# The INEEL Data Integration Mediation System

Bhujanga Panchapagesan
Joshua Hui
Gio Wiederhold

{*priya,wjhui,gio*}*@db.stanford.edu*
Stanford University
Computer Science Department
Stanford, CA 94305-9020, USA.

Stephan Erickson *

*erickson@junglee.com*
Junglee Corp.
1350 Oakmead Parkway,
Sunnyvale, CA 91362, USA.

Lynn Dean
Antoinette Hempstead

{*lad,hem*}*@inel.gov*
Idaho National Engineering
& Environmental Laboratory
Idaho Falls, ID 83415, USA.

## Abstract

Large organizations often store their information in multiple separate, independently-controlled locations. Decision making becomes difficult because of the effort involved in getting an overall picture of the data. Furthermore, it is very difficult to get different groups or individuals to agree on one integrated view of the total collection of data. The INEEL Data Integration Mediation System (IDIMS) addresses the problem of integrating data retrieved from multiple heterogeneous data sources. IDIMS allows differing views across the same or different set of data sources to be created and used. IDIMS was initially applied to an INEEL environmental restoration domain and later applied to a separate State of Pennsylvania domain. This paper discusses the design and implementation of IDIMS within the INEEL environmental restoration domain.

## 1 Introduction

In large organizations, several independent databases are often used simultaneously in different departments for different purposes. Often the relevent information required for an individual or group to make necessary decisions is spread across two or more of these independent databases, each of which often has its own method for data retrieval and schema representation. Compounding the problem, many of these databases or information sources are often independently owned, making the movement of data from one source to the other or the modification of data or schema unacceptable. Traditionally, solutions to these problems have included delegating one or more individuals or groups to handle all of the information requests or creating a new local database, thus duplicating data owned by another group. Unfortunately, these methods can introduce additional delays and errors due to human-related factors and data synchronization issues.

The Environmental Restoration (ER) organization is part of a larger organization at the Idaho National Engineering and Environmental Laboratory (INEEL) called Environmental Operations. The Environmental Operations organization also includes Waste Management and a Sample Management Office. Over the last 30 years, the three Environmental Operations' organizations and their predecessors have developed more than 100 individual databases to serve individual applications. Within the last ten years, a need has developed to pull data from many of these individual databases and present the selected data as integrated information. These individual databases are contained in flat files, formal relational databases such as Oracle, FoxPro, and dBASE, as well as other proprietary formats. The INEEL Environmental Operations organization needed a method to access these many disparate data sources in a generalized fashion so that the same software could be used for more than one specific combination of data sources.

The INEEL Data Integration Mediation System (IDIMS) was built to address the data integration issues of a specific ER domain at the INEEL. IDIMS was designed and implemented as a collaborative effort among the INEEL, Stanford University, and ISX Corporation. IDIMS provides a method of preserving a group and/or individual's knowledge about how to access and integrate data for a variety of domains. This domain knowledge includes the definition of the domain's integrated view, the specification of how the data sources fit to this view, the knowledge of how to integrate data across the different data sources, and the knowledge required to retrieve data from these sources. IDIMS provides the group with the benefits of consistent application of domain knowledge and the reduction of unnecessary data duplication. IDIMS was designed to accept the domain knowledge as input into the system so the same software can be utilized by many different domains.

The INEEL ER problem scenario provided the proto-

*This work was done while the author was at ISX Corp., 4353 Park Terrace Drive, Westlake Village, CA 94086, USA.

type domain for the initial version of IDIMS. It involved integrating data spread across two types of structured databases, Oracle and FoxPro. Due to the fact that most large organization use structured databases for the high performance and comprehensive query support these databases offer, the implemented IDIMS assumes that the data is organized entirely in structured form. Although this is currently the underlying assumption, the IDIMS system architecture was designed to handle a wide range of data source types. To access and integrate data retrieved from non-structured data sources, changes would be needed at the data-access level, but the overall system architecture should remain the same. Figure 1 illustrates the IDIMS system architecture. A similar framework is also used in other mediation systems, such as DISCO[2], GARLIC[3], TSIMMIS[4] and HERMES[5].

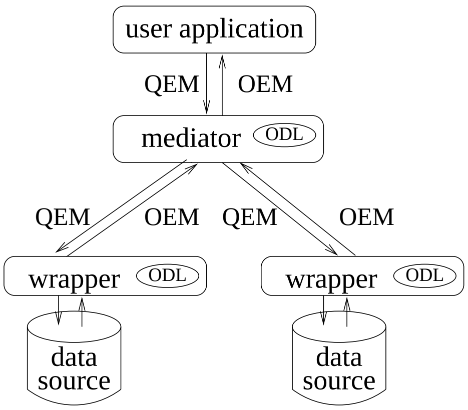



Figure 1: IDIMS System Architecture

There are three subsystems in IDIMS: the user application, the mediator, and the wrapper. As Figure 1 shows, a mediator serves as a middle-layer which provides data access and data integration to a user application so that the user application does not need to distinguish the differences among the data sources. Instead, the user application perceives a central object-oriented database provided by the mediator. When a mediator receives a user query, it decomposes the query into sub-queries (if necessary) and forwards the sub-queries to the correct wrapper(s). A wrapper provides the mapping from the mediator's integrated view to its specific data source view. A wrapper receives queries from a mediator and translates the queries into the source-specific query language and terminology. The query results are returned from the wrapper(s) to the mediator. The mediator then integrates all the results and returns a single response to the user application.

Even though IDIMS was initially built to address the problem in the INEEL ER domain, the system was designed to be domain-independent. In other words, it was designed so that a variety of domains could utilize IDIMS' data integration capabilities as long as the appropriate domain knowledge is provided.

