ERISII Initial Design Document

I. Design Goals

The goals behind the ERISII system are to present to the user a uniform method for data access and retrieval. This system should be platform independent, be capable of accessing both structured and semi-structured data, and provide for a wide span of query capabilities. In addition it should be configurable, easy to maintain, and flexible enough to dynamically incorporate new data sources as soon as they are available. Though most of decisions in the initial design have been made in light of the INEL requirements, we have also been looking closely at what have been the directions of I*3 technology in general. In the first half of this document, we will examine the particular requirements for the INEL, as well as current I*3 technology trends. In the second half we will layout the initial ERIS II design, including the functionality of the main components of the system, as well as the division of tasks among the parties involved, and an approximate schedule based on what work there is to be completed.

II. ERISII Functional Requirements

Based on an initial assessment, the ISX/Stanford team has determined a number of requirements for the ERISII System. These requirements are based on the assumptions thee primary prototype will be used by environmental restoration managers, while being flexible enough to eventually serve the needs of those involved in report generation, lab analysis, and management functions. The main perceived requirements are thus:

A) Uniform access to ERIS and IEDMS information.

- ERISII should be a black box which can answer to queries for either ERIS or EIDMS (both involve structured data)
- The user should not have to specify which source will satisfy a query. .
- The user should have the option to specify source preferences.

B) User Centered Model

- The ERISII system will be based around an object oriented model of the environmental restoration domain which reflects how environmental restoration managers think about thier tasks.

C) Multiple sources with overlapping data

- Much information at the INEL is redundantly stored in multiple sources. The ERISII system will present a single view of a given object, even if some attributes are stored in one source and others are stored in another source, or if data is stored redundantly in in multiple sources.

- The system should support a variable number of sources. The initial implementation will support only structured relational data but will be designed to support heterogeneous types of data and data storage systems.

D) Relating/Associating information across multiple sources

- Presenting a view of data across multiple sources will require that there be an understanding of the relationships between the various objects, or data entities supported by this view. Some of these relationships must be configured by administrator, and others will be directly inferred by the system.

E) Support for simple and complex data types
- numbers
- strings
- dates
- GIS types

F) Simple and Flexible Query Language

- The query language must support query constraints of the following types:
  - constraints by attribute value
    - Find all samples with id = "1234"
  - constraints by related object attributes
    - Find all samples belonging to SAP whose id = "5678"
  - constraints by domain concepts
    - Find all sample results for volatile compounds of a sample
  - standard and domain specific operators and functions
    - Find all samples with constraints lab-entry-date>Jan 14th and lab-entry-date<Feb 20th
    - Find all samples with concentration("Fluoride", 133.2) < 0.23

III. Overall Structure and Functionality

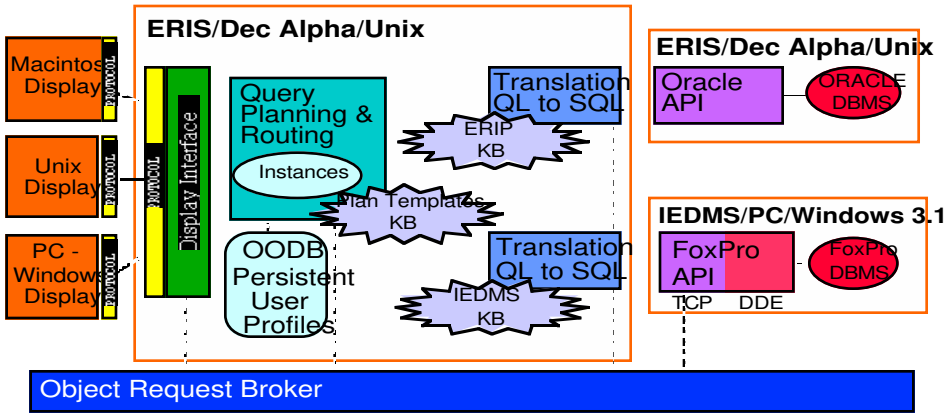


Figure 1 ERISII System Architecture

Figure 1 illustrates the ERISII system. An ERISII interface will generate high level queries to the query planning and routing component, which will in turn decompose these queries into high level queries sent to individual translation components. Each translation component translates high level queries into source specific queries, and returns the results back to the query routing component. The query planning and routing component reassembles the high level query results into a single set of results which is sent back to the interface. The query planning and routing component will be referred to as the 'mediator', and the translation components as 'wrappers'.

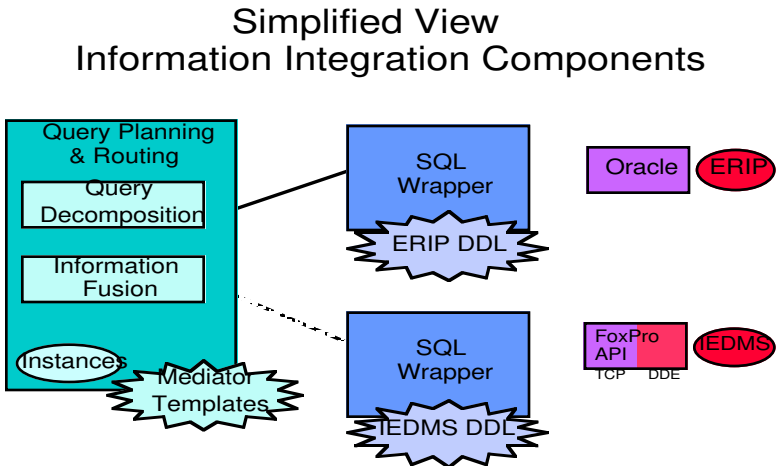


Figure 2 Simplified View of ERISII System Architecture

IV. Main Components

A) User Interface (not described in this document)

B) Communication Translation Component (not described in this document)

C) Communication OEM/QEM Components

B) Mediator, or Query Planning and Routing Component

C) Wrappers, or Translation Components

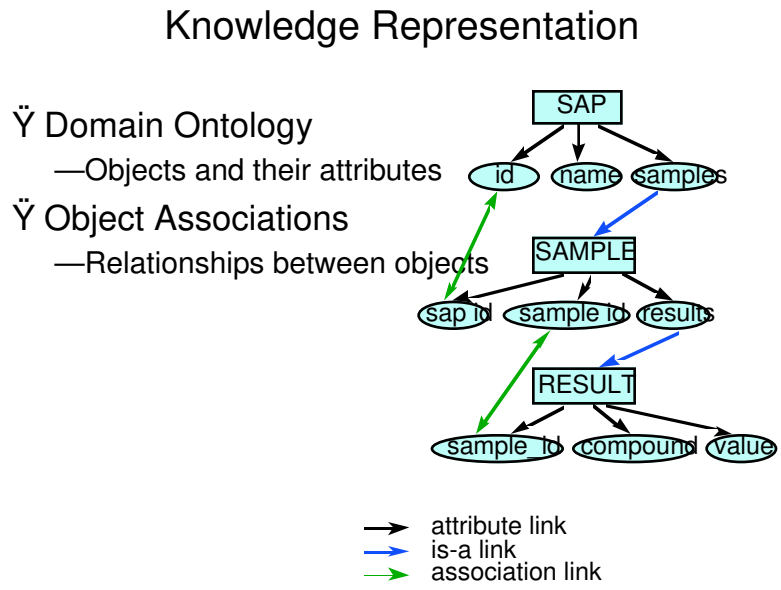
D) Security Component

These components will each be described in detail in the following sections.

