle/SQL query statement. It is also the wrapper's responsibility to collect and package the result in terms of the DIOM objects before sending the result of a subquery back to the mediator (see Figure 15). If the data source is an OODBMS (say ObjectStore), the wrapper will map each IQL expression into an ObjectStore query that is bounded to an ObjectStore class dictionary. When the data source is stored and managed solely by a file server, say in the form of HTML or SGML, then the wrapper will map the IQL expression into, for example, a C module or a Perl 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 multi-source information browser, which currently runs through two types of wrappers: an Oracle wrapper for structured data sources and a HTML wrapper for accessing unstructured or semi-structured web data sources (e.g., WWW pages, NetNews articles). We are currently building wrappers to ObjectStore databases and to *bibtex* files, in addition to the wrappers to Oracle databases and to HTML files.

4.5 Subquery Translation

The subquery translation module is in charge of translating a subquery in IQL into a source-specific query in the source query language or a piece of program that can be executed over the corresponding source. This is done in three steps:

1. **Attribute conversion and type replacement**. This is done by looking up, from the metadata maintained in the implementation repository associated with the corresponding wrapper, the matching expression for each attribute and entity type in the IQL query expression, and then replacing all the attributes and interface types defined at the domain query model level by their matching attributes and entity types at information source model level. The target information source names are used to lead the search and conversion.
2. **Completion of the subquery expression**. Each information source may have its specific requirements for representing a query over the source. For example, if the source is an Oracle database, a subquery over more than one relations must provide explicit join conditions among these relations to make this subquery executable over the Oracle database source. The subquery expression completion step is to guarantee that the subquery to be executed at the source conforms to the source-specific query language requirement.
3. **Subquery refinement** The purpose of subquery refinement is to utilize the available constraints to further reduce the search scope of the subquery and to detect null queries.

Assume that the wrappers to the document repository `MDR2` and to the image repository `MDR3` both take the SQL-like statement as input and invoke the appropriate program modules to search through the repository and find the matching records. By passing the subqueries in IQL expressions through the subquery translation process, the following subqueries are generated, each is executable at the corresponding data source:

```
SubQ1: TARGET MDR1
       SELECT I.patient-name, I.insur-no
       FROM   Claim C, Insur-Agreement I
       WHERE  (C.claim-date - I.start-date) < 90
       AND    I.insur-no = C.insur#;
```

```
SubQ2: TARGET MDR2
       SELECT D.insur-no
       FROM   DiagnosisRep D
       WHERE  D.search("dental" && "bridge");

SubQ3: TARGET MDR3
       SELECT P.insur-num, P.display()
       FROM   Image P, Temp T;
       WHERE  P.insur-num = T.insur-no;
```

4.6 Query Result Assembly

Once all the subqueries are processed and the results are returned to the wrappers (the DIOM local agents), the results must be assembled and attached with additional semantic information before being presented to the user in terms of the user's preferred terminology.

Consider the three subqueries in the previous section. Let us assume that the result of `SubQ1` is stored in `Temp1`, the result of `SubQ2` is stored in `Temp2`, and the result of `SubQ3` is stored in `Temp3`. In terms of the "optimal" access plan `SubQ1` $\|$ `SubQ2` $\mapsto$ `SubQ3`, the subquery result `Temp2` has been used in answering the subquery `SubQ3`. Thus, according to the consumer's original query described in Figure 12, the following query result assembly plan is generated to package the results of the subqueries:

```
SELECT T1->patient-name, T1->insur-no, T3->display()
INTO   patient-name, insur-no, X-ray-pictures
FROM   Temp1 T1, Temp3 T3
WHERE  T1->insur-no = T3->insur-num;
```

One of the interesting features of the DIOM result assembly service is the use of semantic attachment approach to the resolution of heterogeneity problems, rather than enforcing the integration based on system imposed decision. Information consumers may explicitly specify the representation, in which they prefer to receive the query result, either in the IDL interface description or in the user profiles. The techniques we have developed include source attachment, scale attachment, alternation attachment, access path attachment, and absent attribute specification. Readers may refer to [29] for more detail.