In order to provide domain independence, extensibility, and flexibility, the following four elements are critical to IDIMS:

1. *Common Domain Specification*:
   An extended version of the ODMG (Object Database Management Group) ODL (Object Definition Language) [6] is used for both the mediator and the wrapper subsystems to describe their specific views of the domain. This method of domain specification allows the domain knowledge to become a dynamic input into IDIMS.

2. *Common Service Interface*:
   Every wrapper and mediator subsystem shares a common service interface. The most commonly used services include accepting a query, returning data results, and providing schema-related information. This common service interface allows for the dynamic extension of the number of available wrappers and/or addition of vertical mediator layers to the system without modifying the system architecture or software.

3. *Common Query Representation*:
   A new query structure, Query Exchange Model (QEM), was defined for IDIMS to provide a common query representation used by each of the IDIMS subsystems.

4. *Common Data Representation*:
   The Object Exchange Model (OEM) [7] was adopted as the common data representation for IDIMS. OEM is simple and flexible, facilitating data integration across multiple data sources.

In the body of this paper, we provide details about the mediator and wrapper components, along with their supporting Semantic Model libraries and their QEM and OEM structures. We will not address the user application component beyond describing how it interfaces with the mediator component. In the conclusion of this paper, we provide an assessment of the whole system and discuss some related work that has been done in this area.

## 2 ODL and the Semantic Model

In order for an individual or group's domain knowledge to be an input into IDIMS, there needed to be a method of specifying the domain knowledge to the mediation system. ODMG's Object Definition Language (ODL) was extended to provide a highly dynamic method for domain definitions to IDIMS. It is important to note that each mediator and wrapper subsystem within IDIMS must have its own ODL specification. This is necessary because each wrapper has its domain focused on one specific view of one data source, while the mediator's domain is the integrated view of all its underlying wrappers. The mediator's ODL describes the domain that will be available to the user application subsystem.

Once a subsystem's domain has been described in terms of ODL, the subsystem can access its specific ODL specification to create its internal domain representation. A subsystem's internal domain representation is referred to as its *Semantic Model*.

## 2.1 Object Definition Language (ODL)

ODMG originally conceived of ODL in order to provide interoperability between object-oriented databases (OODBs) by enabling the specification of classes of objects. Since IDIMS was designed to utilize virtual OODBs within its mediator and wrapper subsystems, ODL provided an ideal way of describing the objects to be exchanged between the IDIMS subsystems.

ODL provides a mechanism to declare a class of objects. Each of these classes will have a set of attributes, functions, and/or relationships. Attributes represent the data of a class; relationships represent how one or more classes relate to each other; and functions provide a means for additional data abstraction, processing, and manipulation.

IDIMS required some extensions to ODMG ODL. These extensions include the following:

1. Relationship expressions to describe how two classes are related;
2. Data-source mapping specifications (in wrapper subsystems only); and
3. Source-specific data transformations (in wrapper subsystems only).

Each IDIMS mediator and wrapper subsystem must have facilities to understand the objects and attributes supported for its given domain. This knowledge is specified in each subsystem's specific ODL. The mediator subsystem ODL is referred to as *Mediator ODL (MODL)* and the wrapper subsystem ODL is referred to as *Wrapper ODL (WODL)*. The MODL specification provides the integrated domain view. The main purpose of the WODL is to specify mappings from the integrated domain view to specific fields within its associated data source. Moreover, we provide a simple interpreted language to allow data field transformations, such as enumerated data type conversions, unit conversions and field concatenation.

Using ODL for domain definitions is the key to the IDIMS' solution for heterogenous data access and integration issues for a variety of domains. To use IDIMS for a completely new domain, only new ODL files are needed. The underlying code does not change; only the knowledge about the specific domain changes. The ODL specification also provides the capability to dynamically alter knowledge within a domain. An ODL specification can be changed and the next time the subsystem is activated, the new ODL specification is used.

Figure 2 provides a portion of the MODL specification used in the INEEL ER domain. Figures 3 and 4 provide excerpts from the WODL specifications for the ER data sources. In addition to providing an example of IDIMS ODL, these ODL specifications are used by the examples throughout the rest of this paper.

```
interface SoilSample {key SampleID}
 {
  Attribute  String  SampleID;
  Attribute  String  Description;
  Attribute  String  Location;
  Attribute  Float   Depth;
  Attribute  String  LabID;
  Relationship Set<LabResult> lab_results inverse
       LabResult::for_sample
   {
    SoilSample.SampleID = LabResult.SampleID
   };
  Relationship Laboratory at_lab inverse
       Laboratory::samples
   {
    SoilSample.LabID = Laboratory.LabID
   };
 };

interface LabResult {key ResultID}
 {
  Attribute  String  ResultID;
  Attribute  String  SampleID;
  Attribute  String  Contaminant;
  Attribute  Float   Concentration;
  Relationship SoilSample for_sample inverse
       SoilSample::lab_results
   {
    LabResult.SampleID = SoilSample.SampleID
   };
 };

interface Laboratory {key LabID}
 {
  Attribute  String  LabID;
  Attribute  String  LabName;
  Attribute  String  LabAddress;
  Attribute  String  LabCity;
  Attribute  String  LabState;
  Relationship Set<SoilSample> samples
       inverse SoilSample::at_lab
   {
    Laboratory.LabID = SoilSample.LabID
   };
 };
```

Figure 2: INEEL ER MODL Example

## 2.2 Semantic Model

The semantic model is the internal representation of a mediator or wrapper subsystem's collective domain-specific knowledge. It is implemented by a common set of libraries used by each IDIMS subsystem. A subsystem's Semantic Model is dynamically created at run-time by collecting all domain-related knowledge specifications that pertain to that subsystem.