V. Domain Knowledge Representation

In order to systematically provide information in a user centered way, all data will be represented in terms of objects. Every object represents type of entity or grouping of data. For example, Samples, Wells, or Compounds, all can be considered as objects. Objects all have a set of attributes, an attribute either representing a primitive data type in which case it is primitive, or representing the relationship with another object, in which case it is composite. For example, a sample can have the following primitive attributes: date-in-lab, date-out-of-lab, sample-number. A sample can also have the following composite attributes: analysis-results, associated-wells etc..

In this design we are making the assumption that all agents will use the same terminology to communicate among one another. In other words we will be using a common knowledge representation, and every agent will understand at least a subset of this common knowledge representation. Common knowledge representation in this context refers to both the terminology used by all the components of the system, as well as the relationships which link together the various objects. For arguments sake the ERISII common knowledge representation will be referred to as ontology. A simple ontology can be illustrated as follows:



This particular example defines three objects, Sample Analysis Plan Table, Sample, and Result, and shows the various relationships that exist between them. While the actual ERISII domain involves much more information than contained here, this example will suffice for demonstration purposes. A Sample Analysis Plan table contains a set of Samples, and a Sample contains a set of Results. Sample Analysis Plan Tables are linked to Samples by the fact that an SAP id is a Sample sap_id and Samples are linked to Results by the fact that a Sample sample_id is a Result sample_id. We called these links Sample Analysis Plan Table samples, and Sample results respectively.

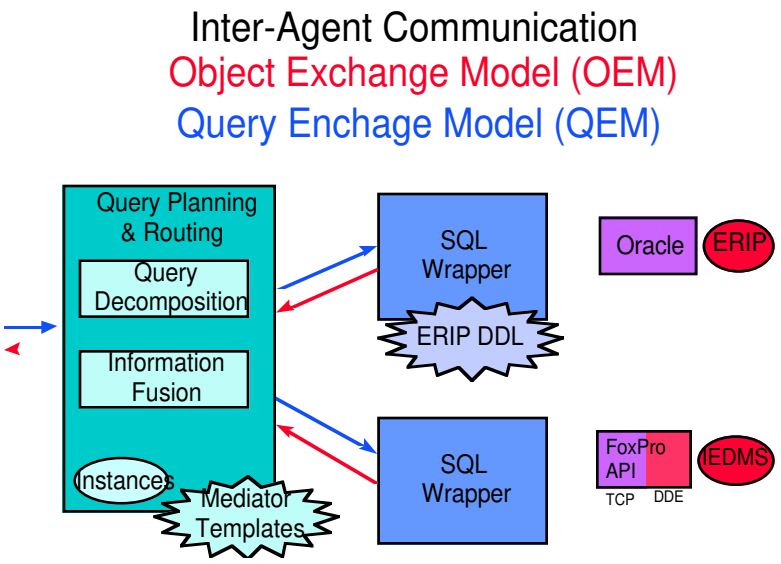
The ability to view a common knowledge representation in abstract form will not be sufficient to insure that all agents in the system will communicate in an unambiguous way. For this reason we have decided to adopt a formal representation for the domain ontology. The syntax of this representation is a superset of the Object Definition Language published by the Object Database Management Group consortium, designed with the goals of enhancing standardization and interoperability between object oriented databases. Following our initial example, an ODL file can be constructed as follows:

```
interface SAP (key id)
  {
     attribute String id;
     attribute String name;
     relationship List<Sample> samples
        {
          SAP.id = Sample.sap_id
        };
   }

interface Sample (key sample_id)
   {
     attribute String sap_id;
     attribute String sample_id;
     relationship List<Result> results inverse Result::sample
        {
          Sample.sample_id = Result.sample_id
         };
   }

interface Result (keys id, sample_id)
   {
     attribute String id;
     attribute String sample_id;
     attribute String compound;
     attribute Number value;
     relationship Sample sample inverse Sample::results
        {
          Sample.sample_id = Result.sample_id
         };
   }
```

There are two necessary modifications to the original ODMG ODL syntax. The first is the presence of expressions defining relationships between objects, and the second is the ability to use user defined primitive data types (String and Number are both user defined data types, and not built in as ODMG defines them).

The use of a precise knowledge domain model enables the various components of the system to use the same terminology in sending queries, receiving responses, and providing schema related information.

Inter-Agent Communication



All agents in the system will communicate using a single protocol. All queries will be represented as Query Exchange Model (QEM) structures, and all results will be represented as Object Exchange Model (OEM) structures. This will enable the easy addition of wrapper components, as well as the creation of multiple levels of mediators, one mediator serving as a 'wrapper' for another mediator. These structures can be shared from agent to agent using the distributed object invocation offered by CORBA.

VI. The Query Exchange Model

The Query Exchange Model is an abstract representation of a query, or request for information. The actual structure used resembles a tree of nodes, each node either representing requested information, or information provided to constrain a request. We have found the need to provide five different types of QEM nodes:

Object Node:

Contains an object name, a constraint link, and a list of return links. This represents a composite attribute. The constraint link generally points to a constraint expression tree, and each return link points either to a primitive node or another composite node.

Attribute Node:

Contains an attribute name representing a primitive attribute. Specifies information to be returned from a data source.

Relationship Node:

Represents the link from one object to another. A relationship here is always local to an object an represents a request for information about another object related to this object. A relationship node also has a constraint link and a list of return links.

Literal Node:

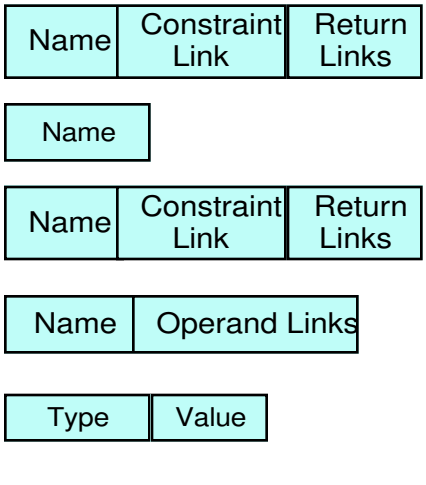
Contains a type and a value. Specifies information provided to constrain a query.

Operator Node:

Contains an operator name, and list of operand links.