5 Implementation Considerations

A prototype of the DIOM information mediation methodology is currently being constructed with the Oracle 7.0 as the back-end service repository support. The first demonstration package [27, 26] is implemented as a WWW application. All the components and its interface functions are created using HTML [13] and Perl CGI-scripts [35, 44]. Linking to the underlying data sources and metadata library databases is implemented with Oraperl [4]. Components of the prototype system interact via HTTP [16] protocol, a popular protocol that provides good flexibility in types of data to be sent or received (both text and binary). In this demo system, the main components implemented include

- the user interface made up of two components: (i) the HTML markup language (including frames) which the Netscape browser actually interprets and displays, and (ii) the underlying Perl CGI modules which are executed by the WEB daemon to fetch, transform, and normalize data, access other Diorama prototype modules or Oracle engine (through Oraperl), and return valid HTML code (new web pages) to display. Generally speaking, the Perl scripts are activated at client side, process data at sever side and return valid HTML code back to the client side browser for display.
- the HTML wrapper which translates a DIOM IDL query into a request to the targeted HTML browser. Currently we have two instances of such a wrapper: Yahoo [49] wrapper and Harvest [7] wrapper.

The rest of the components in our demo system, such as the expert query browser, the query trace function, and the query decomposition module, are currently simulated.

We have made significant progress towards the second phase of the DIOM prototype, which will focus on three main aspects: (1) the creation of consumer's profile, the metadata catalog management, described in [29, 31], and the producers' profile organization; (2) the realization of the query decomposition algorithm, described in [32] and query assembly operations based on consumer's profile, producers' registration data, and domain-specific ontology or thesaurus; and (3) the algorithms for generating optimized parallel access plans. We are also interested in developing generic wrapper functions that can be used to generate wrapper instances for various types of data repositories.

6 Related Work

Since there are many research projects working towards effective support for interoperability across software and applications systems, we summarize here the main two distinctive features of DIOM: (1) the adaptation of reflective concept [23, 33] to distinguish and support interface composition through meta operations; and (2) the augmentation of existing standards such as CORBA and ODMG93 with new ideas such as separation of base and compound interfaces and interface composition meta operations. Several other approaches have been proposed for information gathering from multiple heterogeneous information sources, including multidatabases and federated databases, distributed object management, intelligent integration of information architectures [45], and standard interfaces for software integration and communication such as IBM's DSOM and Microsoft's OLE. We summarize them in turn.

Multidatabase research can be categoried into three approaches. A classic approach [40, 41] for multi-database management relies on building a single global schema to encompass the differences among the multiple local database schemas. The mapping from each local schema to the global schema is often expressed in a common SQL-like language, such as HOSQL in the Pegasus system [1] or SQL/M in the UniSQL/M system [24]. Although the enforcement of a single global schema through data integration yields full transparency for uniform access, component databases have much restricted autonomy, scalability and their evolution becomes difficult.

The federated approach [42] improves the autonomy and the flexibility (composability) of multidatabase management by relying on multiple import schemas and the customized integration at various multidatabase levels. However, the integration of component schemas at each multidatabase level is enforced by the system. The integrated schema is static. The heterogeneity problems are resolved at the schema integration stage. This approach cannot scale well when new sources need to be added into an existing multidatabase system. Also the component schemas cannot evolve without the consent from the integrated schema.

The distributed object management approach [34, 38] generalizes the federated approach by modeling heterogeneous databases of different levels of granularity as objects in a distributed object space. It requires the definition of a common object model and a common object query language. Recent activities in the OMG [36] and the ODMG standard [10], which extends the OMG object model to the database interoperability, are the important milestones for the distributed object management.