Each IDIMS wrapper and mediator subsystem has its own Semantic Model. The mediator's Model represents the integrated domain knowledge along with its mappings to each of its underlying wrappers. Each wrapper's Model represents the domain knowledge which is specific to that wrapper's particular view of its data source. Although

```
join soils_table(id) to samples_table(ssid);

interface SoilSample {key SampleID}
 {
  Attribute  String  SampleID {soils_table.id};
  Attribute  String  Description {soils_table.desc};
  Attribute  String  Location {samples_table.loc};
  Attribute  Float   Depth {soils_table.depth};
 };

interface LabResult {key ResultID}
 {
  Attribute  String  ResultID {result_table.id};
  Attribute  String  Contaminant
                       {result_table.contam};
  Attribute  String  Concentration
                       {result_table.conc};
 };

interface Laboratory {key LabID}
 {
  Attribute  String  LabID {lab_table.id};
  Attribute  String  LabName {lab_table.name};
  Attribute  String  LabAddress {lab_table.street};
  Attribute  String  LabCity {lab_table.city};
  Attribute  String  LabState {lab_table.state};
 };
```

Figure 3: INEEL ER Data Source 1 WODL Example

```
interface SoilSample {key SampleID}
 {
  Attribute  String  SampleID {ss.sample_id};
  Attribute  String  Description {ss.text_desc};
  Attribute  String  Location {ss.place};
  Attribute  String  LabID {ss.resp_lab};
 };

interface LabResult {key ResultID}
 {
  Attribute  String  ResultID {lr.result_id};
  Attribute  String  SampleID {lr.sample_id};
  Attribute  String  Concentration {lr.amount};
 };

interface Laboratory {key LabID}
 {
  Attribute  String  LabID {lab.org_id};
  Attribute  String  LabName {lab.name};
 };
```

Figure 4: INEEL ER Data Source 2 WODL Example

the mediator and wrapper subsystems have differing information in their respective Semantic Models, the general structure and interface of the Semantic Models is the same. This is important because it allows the various subsystems to communicate with each other about the knowledge within their Semantic Models. This commonality also provides support for incorporation of additional wrappers and for layering additional mediators to the mediation system.

The ODL defined for a particular subsystem is a major input item of the subsystem's Semantic Model. However, a subsystem's Semantic Model will contain other knowledge not contained in its ODL file. For example, a mediator's Semantic Model will also have mappings to the appropriate wrapper(s) for each of the attributes defined in the ODL. These mappings are added at run-time. After the mediator has populated its Semantic Model with its MODL contents, it then queries its associated wrappers for their Semantic Models. The mediator subsystem then can add the wrapper mappings to its Semantic Model.

The additional knowledge stored in the mediator's Semantic Model allows the mediator to correctly route user queries to the appropriate wrapper(s), while supporting a very dynamic environment. For instance, if a wrapper is added/deleted or if the contents of a wrapper's ODL file change, the mediator will automatically handle the change since it does not get the wrapper Semantic Model knowledge until run-time. When a user application queries a mediator about its domain, the mediator returns its current Semantic Model knowledge to the user application.

Figure 5 illustrates the contents of a mediator Semantic Model. This figure graphically illustrates a portion of the INEEL ER domain view. The INEEL ER domain view defined to IDIMS contains 39 objects, with 141 attributes spread among these objects. These and objects and their attributes have been mapped across three separate data sources. The underlying data sources provide information about approximately 26,000 Samples and 373,000 Lab Results.

## 3 Query Representation

In order for a user application to issue requests to a mediator and for the mediator to then issue requests to its wrapper(s), there must be a method of specifying a query. We identified the following requirements for an acceptable query representation method for IDIMS:

1. The representation should provide a reasonable subset of a variety of existing query languages' expressive powers, not a superset, in a simple and concise manner.

2. The representation should be in an internal structure rather than an actual language to alleviate the need for multiple language/structure translations within each of the IDIMS subsystems.

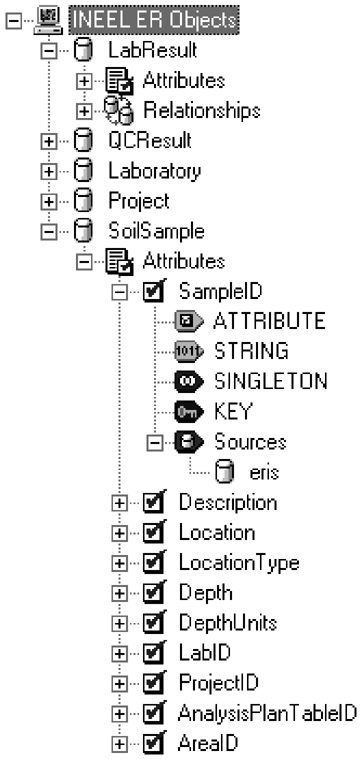



Figure 5: INEEL ER Mediator Semantic Model

3. Query constructs should be defined independently from how data is organized and/or represented in the underlying data sources.

4. The representation must be lightweight and extensible in order to accommodate rapidly-changing data sources, both in terms of content and schema.

5. The query representation must be clearly defined.

During the search for a query representation that met these requirements, a subset of the available query representations (e.g. SQL, ODMG OQL[6], Lorel[8], KQML[9], MSL[4] and Datalog) were considered. None of the considered query representations met all the IDIMS requirements. Therefore, a new query representation was developed specifically for IDIMS. The IDIMS query representation was named the *Query Exchange Model (QEM)* due to its high-level nature and its symmetry with the Object Exchange Model (see Section 4). QEM provides the following benefits to IDIMS:

1. QEM focuses on the commonalities that exist between a few of the more popular query languages used in academic-, commercial- and government-related applications, such as SQL, OQL, and Lorel.