Query Exchange Model

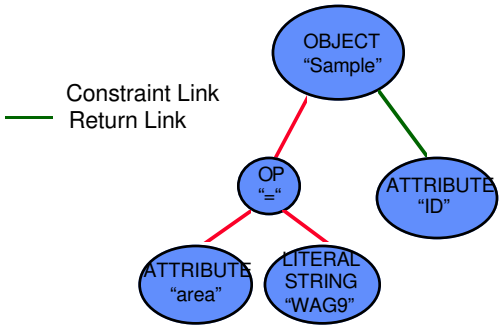
- **QEM Node Types**
  - Object Node
    - Object Name
    - Constraint Tree
    - Return List
  - Attribute Node
    - Attribute Name
  - Relationship Node
    - Relationship Name
    - Constraint Tree
    - Return List
  - Operator Node:
    - Operator Name
    - Operand List
  - Literal Node
    - Type
    - Value.



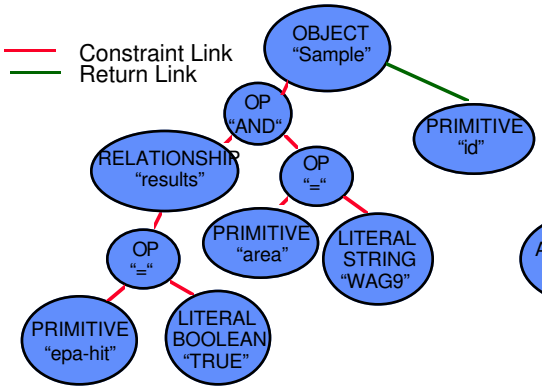
The following page illustrates a few example QEM queries.

EXAMPLE QEM QUERIES

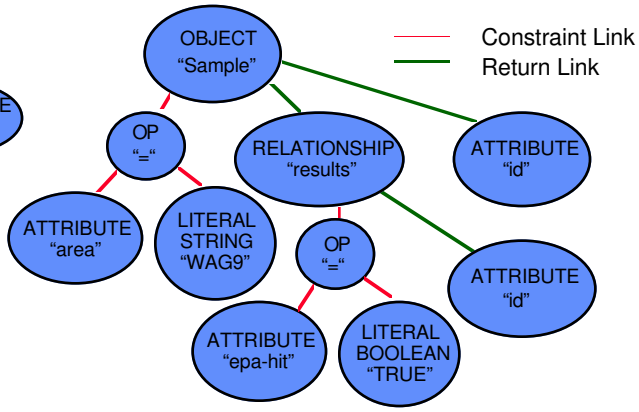
Find all Samples in WAG 9



Find Samples in WAG 9 with any results greater than EPA limits



Find Samples in WAG9, and for each sample, find Results greater than EPA limits.



VII. The Object Exchange Model

The various agents in this system will be using the Object Exchange Model (OEM) as a method of returning query results. The Object Exchange Model is fully described in "Object Exchange Across Heterogeneous Information Sources" (Papakonstantinou Y., H. Garcia-Molina, and J. Widom) and will only be briefly described here. OEM is characterized by the fact that an agent receiving the results of a query need not know the schema of data returned, because the schema is embedded within the return values themselves. An OEM object contains the following information:

1) **An object identifier** - Used to uniquely identify an object . This is a local reference

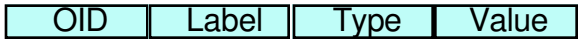
2) **A label** - Represents the schema name of an object

3) **A type** - Represents the type of an object. Is not restricted to standard types.

4) **A value** - Contains the value of a primitive object, or references to other objects.

Representing OEM objects within ERISII involves using a structure we will refer to as an OEM structure. The OEM structure is composed of OEM nodes. These nodes can be constructed, or referenced across a network through the use of CORBA object persistence. The following diagrams illustrate the composition of OEM nodes.

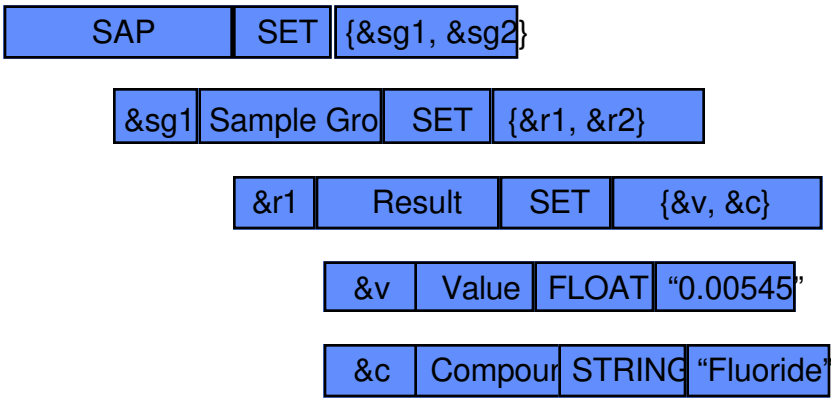
Object Exchange Model (OEM)



- **OEM Node**
  - string OID
  - string label
  - string type
  - union value
    - OEM *
    - list of OEM *
    - int
    - float
    - string
    - other primitive type i.e. binary encoding of a document

OEM Example

- Example of an SAP composition



The previous illustration is not complete due to space constraints. This example can be formally represented as follows:

```
<&s, SAP-Table, set, {&s1, &s2}>
   <&s1, Sample, set, {&r1, &r2}>
      <&r1, Result, set, {&v1, &c1}>
         <&v1, value, float, “0.00545”>
         <&c1, compound, string, “Fluoride”>
      <&r2, Result, set, {&v2, &c2}>
         <&v2, value, float, “0.08675”>
         <&c2, compound, string, “Sodium”>
```

```
<&s2, Sample, set, {&r3, &r4, &r5}>
   <&r3, Result, set, {&v3, &c3}>
      <&v3, value, float, "0.08768">
      <&c3, compound, string, "Zinc">
   <&r4, Result, set, {&v4, &c4}>
      <&v4, value, float, "0.00587">
      <&c4, compound, string, "Fluoride">
   <&r5, Result, set, {&v5, &c5}>
      <&v5, value, float, "0.05666">
      <&c5, compound, string, "Fluoride">
```

This represents one Sample Analysis Plan Table. This table contains two samples. Each sample contains at least two results, each result containing a compound name and a value. Though this is a somewhat simplistic example, this representation extends easily to more complex embedded data.

Though OEM was primarily conceived to represent query results, it is also possible to represent a QEM query using the OEM format. The following text illustrates how to represent a QEM query in OEM. Each QEM node logically maps to an OEM node.