Another approach, called the intelligent information integration (I^3) mediation [45, 48] can be seen as a generic system architecture for information integration. Several projects are currently undergoing to adapt and extend this architecture with different emphases. Examples include the Garlic project at IBM Almaden Research Center [9], which targets at developing a system and tools for the management of large quantities of heterogeneous multimedia information, the TSIMMIS [18] at Stanford, which implements the mediators-based information integration architecture through a simple object exchange model. Another example is the Context Exchange project at MIT [19, 43]. It uses context knowledge in a context mediator to explicitly define the meaning of information provided by a source and that required by a receiver. Similarly, the Intelligent Agent Integration at University of Maryland at Baltimore County uses the Knowledge Query and Manipulation Language (KQML) to integrate heterogeneous databases. The SIMS project [3, 2] uses the LOOM knowledge representation system as the data model and query language to implement the agent-based integration. Another example is the information mediation project at University of Maryland [11, 17], which integrates heterogeneous data and knowledge bases using multiple F-logic object schema, via the KIF knowledge interchange logic. Among these projects, very few have addressed all the requirements of the USECA properties in a systematic manner, which is the goal of our approach.

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) [36], 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 integration and access of multiple heterogeneous information sources. DIOM can be seen as a glue that spans and integrates these interface models at a higher level. For example, unlike in CORBA and DSOM where only specialization is explicitly supported, DIOM supports interface aggregation, interface generalization, and the import/hide meta operation for incremental interface design and construction, which not only provide flexibility and convenience in interface composition but also increase the robustness and adaptiveness of interoperation interfaces against source schema changes.

Another related field is the application of object-oriented technology to the integration of heterogeneous database systems [5, 39]. A survey by Pitoura, Bukhres, and Elmagarmid [39] presents a concrete analysis and categorization of a number of ways that object-oriented technology may influence the design and implementation of multidatabase systems. Comparing with the framework given in [39], the current DIOM development has not yet incorporated the update issues into the architecture. However, we believe that the techniques proposed in multi-level and extended transaction management [14] can easily be applied to the DIOM environment.

7 Conclusion

We have captured the requirements for integration and access of heterogeneous information sources in large-scale interoperable system as the USECA properties: *Uniform access* to heterogeneous information sources, *Scalability* to the growing number of information sources, *Evolution* and *Composability* of software and information sources, and *Autonomy* of participants, both information consumers and information producers.

To strengthen the support of USECA properties in a systematic manner, we introduced the Distributed Interoperable Object Model (DIOM) as an extension to the ODMG-93 object model by adding four adaptive interface composition meta operations and by developing an adaptive query mediation framework. Our main contributions in this paper are the following.

- First, we propose to distinguish base interfaces from compound interfaces in order to provide a semantically clean framework that promotes an adaptive approach to construction of interoperation interfaces. The concept of compound interfaces enables information consumers to concentrate more on their query requirements rather than the heterogeneity of the underlying information sources. The concept of base interfaces promotes the separation of information producers' source data models from the information consumer's domain query model. Such a separation facilitates the building of a scalable and robust query mediation framework that allows new information sources to be seamlessly incorporated into the existing query mediator.

- Second, the interface composition meta operations provided in DIOM, such as *import/hide*, *ag-*

gregation, *generalization* and *specialization*, allow incremental design and construction of new interfaces in terms of existing ones. For instance, by combining the import facility with the aggregation and generalization meta operations, 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.

- Third, DIOM promotes the deferment of semantic heterogeneity resolution to the query result assembly time instead of before or at the time of query formulation. Semantic conflict resolution is considered secondary to the explicit representation of disparate data semantics. Thus, the DIOM approach does not require any commitment to a canonical, predefined integration representation of component databases. Conflicts do not have to be resolved once for all in a static integrated schema. Instead, information consumers may define their preferred query answering formats and make changes whenever necessary. Much of the conflict resolutions are deferred to the query rsult assembly time. More importantly, we apply semantic attachment techniques to the resolution of semantic and representational heterogeneities. Thus, the heterogeneity resolution is not hard-wired in the interoperation interface schema.