2. QEM is translatable to simple queries for a wide array of alternate query representations due to its structural representation rather than language specification.

3. QEM is closely related to the Object Exchange Model (OEM), which is used for returning query results. This similarity provides for a consistent representation format for sending queries and receiving results throughout IDIMS.

QEM is represented in tree form and is similar to a parse structure. A QEM tree is composed of QEM nodes, each of which has an inherent node type. Six different QEM node types are defined for IDIMS. The QEM node types are described below and illustrated in Figure 6.

*Class Node*: Main class or concept of a query.

*Relationship Node*: Subquery representing a relationship between one class and another.

*Attribute Node*: Inherent property of a given class.

*Literal Node*: Explicit value

*Operator Node*: Operator or Primitive Function.

*Function Node*: Domain-specific Function.

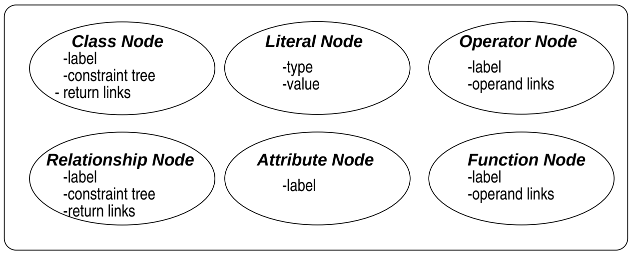


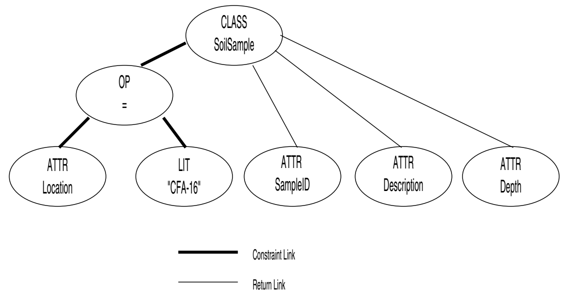

Figure 6: QEM Node Types and Structural Characteristics

The simplest conceptual representation of a query involves naming one class (i.e., the parent class), using an expression to constrain this class, as well as providing a list of attributes to be returned. If the constraint and returned attributes are all from the parent class, this query forms the basic unit of query exchange and is referred to as a primitive query.

A primitive QEM query is divided into three parts: a root Class node, a constraints section, and a return section. The root Class node of a query contains information about which class is to be retrieved. The constraints section is an expression representing how to constrain the root class. The return section represents information about what data needs to be returned for the root class. It is important to note that any item (e.g., class, attribute, relationship) referenced in a QEM node must be defined in the associated mediator's ODL file. Figure 7 illustrates a primitive query in both graphical and QEM node representations.

Although primitive queries provide reasonable expressive power for many applications, primitive queries lack the capability to constrain or return results based on

Query: Return the ID, description, and depth for all soil samples taken from location "CFA-16"

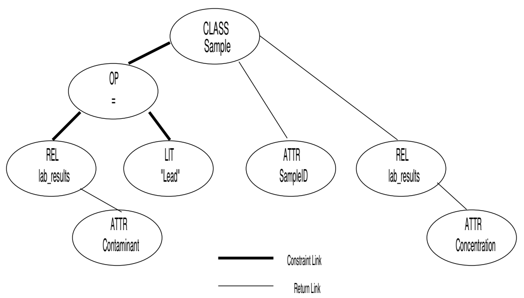



```
root:  <Class  SoilSample  &c1  {&r1  &r2  &r3}>
c1  :    <Operator  =  {&op1  &op2}>
op1 :      <Attribute Location>
op2 :      <Literal String "CFA-16">
r1  :    <Attribute  SampleID>
r2  :    <Attribute  Description>
r3  :    <Attribute  Depth>
```

Figure 7: Primitive Query Example

associations between attributes from different classes. Queries that provide the capability to associate classes and their attributes are called composite queries. QEM expresses composite queries through the use of a Relationship node. Relationship nodes can be utilized on the constraint section and/or the return section of the root Class node (or other Relationship nodes). Figure 8 illustrates a composite query.

Query: Return the soil sample ID and its associated lab results' concentration values for soil samples which have at least one lab result with the contaminant name of Lead.




```
root:  <Class  SoilSample  &c1  {&r1  &r2}>
c1  :    <Operator  = {&op1  &op2}>
op1 :      <Relationship  lab_results    Ø  {&r1a}>
r1a :        <Attribute  Contaminant>
op2 :      <Literal  String "Lead">
r1  :    <Attribute  SampleID>
r2  :    <Relationship  lab_results   Ø  {&r2a}>
r2a :      <Attribute  Concentration>
```

Figure 8: Composite Query Example

Data may be stored in a variety of forms, units, and measures. Furthermore, the form of data requested by the user or application is not always accessible at the level of the data source. Therefore, it is necessary to allow the use of functions or predicates within a query. The QEM Function node allows for queries involving domain-specific functions. Although the current implementation of IDIMS supports only literal arguments to its QEM functions, domain-specific functions should behave equally well with arguments acting on individual attribute values, as well as arguments acting on aggregate values.

## 4 Object Exchange Model (OEM)

In order for mediators and wrappers to exchange data, they need to use a common data representation. The structure of this data can vary among queries as well as among sources. For example, the data results from a relational database would be different from the data results from an object-oriented database. Due to these differences, the following requirements were defined for the data representation to be used throughout IDIMS:

1. The representation must be flexible enough to handle data from different types of sources.
2. The representation must be capable of adapting to a dynamic environment where data sources can be added or removed at any time.
3. The representation must be independent of how data is stored in the underlying data sources.
4. The representation must be capable of representing objects which may or may not have data available for every attribute in every instance.
5. The representation must facilitate data integration.