Using OEM to represent QEM Queries

Attribute Node:

```
<&oid, name, attribute, null>
```

Object Node:

```
<&oid,  name, object, {&constraint, &return1, &return2, …}>
```

Relationship Node:

```
<&oid,  name, relationship, {&constraint, &return1, &return2, …}>
```

Operator Node:

```
<&oid, name, operator, {&operand1, &operand2, …}>
```

Literal Node:

```
<&oid, null, type, value>
```

VIII. Mediator Composition

The main function of the mediator is to decompose queries it receives into queries destined for multiple sources and to recompose the results of these queries into a single set of results. An agent using the services of a mediator must not be required to provide any source related information. In this case the query received by the mediator is a QEM structure. Based on the QEM structure the mediator must determine how to divide the larger query into single source sub-queries, as well as determine which sources these sub-queries must be sent to. The mediator must have *at least* the following information to decompose a QEM query into multiple sources:

1) Information about which objects and attributes it supports, as well as the relationships between the objects it supports.

2) Information about where the sources are.

3) Information about which objects and which attributes are supported by which sources

The mediator must also convey information to the agents it is serving including where it is, as well as what objects and attributes it supports. We will initially construct a simplified model representing all the information listed above, and proceed to identify exactly how the mediator will access this information.

A) Mediator Architecture

The mediator consists of three main components: the domain model access component, the query decomposition component, and the information fusion component. Building the domain model involves both reading the ODL specification file, and sending meta-queries to each source. The domain specification file syntax is described in chapter II (ODMG ODL with some extensions). The process of building the domain model using the ODL file and meta-query information will be described in the following two sections. In the third section we will discuss the query decomposition and information fusion process through the use of an example.

B) The Mediator Domain Model

Since the ER domain involves manipulating objects and attributes, it is only sensible for the mediator model to contain objects and attributes as well. It is safe to conclude that the mediator should contain a list of objects, and that each represented object should contain a list of attributes. An object in this sense denotes a type of object, as opposed to a particular instance of an object, such as a sample in general as opposed to a particular sample.

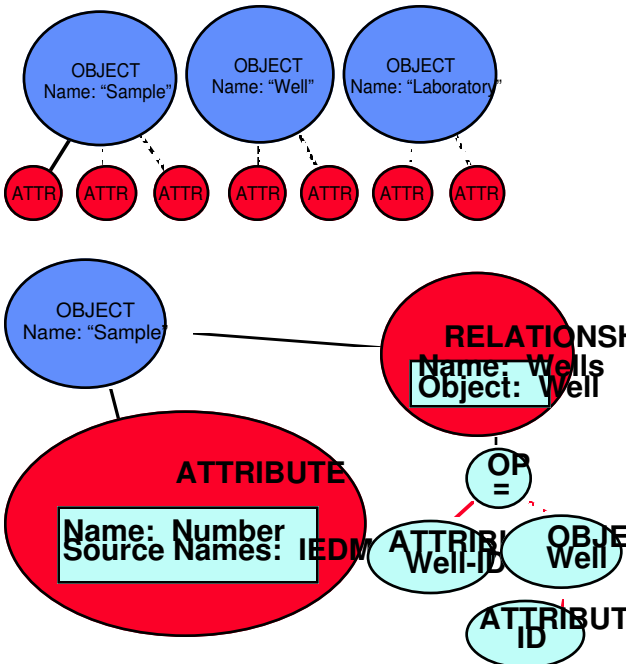
To allow the overlapping of objects over multiple sources, each object is associated with a list of sources, and to allow for duplicate information, each attribute is associated with a list of sources as well. A sample is present in ERIP, as well as in IEDMS. It is therefore associated with the list of sources {ERIP, IEDMS}. The attribute sample-number of sample is also associated with the same list of sources, because a sample-number is stored both in ERIP and IEDMS.

This model takes care of information about which objects and attributes the mediator supports, as well as information about how these objects and attributes are distributed among the sources. Now remains the relationships between objects. One possible way of representing this information could be to have a separate model containing associations for each pair of related object types. In this case association information can be obtained by specifying two object types such as “well” and “sample”, enabling the mediator to find the associated well of a particular sample, or the associated samples of a particular well. Another way of representing this information is to give a name to these associations, and have them be local to an object. The ODMG model generally calls these associations relationships. For example, a well can have a relationship named “associated-samples” which links well objects to sample objects. The advantage of the latter approach is the ability to define more than one type of association between two given object types (EG: A well can have relationships such as “samples-with-hits”, “analyzed-samples”, “interim-samples”, “past-30-day-samples” etc…)

Supposing we are using the second approach, how do we represent these associations in our model? The simplest way to do this would be to incorporate relationships within the original model. Thus the mediator model is a list of objects, each object containing a list of attributes and a list of relationships. A relationship is an expression tree defining the association between the object it belongs to and another specified object.

Here is a graphical representation of the model:

Mediator Domain Model



C) Populating the Model

Most of the information concerning the objects and attributes the mediator supports, as well as the associations between these objects are read directly from the ODL file, which is configured by the administrator and is loaded into memory upon program execution. Information about which sources can be accessed will ideally be in a file as well in the form of a list of source names. All of the information pertaining to which objects and attributes are supported by which sources are dynamically supplied by the wrappers via *meta-queries*.

Meta Queries

In this design meta-queries use the same mechanism as traditional queries. The only difference lies in the nature of the returned information. Simple queries are asking for data, whereas meta-queries are asking for how the data is organized. In this case the meta-query information involves determining which objects and which attributes are supported by which sources, as well as which sources are there to begin with.

Every wrapper in the ERISII system as well as the mediator will support an object called SchemaObject and an object called SchemaAttribute. In order for the mediator to find out about the schema supported by a given source it will send a QEM query requesting information about SchemaObjects or SchemaAttributes. The ODL specification of SchemaObject and SchemaAttribute is as follows:

```
interface SchemaObject
  {
    attribute String name;
    attribute List<SchemaAttribute> attribute-list;
```

```
  }

interface SchemaAttribute
  {
    attribute String name;
  }
```

A schema-object must have at least an object name and a list of attribute names. This composition can be expanded to include other schema related information as it is made available at the level of the wrappers, such as whether it is more expensive to access one attribute rather than another, etc..

Querying About Schema Information

ŸQUERY

```
<&q1, schema_object, composite, {null, &r1, &r2}>
<&r1, object_name, primitive, null>
<&r2, attribute_list, primitive, null>
```

ŸANSWER

```
<answer, set, {&o1, &o2}>
    <&o1, schema_object, set, {&on1, &oa1}>
        <&on1, object_name, string, "Sample_Activity">
        <&oa1, attribute_list, set, {&a1, &a2,...}>
            <&a1, attribute_name, string, "Name">
            <&a2, attribute_name, string, "Description">
            ...
            <&a8, attribute_name, string, "Results">
    <&o2, schema_object, set, {&on2, &oa2}>
        <&on2, object_name, string, "RESULT">
        <&oa2, attribute_list, set, {&o2a1, &o2a2, &o2a3}
            <&o2a1, attribute_name, string, "SAMPLE_ID">
            <&o2a2, attribute_name, string, "COMPOUND">
            <&o2a3, attribute_name, string, "VALUE">
```

Sample_Activity
- string Name
- string Description
- string SA_id
- string Area
- DATE date_taken
- DATE date_returned
- LOC Location
- RESULT Results