We have also informally described the DIOM query processing steps and provided an example to illustrate the flavor of the DIOM query mediation framework and the role of DIOM IDL in various steps of DIOM query processing. We believe that the framework developed in this paper presents an interesting step towards enhancing the support of the USECA properties in large scale interoperable database systems.

Finally, we would like to emphasize that the DIOM approach presented in this paper proposes an adaptive information mediation framework and a collection of techniques for incremental design and construction of interoperation interfaces, rather than a new data model. This framework is targeted towards large-scale modern information systems which require the support for more flexible and scalable query interoperation across heterogeneous information sources than traditional distributed or federated database application domains. It is, of course, not expected that the proposed interace composition framework be incorporated in a given system in its entirety. Rather, the information mediation designer or the information consumers may select a useful subset of the interface composition meta operations, given a particular application domain and information gathering requirement.

Acknowledgement

This research was partially supported by NSERC grant OGP-0172859, NSERC grant STR-0181014 for the first author, and ARPA grant N00014-94-1-0845, NSF grant IRI-9510112, grants from the Hewlett-Packard Company and Tektronix for the second author. We thank the anonymous reviewers for their valuable comments and suggestions.

References

[1] R. Ahmed, P. Smedt, W. Du, W. Kent, M. Ketabchi, W. Litwin, and A. Raffi. The pegasus heterogeneous multidatabase system. *IEEE Computer Magazine*, 24(12):19–27, December 1991.

[2] Y. Arens and et al. Retrieving and integrating data from multiple information sources. *International Journal of Intelligent and Cooperative Information Systems*, 2(2):127–158, 1993.

[3] Y. Arens and C. Knoblock. Planning and reformulating queries for semantically-modeled multidatabase systems. In *Proceedings of the first International Conference on Knowledge and Information Management*, 1992.

[4] K. Bath. What is Oraperl? http://www.bf.rmit.edu.au/ orafaq/perlish.html#oraperl, 1995.

[5] E. Bertino, M. Negri, G. Pelagatti, and L. Sbattella. Applications of object-oriented technology to integration of heterogeneous database systems. *Distributed and Parallel Databases*, 2(4), 1994.

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

[7] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, M. F. Schwartz, , and D. P. Wessels. Harvest: A scalable, customizable discovery and access system. Technical Report CU-CS-732-94, Department of Computer Science, University of Colorado, March 1995. http://newbruno.cs.colorado.edu/harvest/papers.html.

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

[9] 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.

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

[11] Y. Chang, L. Raschid, and B. Dorr. Transforming queries from a relational schema to an equivalent object schema: a prototype based on f-logic. In *Proceedings of the International Symposium on Methodologies in Information Systems (ISMIS)*, 1994.

[12] U. Dayal. Query processing in a multidtatabase system. *Query Processing in Database Systems*, 1985.

[13] J. December and M. Ginsburg. *HTML & CGI Unleashed*. Sams.Net Publishing, Oct. 1995.

[14] A. Elmagarmid. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, San Mateo, Calif.), 1992.

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

[16] R. Fielding and H. Frystyk. Hypertext transfer protocol – HTTP/1.0. http://www.w3.org/pub/WWW/Protocols/HTTP1.0/draft-ietf-http-spec.html, february 1996.

[17] D. Florescu, L. Raschid, and P. Valduriez. Using heterogeneous equivalences for query rewriting in multidatabase systems. In *Proceedings of the International Conference on Cooperative Information Systems (CoopIS)*, 1995.

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

[19] 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.

[20] R. Hull and R. King. Reference architecture for the intelligent integration of information (version 1.0.1). http://isse.gmu.edu/I3_Arch/index.html, May 1995.

[21] W. Kent, R. Ahmed, J. Albert, M. Ketabchio, and M. Shan. Object identification in multidatabase systems. In *Proc. of IFIP DS-5 Working Conference on Semantics on Interoperable Systems*. Elsevier, 1993.