The Object Exchange Model (OEM) is a self-describing model developed by the TSIMMIS project at Stanford University. As the name implies, the purpose of OEM is to provide a method to exchange objects. OEM not only contains the data but also its own schema, which can facilitate data integration from disparate data sources.

OEM represents data results via a collection of OEM node objects. An OEM node consists of four fields: object-id, label, type, and value. It is often represented as follows:

```
<object-id label type value>
```

The *object-id* is used to uniquely identify a specific OEM node. It can be either a memory location or an identifier. The *label* is a string which describes what the OEM node represents. The *type* is the data type for the OEM node's value field, which can be either a scalar data type (e.g., string, float, integer) or a collection type (e.g., set, list, bag). The *value* is either a scalar data content value or a reference to a collection of OEM nodes, depending on the specified type.

Figure 9 provides an example of data results represented using OEM. These results are associated with the query shown in Figure 8.

```
<&oid1  SoilSample  Set  {&oid11  &oid12}>
  <&oid11  SampleID  String  "4040040041">
  <&oid12  lab_results  Set  {&oid121  &oid122}>
    <&oid121  LabResult  Set  {&oid1211}>
      <&oid1211  Concentration  Float  51.3>
    <&oid122  LabResult  Set  {&oid1221}>
      <&oid1221  Concentration  Float  75.2>
<&oid2  SoilSample  Set  {&oid21  &oid22}>
  <&oid21  SampleID  String  "X911OS5687">
  <&oid22  lab_results  Set  {&oid221}>
    <&oid221  LabResult  Set  {&oid2211}>
      <&oid2211  Concentration  Float  4.8>
```

Figure 9: OEM Example

## 5 Mediator

The mediator provides users integrated access to multiple heterogeneous data sources. Users formulate queries according to the mediator domain model (MODL). Since the MODL describes the global view of the combined schemas of all sources, any single attribute inside a query can be spread across multiple sources.

In this system, the Mediator is composed by three components, and each of the components have a very specific job.

The relationship among these components can be shown in figure 10 and the description is as follows.

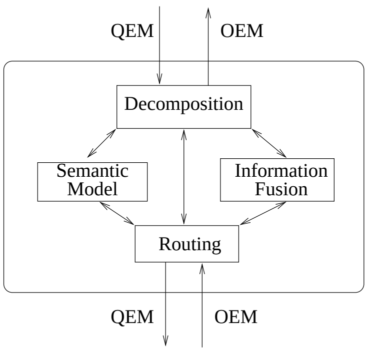


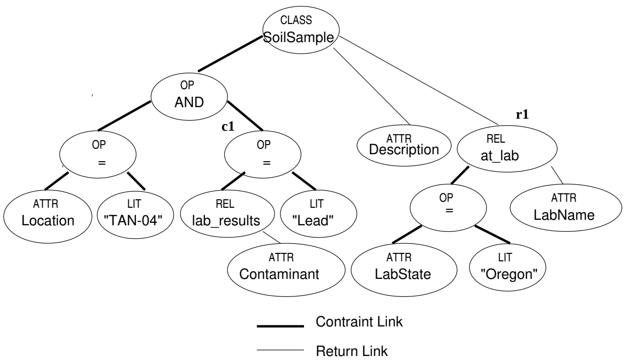

Figure 10: The mediator and the interaction between its components

1. **Decomposition Component:** examines the query and decomposes it into several sub queries according to its structure.

2. **Routing Component:** sends the queries to the correct data sources and retrieves the results.

3. **Fusion Component:** groups the data together, removes redundancy, and resolves inconsistencies.

The Routing, as well as the Decomposition component make use of the Semantic Model. As it was mentioned previously, the semantic model at the mediator level is used to get source specific information related to the underlying schema. To illustrate the purpose of each component, we will use the following example.




```
root: <Class SoilSample &c1 {&r1 &r2}>
c1:     <Operator AND {&op1 &op2}>
op1:      <Operator = {&op11 &op12}>
op11:       <Attribute Location>
op12:       <Literal String "TAN-04">
op2:      <Operator = {&op21 &op22}>
op21:       <Relationship lab_results NULL &r3>
r3:           <Attribute Contaminant>
op22:       <Literal String "Lead">
r1:     <Attribute Description>
r2:     <Relationship at_lab &c2 &r4}
c2:       <Operator = {&op3 &op4}>
op3:        <Attribute LabState>
op4:        <Literal String "Oregon">
r4:       <Attribute LabName>
```

Figure 11: Query Q

### 5.1 Decomposition

As defined in Section 3, a composite query is a query which has at least one relationship node, either on the constraint side or on the return side (See Figure 11). The main task of the Decomposition component is to transform any composite query into several primitive queries. To accomplish that, we make use of the expression associated with the relationship attribute because it represents the way of relating one class to another.

The decomposition component creates a sub-query from the relationship node and then replaces the relationship node by attributes used in the expression and the result of the sub-query.

Consider the composite query $Q$ (figure 11) which has relationship node on both the constraint and the return side.

On the constraint side, we first take the relationship node and form a sub-query. Then the results of the sub-query are extracted to form a new sub-tree replacing the original relationship node. The process is illustrated by figure 12.

The above process can also be thought as a semi-join between the object represented by the relationship node and the object in the query node.