RESULT
- string SAMPLE_ID
- string COMPOUND
- float VALUE

D) Query Decomposition and Information Fusion

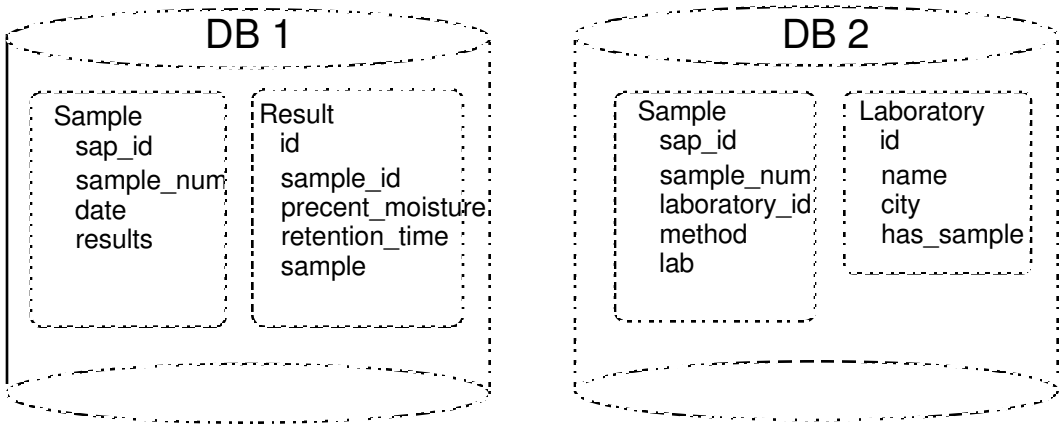
After the mediator reads the ODL domain specification file and sends the meta-queries to locate the sources of each object and attribute, the mediator can process incoming queries. It performs decomposition, assembles the results, and sends them back to the agent making the request. Decomposing the query mainly depends on the sources associated with each object and attribute in the query. Given a query, the mediator will first make an in-order traversal (first look at the left child, the root and then the right child) of constraint portion of the query. If the source of the parent object is different from the source of its composite attribute, a query will be constructed and sent to the source of this attribute. The return value will become a new constraint of the parent object, and this process is continued until the root object is reached. Once the constraint tree is traversed, the return side is examined in determining which information is sent back to the interface.

Instead of showing a detailed algorithm for query decomposition and information fusion, we will describe a simple example. Hopefully, through this example, you will have a good idea of how the algorithm works.

The domain specification file :

```
interface Sample (key sample_num)
  {
    attribute String sap_id;
    attribute String sample_num;
    attribute String laboratory_id;
    attribute Date date;
    attribute String method;
    relationship List<Result> results inverse Result::sample
        {
         Sample.sample_num = Result.sample_id
        };
    relationship Laboratory lab inverse Laboratory::has_sample
        {
         Sample.laboratory_id = Laboratory.id;
        }
  }

interface Result (keys id)
  {
    attribute String id;
    attribute String sample_id;
    attribute Number precent_moisture;
    attribute Number retention_time;
    relationship Sample sample inverse Sample::results
       {
         Result.sample_id = Sample.sample_num
        };
  }

interface Laboratory (keys id)
  {
    attribute String id;
    attribute String name;
    attribute String city;
    relationship List<Sample> has_sample inverse Sample::lab
       {
         Laboratory.id = Sample.laboratory_id
        };
  }
```

Suppose the above schema is organized in the following way.



Note: DB in this example refers to the wrapper name, and not a database name. EG: DB1 = ERIS, and DB2 = IEDMS. This algorithm is also applicable for {DB1, DB2, … DBn}

A meta-query is sent to each source and the mediator receives the above information. For each object and attribute specified in the domain specification file, we will associate the source information with it. The mediator will have the following global meta-schema.

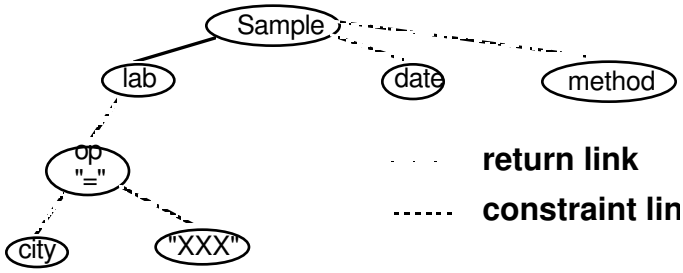
```
interface Sample {DB 1, DB 2} (key sample_num)
  {
     attribute String sap_id; {DB 1, DB2}
     attribute String sample_num; {DB 1, DB 2}
    attribute String laboratory_id; {DB 2}
    attribute Date date; {DB 1}
    attribute String method; {DB 2}
   relationship List<Result> results inverse Result::sample
       {
        Sample.sample_num = Result.sample_id
       };
   relationship Laboratory lab inverse Laboratory::has_sample
       {
        Sample.laboratory_id = Laboratory.id;
       }
  }

interface Result {DB 1} (keys id)
  {
    attribute String id; {DB 1}
    attribute String sample_id; {DB 1}
    attribute Number precent_moisture; {DB 1}
    attribute Number retention_time; {DB 1}
   relationship Sample sample inverse Sample::results
      {
        Result.sample_id = Sample.sample_num
       };
  }
```

```
interface Laboratory {DB 2} (keys id)
  {
     attribute String id; {DB 2}
     attribute String name; {DB 2}
     attribute String city; {DB 2}
    relationship List<Sample> has_sample inverse Sample::lab
       {
         Laboratory.id = Sample.laboratory_id
        };
  }
```

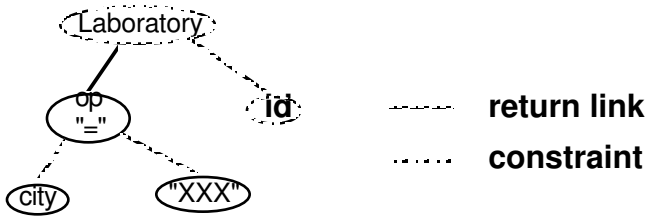
where **{DB 2}** means the attribute/object is located in the source DB 2.

