DYNAMIC META-DATA SUPPORT FOR INFORMATION INTEGRATION AND SHARING ACROSS HETEROGENEOUS DATABASES

by

Wiput Phijaisanit

A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
the Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:

_____ Dr. Larry Kerschberg, Dissertation Director

_____ Dr. Alex Brodsky

_____ Dr. David A. Schum

_____ Dr. Xiaoyang Wang

_____ Dr. Carl Harris, Interim Associate Dean for Graduate Studies and Research

_____ Dr. W. Murray Black, Interim Dean, School of Information Technology and Engineering

Date:_____ Summer Semester 1997
George Mason University
Fairfax, Virginia

Dynamic Meta-Data Support for Information Integration and Sharing Across Heterogeneous Databases

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University.

By

Wiput Phijaisanit

M.B.A, Management, Oklahoma City University, Oklahoma, 1991
B.E., Electrical Engineering, King Mongkut's Institute of Technology Ladkrabang, Thailand, 1988.

Director: Dr. Larry Kerschberg
Chairman,
Information and Software Systems Engineering

Summer 1997
George Mason University
Fairfax, Virginia

Acknowledgments

I would like to thank my dissertation director, Dr. Larry Kerschberg, for his encouragement and support throughout this research. I would like to thank the rest of my committee members, Dr. Alex Brodsky, Dr. David A. Schum, and Dr. Xiaoyang Wang for their comments and suggestions. I would like to thank William Wong, and Atcha Wong for their understanding and sharing.

Above all I would like to thank my mother, M.R. Narumol Phijaisanit, and my brother, Wichack Phijaisanit, for their unconditional love and support. Without them, I will never be as I am today.

Table of Contents

Table of Figures

Abstract

DYNAMIC META-DATA SUPPORT FOR INFORMATION INTEGRATION AND SHARING ACROSS HETEROGENEOUS DATABASES

Wiput Phijaisanit

George Mason University, 1997

Dissertation Director: Dr. Larry Kerschberg

In a multidatabase environment, meta-data plays a more important role than in a single database environment. The federated database system manipulates data from several data sources. Users hardly have enough knowledge about the underlying meaning of the global schema. Meta-data describes the data semantic information that can help users to interpret the data. It provides necessary information for both query processing and data integration in a federated database system. However, traditional data modeling techniques such as relational data model and semantic data model are not designed to support efficiently the definition, storage and manipulation of complex meta-data.

The dissertation investigates on how the schema of the data model can be extended to support the meta-data, how this meta-data can enhance the semantics of a

database schema, support data integration and mediation services, and facilitate query processing in a federated database system. The dissertation develops the Mediation Data Model. The Mediation Data Model provides an extensible schema that supports the dynamic meta-data, and a framework for specifying mediation services within the data model. The dynamic meta-data describes different types of data semantic information, in addition to the traditional static meta-data. The dynamic meta-data allows each instance object of a class to have associated with it distinct meta-data, whereas static meta-data assigns the same meta-data to an entire class rather than instances of a class. The Mediation Data Model provides the association of multiple unit values to an instance of a data type. The concept of Multiple Unit Value (MUV) is implemented by means of the Unit-type Dynamic Meta-Data, the Unit Value Mediator and the extended DML. The MUV concept allows object attributes to have values expressed in convertible units.

The dissertation also develops the federated database system architecture and a working prototype, called InfoFED. InfoFED makes use of the Mediation Data Model as a common data model to provide the self-describing data as well as an intelligent data manipulation language to users, and to improve data integration by reducing the schema integration conflicts. InfoFED also provides a browser for both data and meta-data, an object-oriented federated schema, and the data source registration and management.