Now we look at the return side of the query $Q$. First, the relationship node, *r1*, is replaced by the attributes

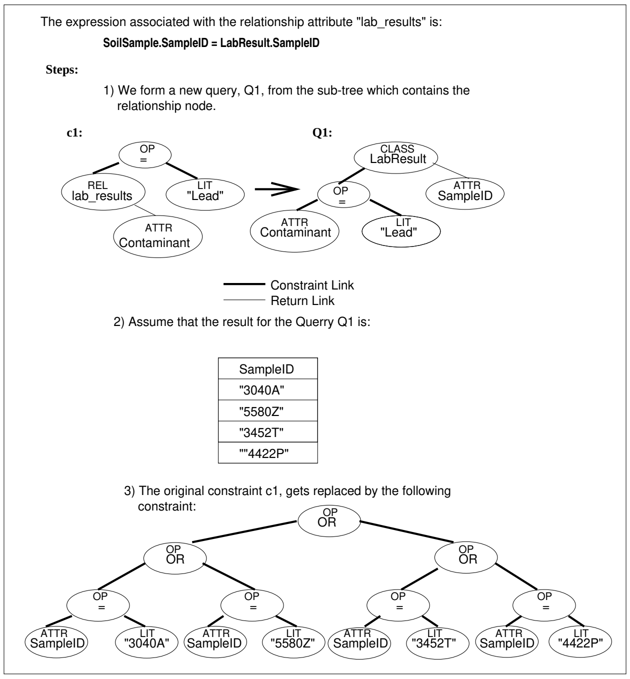



Figure 12: Decompose a composite sub-tree in the constraint side

of the query object from the expression associated with that relationship. (figure 13)

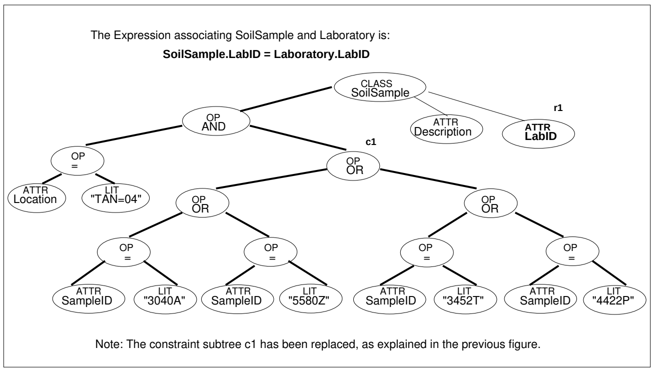



Figure 13: The new query Q'

Then the results of *Q'* become a new constraint for the relationship node and form a subquery *Q2*. (See Figure 14)

We will use a method provided by the fusion component to merge the two results based on the values of *Lab_id*.

After these steps, the original query has been decomposed into several primitive queries (*Q1*, *Q'* and *Q2*). One query may depend on the result of the other one (e.g. *Q'* uses the result of *Q1* to form itself) which implies an ordering of execution. To get the result, each primitive query will be passed to the Routing component.

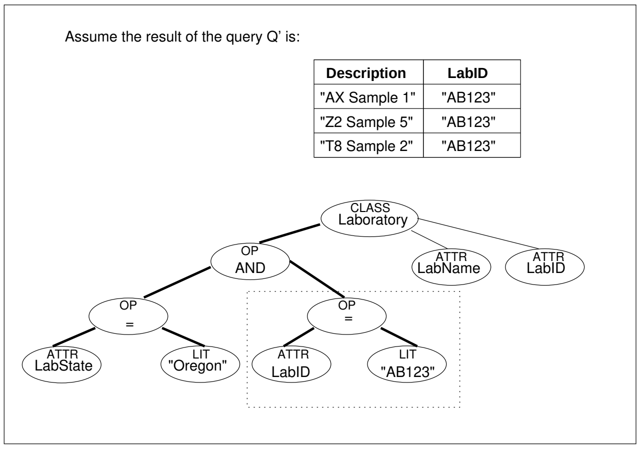



Figure 14: The new subquery Q2

## 5.2 Routing

At the beginning of the routing, the attributes involved in the query all belong to a single class. However, they may be distributed among several data sources. The Routing component retrieves all the data by routing the query to the correct data sources.

We make the assumption here that partial data pertaining to objects from different sources can be correlated if their key attributes have the same values. Furthermore, this implies that all keys of a given object must be available from all sources which export that object.

Let us illustrate the use of Routing by continuing our example. The Decomposition component decomposes the query *Q* into three queries : *Q1*, *Q'* and *Q2*. Each is sent to the Routing component one after another.

Consider the query *Q1* (figure 15) with attribute source information:

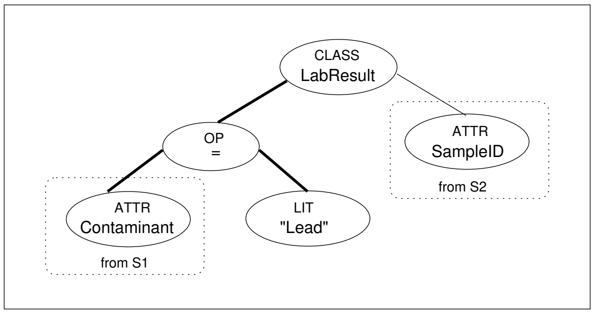



Figure 15: Query Q1 with source information

The return attribute is not from the same data source as the constraint attribute. In this case, we cannot directly use *Contaminant="Lead"* to get the corresponding *Sample_IDs*. We first obtain all the objects which satisfy the constraint from source *S1* along with their key attributes, and then use the obtained key values to retrieve the values of *Sample_ID* from source *S2*.