[22] S. Khoshafian and G. Copeland. Object identity. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, pages 406–416. ACM, 1986.

[23] G. Kiczales. Towards a new model of abstraction in software engineer. In *Proceedings of the IMSA'92 workshop on Reflection and Meta-level Architecture*, http://www.xerox.com/PARC/spl/eca/oi.html, 1992.

[24] W. Kim and et al. On resolving semantic heterogeneity in multidatabase systems. *Distributed and Parallel Databases*, 1(3), 1993.

[25] W. Kim, D. Reiner, and D. Batory. *Query Processing in Database Systems*. Springer-Verlag, 1985.

[26] Y. Lee. A prototype implementation for the diom interoperable system. In *demo at the International Forum on Research and Technology Advances in Digital Libraries (ADL'96)*, Washington D.C., May 13-15 1996.

[27] Y. Lee. Rainbow: A prototype of the diom interoperable system. MSc. Thesis, Department of Computer Science, University of Alberta, A demo version of the prototype is available at http://ugweb.cs.ualberta.ca/ diom, July, 1996.

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

[29] L. Liu and C. Pu. Customizable information gathering across heterogeneous information sources. Technical report, Department of Computer Science, University of Alberta, Dec. 1995.

[30] L. Liu and C. Pu. The distributed interoperable object model and its application to large-scale interoperable database systems. In *ACM International Conference on Information and Knowledge Management (CIKM'95)*, Baltimore, Maryland, USA, November 1995.

[31] L. Liu and C. Pu. Metadata in the interoperation of heterogeneous data sources. In *Proceedings of the First IEEE Metadata Conference*, NOAA Auditorium, Silver Spring, Maryland, April 16 - 18 1996.

[32] L. Liu, C. Pu, and Y. Lee. An adaptive approach to query mediation across heterogeneous databases. In *Proceedings of the International Conference on Coopertive Information Systems*, Brussels, June 19-21 1996.

[33] P. Maes. Concepts and experiments in computational reflection. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, Oct. 1987.

[34] F. Manola and et al. Distributed object management. *International Journal of Intelligent and Cooperative Information Systems*, 1(1), March 1992.

[35] NCSA HTTPd Development Team. The common gateway interface. http://hoohoo.ncsa.uiuc.edu/cgi/overview.html.

[36] OMG. *The Common Object Request Broker: Architecture and specification.* Object Management Group, Object Request Broker Task Force, 1993.

[37] M. Ozsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice-Hall, 1991.

[38] T. Ozsu, U. Dayal, and P. Valduriez. *Distributed Object Management.* Morgan Kaufmann, 1993.

[39] E. Pitoura, O. Bukhres, and A. Elmagarmid. Object-orientation in multidatabase systems. *ACM Computing Surveys*, 27(2), 1995.

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

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

[42] A. Sheth and J. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, Vol. 22, No.3 1990. 183-236.

[43] M. Siegel and S. Madnick. Context interchange: sharing the meaning of data. In *ACM SIGMOD RECORD on Management of Data*, pages 77–78, 20, 4 (1991).

[44] L. Wall and R. L. Schwartz. *Programming Perl.* O'Reilly and Associates, Jan. 1991.

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

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

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

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

[49] Yahoo Inc. Yahoo! homepage. http://www.yahoo.com/.