1. Introduction

1.1 Overview

Multidatabases [HBP92, HBP94, and Litwin88] are becoming more important as applications increasingly require access to diverse databases. Multidatabases are sometimes referred to as heterogeneous distributed databases. Multidatabases are an important area of current research. However, the problems and issues faced by multidatabase architects and designers are numerous because multidatabase management systems (DBMS) are composed of heterogeneous hardware, operating systems, database management systems and applications which result in several information integration conflicts [Bertino91, Heiler&Siegel91, and BLN86].

One of the solutions to the information integration problems among heterogeneous distributed databases is the federated database system approach [Sheth&Larson90]. A federated database system is a distributed system that acts as a front end to multiple local DBMSs. The federated database system provides a logically integrated view of existing heterogeneous, distributed databases. It entails developing a global schema (view) of the component heterogeneous databases, where definitions of these databases are expressed in a common data definition language, and discrepancies among these definitions are resolved before they are integrated into the global schema.

The federated database system usually provides full query facilities and insulates users from the component databases via its global schema. Although the integration process may use a common data model such as the traditional relational, functional, or object-oriented data model, it usually presents mutual semantic conflicts. These conflicts involve differences, redundancies, and incompatibilities with respect to names, unit values, and meanings among similar data.

This dissertation discusses the roles of meta-data and mediators that enhance the expressive capability of the data model, support query processing, and facilitate the data integration in the federated database system. The dissertation proposes the Mediation Data Model which extends its schema to support the Property Knowledge Package that is associated to each data element in the data model. The Property Knowledge Package is, in addition to the schema, the description package of the data element. It contains Dynamic Meta-Data types that enhance the data model by supporting the Multiple Unit Value Concept and provide information on unit type and data source that is useful for automating conflict resolution in federated database systems. The unit type information enhances the data model to support the Multiple Unit Value Concept which, in turn, allows heterogeneous databases to export their data in their own unit values without any unit conversion. The data source information, when presented with the data element, helps users to distinguish among different sources of the same data. These concepts will be presented in subsequent chapters.

1.2 Terminology

Before proceeding, it is useful to discuss the terminology used in this dissertation. The dissertation proposes the use of the federated database system as a basic architecture for integrating data from heterogeneous database systems.