QUERY:

Find the date and method of all the samples which were tested in the laboratory located in the city "XXX"



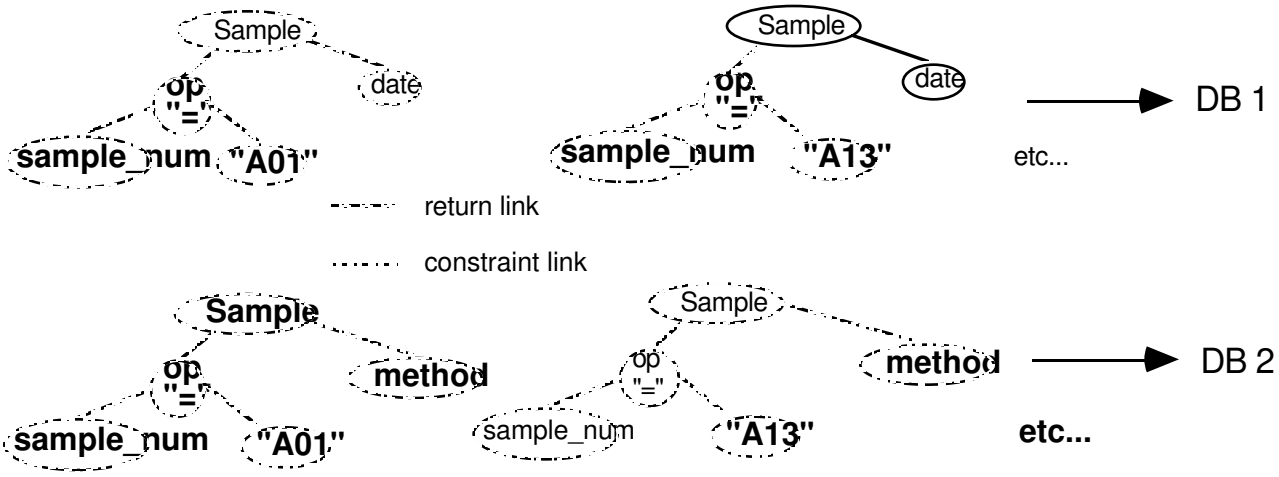
The mediator uses in-order (left child > root > right child) traversal to look at the first composite object of the constraint tree. Since "lab" is the only composite attribute, the mediator will first look at it. According to the global meta-schema, "lab" is a relationship which links to the Laboratory object, which is from the source DB 2. However, the parent composite object (in this case, it is the root object) of "lab", which is "Sample", is not only from DB 2, but also from DB 1. Therefore the mediator uses the association expression "*Sample.laboratory_id = Laboratory.id*" to find all the laboratory ids given the city is "XXX" from the source DB 2. The query can be viewed as:



Suppose the query results are "A1234" and "B5678". Since "laboratory_id" is not the unique key of the "Sample" object, the mediater needs to send another query to the source DB 2 to get the sample_nums given that the laboratory_ids are "A1234" and "B5678". Suppose the results are "A01", "A13", "A29" and "B11".

The mediator looks at the attributes on the return side. It finds that "date" (from DB 1) and "method" (from DB 2) are not from the same sources. Therefore, the mediator will

construct two sets of queries, one is to DB 1 and the another is to DB 2. The queries will be as follows:



Suppose the result is :

From DB 1 :

for “A01”,
 <&r1, result, {&s1}>
 <&s1, date, string, “Feb 2, 93”>

for “A13”
 <&r1, result, {&s1}>
 <&s1, date, string, “Mar 23, 93”>
…

From DB 2 :

for “A01”,
 <&r1, result, {&s1}>
 <&s1, method, string, “ABC”>

for “A13”
 <&r1, result, {&s1}>
 <&s1, method, string, “CCC”>
…

Then the mediator will pack the results and return them as follows:

<&r1, result, {&s1, &s2, &s3, &s4}>
 <&s1, result1, set, {&ss1, &ss2}>
 <&ss1, date, “Feb 2, 93”>
 <&ss2, method, “ABC”>
 <&s2, result1, set, {&ss3, &ss4}>
 <&ss3, date, “Mar 23, 93”>
 <&ss4, method, “CCC”>

....

IX) Handling Multiple Clients

One of the important requirements of the mediator is the ability to simultaneously handle multiple queries from multiple clients.

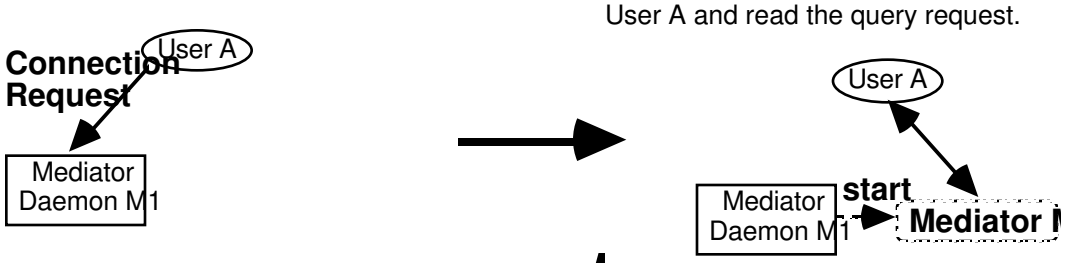
There are several different ways of solving this problem. First, we can have only one mediator running at a time, and enable this mediator to handle multiple query requests. In order to identify which query belongs to which client, information will have to be sent by the client uniquely identifying itself, and the mediator will need to contain a table to keep track of information relating queries to clients. This causes overhead to search as well as update such a table. Moreover, since there is only one mediator process, if there are many requests, the mediator may not be able to handle them all at the same time. As a result, a long queue of queries will be created, and the mediator will become the bottleneck of the whole system. Therefore this approach is not acceptable.

Instead, we will have a mediator daemon which runs continuously in the background once the system is started. Before a client program (either mediator or user application) sends a query to the mediator daemon, it will first try to establish a connection. When the mediator daemon receives this request, it will start a “real” mediator. This “real” mediator handles the requests from this client only. Therefore, if there are ten clients, there will be ten different mediators, each one working with only one client. When the client finishes the query, it can decide whether it still wants to keep the connection. If not, once a connection is closed, the corresponding mediator will be destroyed as well as its connection to other mediators or wrappers.

The advantage of this approach is that each mediator will only focus on serving one client, which makes the mediator simple. Moreover, a long waiting list won’t exist. The idea can be illustrated by the following examples.

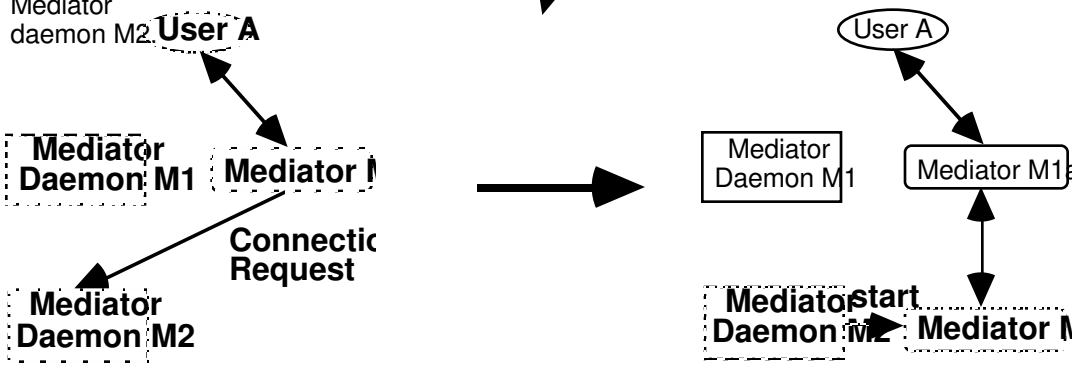
User A want to send a query to the mediator M1. First, it will try to establish a connection.