Consider the query *Q'* (figure 16) :

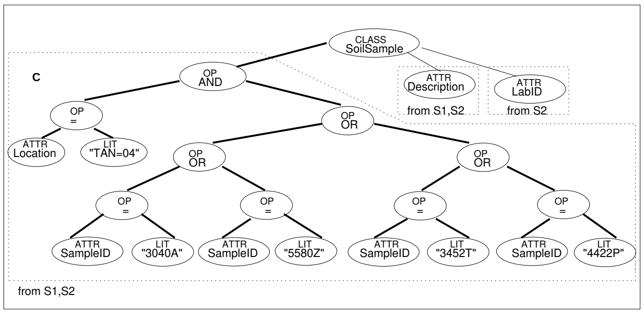



Figure 16: Query Q' with source information

All the attributes in the constraint section are from the same sources. However, not all return attributes are from the same sources. In this case two modified queries (figure 17) will be sent to each source. They both have the same constraint section but the return section is different. Since *SampleID* is the key, we will group the results together according to it's value.

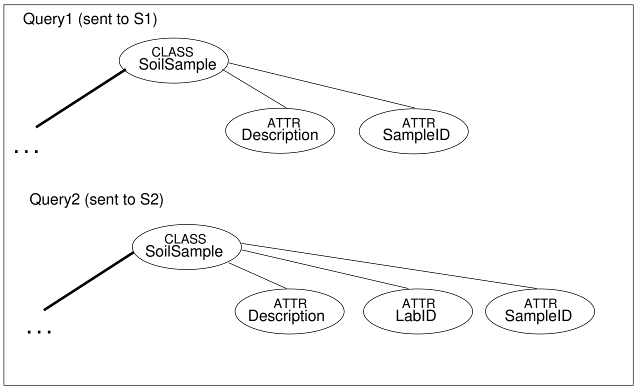



Figure 17: Two new queries

Consider the query *Q2* (figure 18) :

Since the attributes on the left side and right side of the operator node *OP'* are not from the same sources, we construct a query for each side requesting the key attribute, *LabID*. From these results, we find the common set of key values (due to the fact that the operator is "AND") and then use them to retrieve the return values of the original query *Q2*.

Using keys is one of ways to correlate data from different sources. Once we have more knowledge or understanding about the data, we can do more than solely rely on the key.

## 5.3 Information Fusion

The Information Fusion component contains two operations to integrate data. The first one is the Union operation. It uses the same assumption about key attributes that the routing component does. It groups data objects

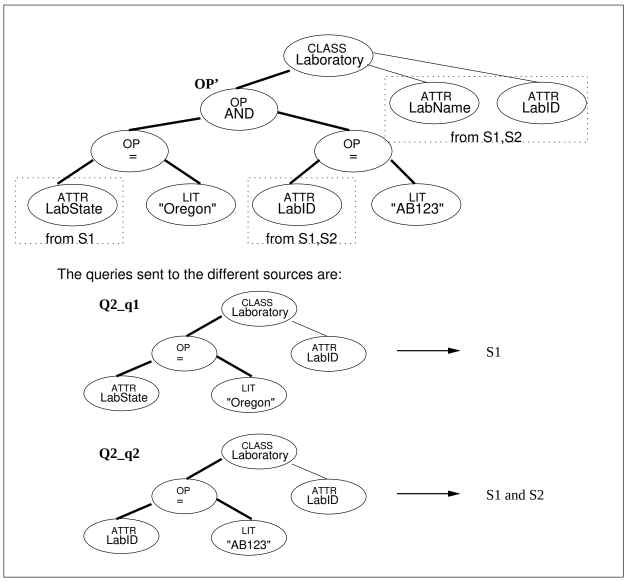



Figure 18: Query Q2 with source information and source dependent queries

together if they belong to the same class and have the same key values. At the same time, duplicated attributes are eliminated as shown in figure 19.


Figure 19: Union operation example

However, we may encounter inconsistency where objects have the same key values but conflicting values for their other attributes. In this case, both conflicting attribute values will be returned, each value labelled with its data sources of origin.

In general, the Union operation can be thought like the application both the join and union relational algebra operations together. It is mainly used by the routing component in order to group data together that is from the same class, but from different sources.

The second operation is Merge, which is used by the

Decomposition component to assemble the results of two primitive queries (figures 13 and 14) where one is from the root node of the query and the other is from the subquery formed by one of relationship nodes in the return side. The Merge operation evaluates the relationship expression for every pair of objects from the two results sets, and assembles those objects satisfying the expression. This operation is shown in figure 20.

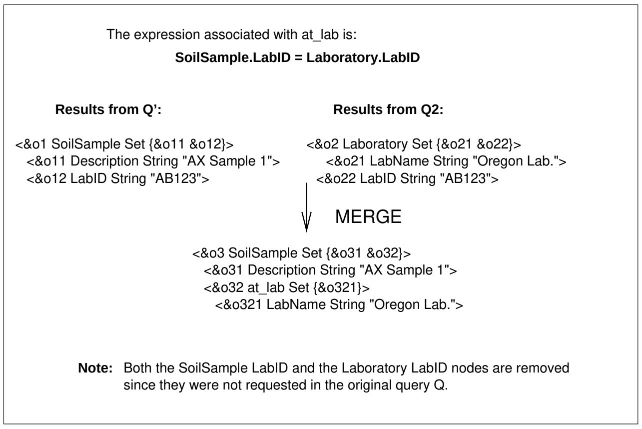



Figure 20: Merge operation example

## 6 Wrapper

The wrapper component represents access to a single data source. It receives source independent queries, translates them to source specific queries, accesses the data source, and translates source specific return values into source independent return values. As shown in figure 21, the wrapper has four components: the QEM To SQL component, the Table To OEM component, the Semantic Model component, the SQL Access component, and the Communication Transport Component.