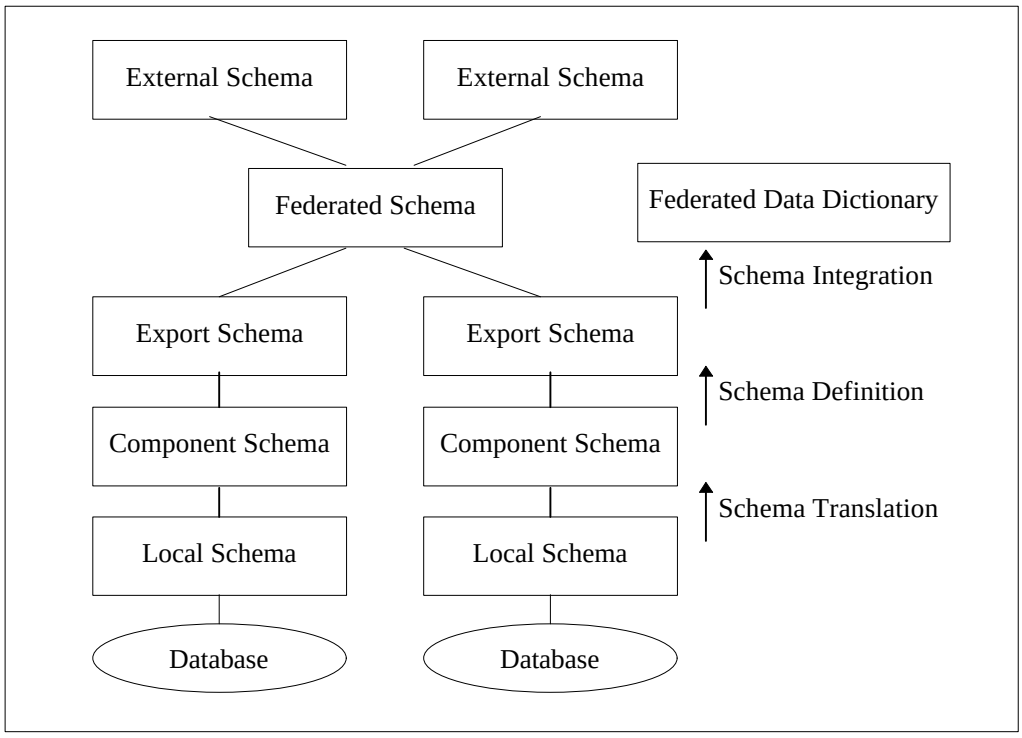


Figure 1-1 Federated Database System Architecture

A federated database system provides a global schema, expressed using the common data model's data representation, for resolving the differences in the data representation and functions among local DBMSs. Local DBMSs maintain their autonomy in which they preserve existing organizational investments in local applications

and user training, while providing a significant new function of global data access. Many of the existing federated database systems share the common architecture as shown in Figure 1-1 [Sheth&Larson90].

Each database has its own local schema. The Component schema is derived by translating local schemas into the common data model of the Federated Database System. The common data model facilitates the integration and allows access through a common query mechanism. Each database presents an export schema to the federation. This schema is either its actual component schema or a derived schema hiding some private local schema. For negotiation between databases, each federation has a single federated data dictionary, which is a distinguished component whose information domain is the federation itself.

The common data model is used in the federated database system to facilitate the schema integration from heterogeneous database systems. There are several types of integration conflicts that the federated database system designer has to deal with. The following is a classification of some of the possible conflicts between any two database schemas.

1) Identity conflicts. Identity conflicts occur when the same concept is represented by different objects in different databases. For example, copies of the same book stored in both IS-Library schema and Math-Library schema with different local identifiers.

2) Schema conflicts. Schema conflicts occur when the schemas that represent the same concept are not identical. Two major types of schema conflicts are naming conflicts and structural conflicts.

Naming conflicts occur when one name is used for more than one concept (homonyms) or when one concept is described by more than one name (synonyms). For example, in a homonyms case, the term “media” refers to magazines in one schema while it refers to videotapes in another. In a synonyms case, the term “reference” in one schema and the term “bibliography” in another schema are used to refer to the same concept as “a list of writings”.

Structural conflicts occur when the same concept is represented by different constructs of the data model, i.e., by a method in one database and a class in the other. For example, in the IS-Library schema, the number of citations is an attribute of the class ‘Book’ but in the Math-Library schema, it is a method recomputed upon request. The Structural conflicts also occur when the same concept is modeled by the same constructs, but classes have either different structure (missing or different relations/dependencies) or different behavior (different or missing operations). For example, the class ‘Book’ has an attribute ‘keyword’ in one schema but not in the other.

3) Semantic conflicts. Semantic conflicts occur when the same concept is interpreted differently in different databases. This category includes scale or rate differences. For example, a term ‘Conference’ is referred to conference in one schema but not in the other.

4) Data conflicts. Data conflicts occur when the data values of the same concept are different in different databases. For example, the same person appears with different salary values.

Recently, the object-oriented data model [Brathwaite92, Edward90, PBE95, RBPEL91 and Zhu&Maier88] has become a choice to represent the real world concept. It

supports a richer semantic compared to other types of data models such as the relational data model.

Object Orientation is an abstraction mechanism in which the world is modeled as a collection of independent objects that communicate with each other by exchanging messages. An object is characterized by its state and behavior and has a unique identifier assigned to it upon its creation. The state of an object is defined as a set of values of instance variables. The value of an instance variable is also an object. The behavior of an object is modeled by a set of operations or methods that are applicable to it. Methods are involved in sending messages to the appropriate object. The state of an object can be accessed only through messages; thus, the implementation of an object is hidden from other objects.