Then the mediator daemon M1 starts a media called M1a, which establishes a connection w User A and read the query request.

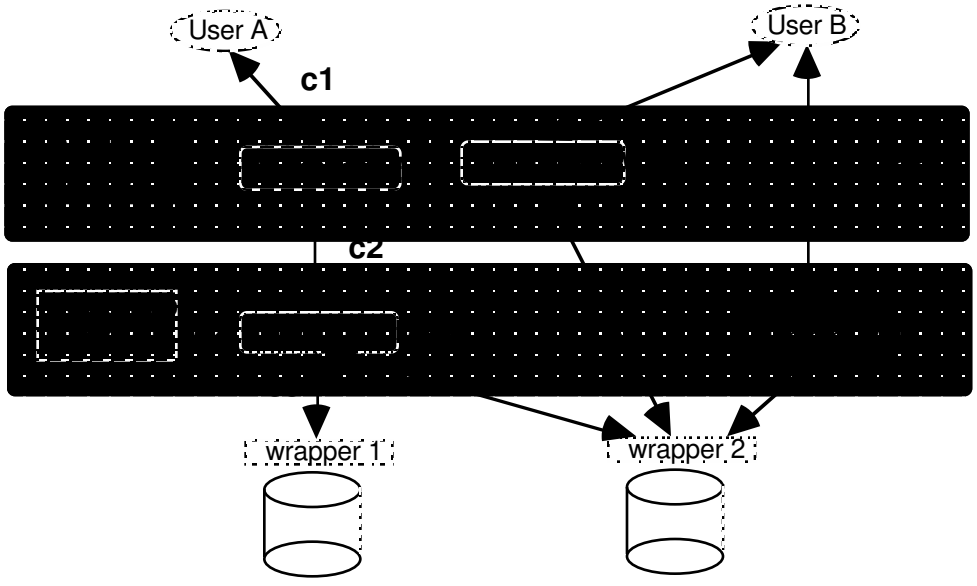


Later during the decomposition, Mediator M1a decided to send a subquery to the other Mediator M2.
Then M1a will do the similar things as User A. First, it will send a connection request to Mediator daemon M2.

Mediator M2 will start a new media called M2a, which will take care of M1a.



Basically, the following situation will happen.



Please note that each mediator will only have one "parent".

Later, when connection c1 is closed, the mediator M1a will be killed. At will be disconnected as well. Then, similarly, the mediator M2a will be d so are the connection c3 and c4.

Similar idea can also be applied to the wrapper so that each wrapper will deal with one mediator at a time. Within the CORBA framework, this involves providing the client access to a Daemon class. The Daemon Class will have the following IDL specification:

```
interface Daemon
  {
    QEMReference open_connection(server_name);
    close_connection(QEMReference);
  }
```

The mediator opens a connection with a wrapper by remotely invoking the open_connection method of the wrapper daemon, and by specifying the name of the wrapper as an argument to this call. This causes the wrapper daemon to start a new wrapper process. The wrapper process creates an instance of a QEM tree, and a reference to this tree is passed back to the mediator as a result of the call to open_connection. Any further manipulation of the QEM tree by the mediator is transmitted to the QEM tree in the newly created wrapper through remote invocation.

X. Wrapper Composition

The primary function of the wrapper is to receive queries in QEM form, to translate these queries into source specific queries, and return a set of OEM results. For the first portion of this project we will be focusing on accessing relational databases, and later build wrappers for accessing information in non-relational form such as GIS, or flat file information. Thus most of the initial design will involve the description of a generic SQL wrapper, and will be based on the design of the existing SQL wrapper developed by ISX for the Integrated Weapon Systems Database project at Lockheed.

From the point of view of the client requesting information, the wrapper can service identical queries that the mediator would service. The main difference between the mediator and the wrapper lies in the fact that the wrapper is generally designed to answer for either a unique source, or a unique type of source (in this case an SQL source only), whereas the mediator can transparently answer for several different types of sources such as SQL, flat-file, or non-SQL graphical information systems.

For this reason, the model the wrapper uses to represent information will be very similar to the one used by the mediator, and the tools used to build the model will be the same as well. The particular subset of the domain knowledge representation that the wrapper understands will be represented in the same form that it is represented at the level of the mediator: through the configuration of an ODMG ODL file. The main difference in the wrapper representation is the necessary addition of mappings from object terminology to relational terminology, joins between tables, and database specific source related information.

Supposing that we were to configure an ERIP wrapper which happened to support the concepts of Sample and Result, we might create the following ODL file for the wrapper:

```
database ERIP (user erickson
               password sesame
               type oracle7)
```

```
{
  join sc_results(CAS_NUM) to compound(CAS_NUM);

  interface Sample (key sample_id)
    {
      attribute String sap_id(sc_field_sample.SAP_NUM);
      attribute String sample_id(sc_field_sample.SAMPLE_NUM);
      relationship List<Result> results inverse Result::sample
        {
          Sample.sample_id = Result.sample_id
        };
    }

  interface Result (keys id, sample_id)
    {
      attribute String id(sc_results.CAS_NUM);
      attribute String sample_id(sc_results.SAMPLE_NUM);
      attribute String compound(compound.COMPOUND_NAME);
      attribute Number value(sc_results.CONCENTRATION);
      relationship Sample sample inverse Sample::results
        {
          Sample.sample_id = Result.sample_id
        };
    }
}
```

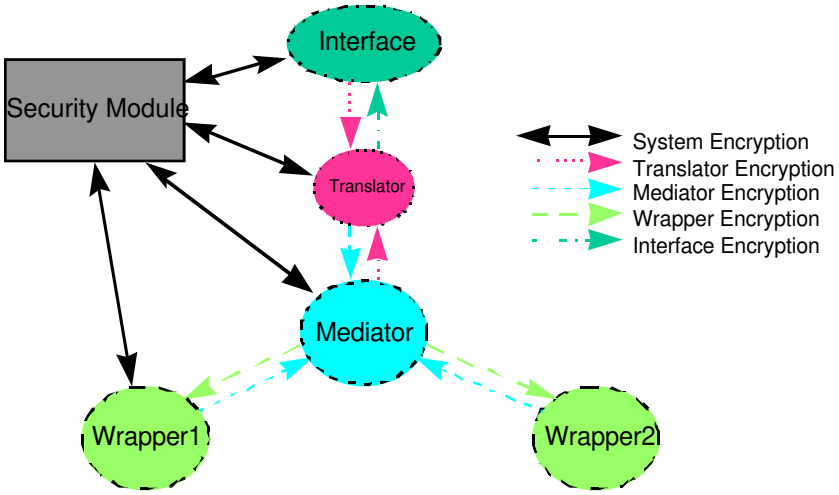
Note1: If two tables are used by the same object, there must be an explicit join between the two tables, specified by the join statement above. A join can involve one or more fields in common. In this example there is join between the compound table and the sc_results table.