A Syntax of DIOM IDL

We here provide the portion of the DIOM IDL specification syntax that are primarily used for compound interface description.

```
       <interface_spec> ::= CREATE INTERFACE <identifier> <type_property_list>
                            [: <persistence_mode>]
                            {[<inheritance_spec>] [<generalization_spec>]
                            [<aggregation_spec>] [<import_spec>]
                            [<instance_property_spec>]
                            [<instance_operation_spec>]};
       <type_property_list> ::= ([<extent_spec>][<key_spec>])
       <extent_spec>      ::= EXTENT <identifier>
       <persistence_mode> ::= persistent | transient
       <key_spec>         ::= KEY <property_identifier_list>

       <inheritance_spec>    ::= SPECIFICATION OF <interface_list_spec>
       <generalization_spec> ::= GENERALIZATION OF <interface_list_spec>
       <aggregation_spec>    ::= AGGREGATION OF <interface_list_spec>
       <interface_list_spec> ::= <interface_id_list> |
                                 SELECT interface-name FROM InterfaceRespository
                                 WHERE  <fetch-conditions>
       <interface_id_list>   ::= <interface_identifier>
                                 | <interface_identifier><interface_id_list>
       <import_spec> ::= FROM <source_db_identifier> <import_body>
       <import_body> ::= IMPORT ALL | <import_dcl>
       <import_dcl>  ::= [IMPORT TYPE <type_identifier_list>]
                         [HIDE TYPE <type_identifier_list>]
                         [<import_type_spec>]
       <import_type_spec>  ::= <import_property_spec> | <derived_type_sepc>
       <derived_type_spec> ::= ADD TYPE <type_identifier> ISA <type_id_list>
                               <type_derivation_spec>
       <type_derivation_spec> ::= <iml_query_spec>
       <import_property_spec> ::= [HIDE <property_identifier_list>]
                                  [IMPORT <property_identifier_list>]
                                  [MODIFY <property_dcl]
                                  [ADD <property_derivation_spec>]

       <instance_property_spec> ::= <property_spec>
                                    | <property_spec>, <instance_property_spec>
       <property_spec>          ::= <attr_spec> | <rel_spec>
       <instance_operation_spec>::= <op_spec> | <op_spec>,
<instance_operation_spec>
       <attr_spec>        ::= ATTRIBUTES <domain_type><attr_name> [ READONLY]
```

```
    <domain_type>       ::= <simple_type_spec> | <struct_type> | <enum_type>
                            | <attr_collection_specifier><domain_identifier>
    <domain_identifier>::= <type_identifier> | <literal>
    <attr_collection_specifier> ::= Set | List | Bag | Array
```

B Syntax of DIOM IQL Expressions

The syntax of a DIOM IQL query expression is provided as follows:

```
<IQL-query-expression>::= [TARGET <Source-expressions>]
                          SELECT <Fetch-expressions>
                          FROM   <Fetch-target-list>
                          [WHERE  <Fetch-conditions>]
                          [GROUP BY <Fetch-expression>]
                          [ORDER BY <Fetch-expression> ]
<Source-expressions>::= <target-Info-Source name>
<Fetch-expressions> ::= <Fetch-expr> | <Fetch-expr>, <Fetch-expressions>
<Fetch-target-list> ::= <Fetch-target> | <Fetch-target>, <Fetch-target-list>
<Fetch-target>      ::= <interface type> | <target-Info-Source name> |
<Fetch-conditions>  ::= <Fetch-condition> | <Fetch-condition> <Logic-ops>
<Fetch-conditions>
<Fetch-condition>   ::= <Fetch-expr><comparison-op><Fetch-expr>
                   | <Fetch-expr><Comparison-op><const-object>
                        | <Fetch-expr><partial-match-op><const-object-set>
<Fetch-expr>       ::= <Path-expr>-><attribute> |
<Aggregate-Funs>(<Path-expr>-><attribute>)
<attribute>        ::= <consumer-defined attribute> | <source-attribute>
<Path-expr>        ::= <interface type> | <interface type>-><Path> | *
<Path>             ::= <consumer-defined attribute>-><Path> | ?
<Comparison-op>    ::= < | > | <= | >= | <> | =
<Logic-ops>        ::= AND | OR
<Aggregate-Funs>   ::= MAX | MIN | SUM | COUNT | AVG
<partial-match-op> ::= LIKE | CONTAIN
<const-object-set> ::= <const-object> | {<const-object>,<const-object-set>}
```