Both QEM to SQL and Table to OEM belong to a larger class of translation components, whose purpose are to translate from QEM to a source specific query language, and to translate from a source specific data representation to OEM. Similar components are used to translate queries from QEM to a subset of Lorel, MSL, and MQL as well as from OEM to SQL Tables and HTML.

Though the individual translation components might internally have different architectures and use different translation algorithms, all rely on a local Semantic Model component to obtain runtime domain specific information such as class terminology, attribute names, and table mappings. Varying levels of capabilities can be supported at the level of the source depending on the extent of translation involved.

The SQL Access component, is responsible for direct access to the data source. Though all sources in IDIMS are currently relational, other types of sources which support alternate forms of query access can easily be integrated into the system. Defining whether a data source can or cannot be integrated into IDIMS is equivalent to defining whether the queries of the data source can be translated into or from primitive QEM queries. The range of queries supported at the level of sources range from full blown SQL to predicate matches in the form 'value = x'.

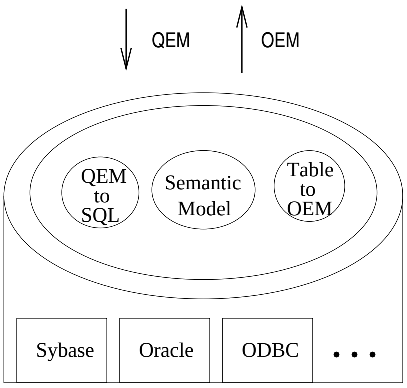



Figure 21: Generic Wrapper Architecture

## 7 Assessment and Conclusions

The mediation system described in this paper, IDIMS, provides a method to preserve an individual and/or group's knowledge and memory about retrieving and integrating data residing in multiple, heterogeneous data sources. IDIMS also provides a generalized information integration system. The initial prototype software was developed to integrate data retrieved from multiple sources defined within an INEEL environmental restoration domain. The same software has been successfully used to implement a data integration mediation system for a domain within a State of Pennsylvania organization that issues and tracks permits provided to clients responsible for performing clean-up activities at sites located in Pennsylvania. The key to this reuse of the same software components within this new domain was through the use of a new domain model described via the Object Definition Language (ODL) to the mediator and wrappers. No modifications were required in any of the software components.

In addition, the original version of IDIMS accessed and integrated data divided between only two separate sources. One source was an Oracle database and the second source was a FoxPro database. However, by the time the initial version of IDIMS was implemented in the end-users' environment, the FoxPro database had been reorganized into two separate source directories. The mediation system was able to handle this data reorganization seamlessly by treating the two directories as separate sources. Simply by creating a wrapper ODL file for each FoxPro directory and defining three wrappers (two FoxPro and one Oracle) to the mediator, the system was able to handle this data source reorganization without any software modifications. Through the use of these ODL files, the mediation system semantic model becomes truly dynamic. No recompilations of any software components are required. Changes in the domain model or data source mappings are immediately reflected the next time the mediation system is activated.

## 8 Future Work

This mediation system has been successfully used in two separate domains, additional capabilities are planned for inclusion in future versions of the system. Follow-on work is being conducted to enhance the capability of the mediation system to capture and utilize additional *human knowledge* about the data, its sources, and its integration process. An expert system tool will be used to encode this additional knowledge into a computer-useable form. This will allow the mediator to preserve and utilize information and data not currently stored in the data sources themselves. Additional rules for integrating data will also be provided. Other capabilities which have been identified for inclusion are listed below:

1. Designation of a primary and secondary source for attributes based upon data content.

2. Redefinition of the primary source for each attribute when some specified condition(s) are met.

3. Additional data abstraction and aggregation, such as the ability to provide an average of the contents of a specified attribute when the individual data occurrences are spread across multiple sources. (The wrapper for each separate source could not compute the average because each wrapper does not have access to the entire set of occurrences.)

4. Ability to reuse wrappers with multiple mediators that have varying domain terminologies by incorporating a domain translation capability. This domain translation will allow the reuse of wrappers that may use different terminology from the mediator terminology, as long as the wrapper contains a proper subset of the mediator domain model.

5. Provision for an object in the mediator's domain model to have attributes that map to attributes from multiple objects within one associated wrapper.

## 9 Acknowledgements

We would like to thank Nancy Lehrer and Charles Channell for their guidance and many suggestions in the course of the project. We would also like to thank Catherine Tornabene for initial proof-reading and corrections.

## References

[1] G. Wiederhold, "Mediators in the Architecture of Future Information Systems", *IEEE Computer*, March 1992, pp. 38-39.

[2] A. Tomasic, L. Raschid, and P. Valduriez, "Scaling Heterogeneous Databases and the Design of Disco", *Proc. of the 16th ICDCS*, 1996, pp. 449-457.

[3] M. Carey et al., "Towards Heterogeneous Multimedia Information Systems: The Garlic Approach", *Proc. of the Intl. Workshop on Research Issues in Data Engineering*, March 1995, pp. 124-131.

[4] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman, "Medmaker: A mediation system based on declarative specifications", *IEEE Conference on Data Engineering*, 1996

[5] V. Subrahmanian et al., "HERMES: A Heterogeneous Reasoning and Mediator System", *http://www.cs.umd.edu/projects/hermes/overview /paper.*

[6] R. Cattell et al., *The Object Database Standard - ODMG 93*, Morgan Kaufmann, 1993.

[7] Y. Papakonstantinou et al., "Object exchange across heterogeneous information sources", *IEEE Conference on Data Engineering*, 1995.

[8] S. Abileboul, D. Quass, J. McHugh, J. Widom and J. Wiener, "The Lorel query language for semistructured data", *Journal of Digital Libraries*, November 1996.

[9] T. Finin et al., "Specification of the KQML Agent-Communication Language", *http://www.cs.umbc.edu/kqml/papers.*