Note2: The relationships specified above do not have to be present if they are present in the mediator ODL file.

Note3: Any number of databases can be defined within the same wrapper ODL file with different types, usernames, and passwords.

XI. Security

Security is another big issue which needs to be handled properly in the ERISII system. All of the security functions of ERISII will be handled in the Security Module. Each individual component of the ERISII system is associated with a unique encryption key. Any message sent from component A to component B in the system must be encrypted using the encryption key of component B. The only exception to this rule is communication between all components and the security component. In this case a system wide encryption key is used for messages going to the security component, and coming from the security component. Since none of the non-security components know the encryption keys of the other non-security components, anytime a query or response is sent from one component to another, authorization must be obtained first from the security module.



ERISII Security System

The security module contains mappings from system component id's to encryption keys. When one component wishes to communicate with another component, it must provide the security component with

a) A user name

b) A user password

c) A list of object names, either obtained from an OEM tree, if the communication involves sending return values, or obtained from a QEM tree, if the communication involves sending a query.

d) The source component id

e) The destination component id

The security component will determine from this information whether communication is authorized. If the communication is authorized, the security component will send back to the sending component the encryption key of the receiving component. Care must be taken in insuring that the encryption key is immediately destroyed after the authorized message is sent.

How does the security component determine whether communication should be authorized? Within this component an account will be created for each user. Each user has an assigned security *type*. Each security type contains a list of accessible component ids, and for each component, a level of access. EG:

Security Type 1

Component A, Level 1.

Component B, Level 2.

Component C, Level 0.

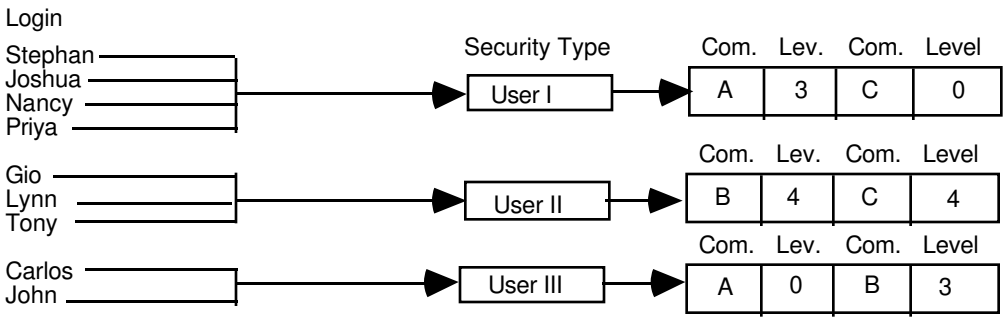
Security Type 2

Component A: Level 3.

Component B: Level 4.

The security level is a number defining the level of security within a given component. Each security level is associated with a list of object names. If a user has security level 4 access to component A, and level 4 is associated with objects Sample, Result, and Lab, this user can only access Sample, Result, and Lab objects in component A.

The security module can determine, for each user, which components the user can access, and for each component which objects are allowed. The first steps in the authorization process involve the verification of the user and password, and the determination of the user security type. For this reason there is a table of the different logins that can access the system and for each of them the associated security type. This table will be updated if necessary, and the insertions, deletions and updates can be only be done if a superuser password is given.



A = Eris Wrapper
B = IEDMS Wrapper
C = Mediator
D = Interface
E = Comm Translator

Com. = Component
Lev. = Level

Process Steps:

- The user interface will send to the Security Module the login and password of the user, the list of objects used in the query, its id and the Communication translation module's id in a message encrypted with the use of a system encryption key.

-The Security Module will check to see if the particular user involved has access to the specified objects. If so, the Security Module returns the encryption key of the Communication Translation Component, encrypted with the system encryption key, back to the interface.

-The interface will decrypt the Translation Component key with the system decryption template. Once the user interface sends the query, it will encrypt this query using the Translation Component encryption key.

-The Communication Translation Component will decrypt the message, obtain the encryption key of the mediator from the security module, encrypt it again, and send it on.

-At the Mediator level the received information will be decrypted, and the query will be decomposed, and then the decomposed queries will be encrypted with the individual Wrapper encryption keys, and sent to the wrappers. Each wrapper encryption key must be obtained from the security module according to the standard process.

- The wrapper will decrypt the message with it's own decryption template. Once the wrapper has this information, it will execute the query. Once it gets the information from the source, it obtains the mediator encryption key from the security module, and results are encrypted again and send back to the mediator.

- The same authorization process is used from the mediator to the translation component, back to the interface.

Note: For efficiency reasons, a few alternate options are available. The Communication Translation Component need not be considered a component from a security point of view. In this case the interface would directly use the mediator encryption key. Other options are to use authorization on a per-connection basis, and not on a per query basis. In other words, once a component obtains authorization to access another component, it would retain the encryption key of that component. Future queries would not require authorization from the security component.

This method of encryption is very secure as long as the system encryption key, and the encryption keys of all the components are unknown to the user. If system key becomes available, the entire system is open. If an individual component key becomes available, only that particular component will be open.

The following precautions will be followed, to insure that the system remains secure:

1. None of the encryption keys will be stored on file without pre-processing.

2. The keys will be of variable length.

3. The ODL files of each component will not be stored on file as is. They will be encrypted with the particular encryption key of that component (to be investigated..maybe too easy to decifer, given knowledge of ODL file syntax).

4. The administrator will be given tools to encrypt and decrypt ODL files. These tools will require password verification.

XII) Conclusion

The ERISII information access components have been designed with several goals in mind. We found the need for a small domain independent, yet expandable system. ERISII should eventually provide all of the advantages of object oriented database access. In addition it should provide a systematic and uniform way of accessing information, independent of the type of information provided, and have the ability support rapidly changing schema, at first through run-time ODL parsing, eventually through automated ODL file creation.

The system is designed to be dynamic, not only with respect to schema evolution, but also with respect to the rapid introduction of new sources. Providing simple query structure and return value structure specifications for use throughout the design, and publishing the interface for *both the query structure and return value structure* in a clear unambiguous fashion will facilitate the future addition or modification of the various components in the system, including introduction of wrappers designed to access flat file information, the use

of multiple levels of mediators, or the design of high level ‘accessory’ agents which require the uniform access to information as a prerequisite.

The functions of the system have been divided into separate components as much as it was deemed feasible, the main components being the interface, responsible for user access, the security module, the mediator, responsible for query decomposition and information fusion, the wrapper, responsible for source specific query translation, the communications module, responsible for inter-component query and return value composition, and the communication TCP/IP-CORBA translation module.

Careful thought was put into ensuring that the system will be suitable for a production environment, particularly with respect to the handling of simultaneous multiple client access, as well as the design of a multiple level security system capable of placing user restrictions both on a per source, and a per object basis.