Each object is an instance of a class. A class is a template from which objects may be created. All objects of a class have the same kind of instance variables, share common operations, and therefore demonstrate uniform behavior. Classes are also objects. The instance variables of a class are called class variables and the methods of a class are called class methods. Class variables represent properties common to all instances of the class. A typical class method is “new”, which creates an instance of the class.

Classes of objects are arranged in a hierarchy or in a graph to describe the relationships of objects in a system. Classes are arranged in a hierarchy with the most general classes at the top and the more specialized classes below. When a class B is defined as a subclass of class A, class B inherits all the methods and variables of class A. Class A is called a superclass of class B. Class B may include additional methods and

variables. Furthermore, class B may redefine any method inherited from class A to suit its own needs.

The relations typically supported by the object oriented data model are: the classification or instance-of relation between an object and the class (typically one) of which it is an instance; the generalization/specialization or is-a relation between a class and its superclasses; and the aggregation relation between an object and its instance variables.

A mediator [Wiederhold92, and Wiederhold95] is a software module that exploits the encoded knowledge about certain sets or subsets of data to create information for a higher layer of an application. The mediator provides domain specialization services and will play a more important role in the information integration process from multiple data and information sources. Information-processing tasks in mediators include accessing to appropriate resources, data selection, format conversion, bringing data to common abstraction levels, integration of information from different sources, and preparing information for delivery to the customer.

1.3 Problem Statement and Contribution

In a multidatabase environment, meta-data plays a more important role than in a single database environment. The federated database system manipulates data from several data sources. Users hardly have enough knowledge about the underlying meaning of the global schema. The meta-data describes the data semantic information that can help users to interpret the data. In the data integration process, meta-data provides necessary

information for mediators to use in both query processing and data integration in a federated database system. However, traditional data modeling techniques such as relational data model and semantic data model are not designed to support efficiently the definition, storage, and manipulation of complex meta-data. Investigations should be made on how the schema of the data model can be extended to support the meta-data, how this meta-data can enhance the semantics of a database schema, support data integration and mediation services, and facilitate query processing in a federated database system.

The work described in this dissertation provides three major contributions. The first is the introduction of the concept of dynamic meta-data and its specification for describing different types of data semantic information, in addition to the traditional static meta-data. The dynamic meta-data allows each instance object of a class to associate within itself its distinct meta-data information, whereas the static meta-data assigns the same meta-data information to all instance objects of a class.

The second contribution is the Mediation Data Model. The Mediation Data Model provides an extensible schema that supports the dynamic meta-data, and a framework for specifying mediator services within the data model. The Mediation Data Model also provides the association of multiple unit values to an instance of a data type. The concept of Multiple Unit Value (MUV) is implemented by means of the Unit-type Dynamic Meta-Data, the Unit Value Mediator and the extended DML. The MUV concept allows object attributes to have values expressed in convertible units.

The final contribution of this dissertation is the federated database system architecture, called InfoFED. InfoFED makes use of the Mediation Data Model as a common data model to provide the self-describing data as well as an intelligent data manipulation language to users, and to improve data integration by reducing the schema integration conflicts. The schema integration conflicts are reduced because the Mediation Data Model supports the MUV concept allowing heterogeneous databases to export their data in their own unit value without any unit conversion. The Unit Value Mediator provides translation and conversion services. The architecture also provides a browser for both data and meta-data, an object-oriented federated schema, and the data source registration and management.

1.4 Organization of the Dissertation

In Chapter 2, the background and related work is discussed. Briefly discussed in the presentation is the survey of the federated database system architectures and mediators. Chapter 3, explains the definitions of data and meta-data which is further characterized as dynamic and static. The specification of the data, meta-data, and the Property Knowledge Package to which a list of the dynamic meta-data for each data element is attached are described in this chapter.

To represent the real world, a data model may involve unit types. Since each current data model supports only one unit value for each attribute, Chapter 3 introduces the Unit Value mediator that provides the unit conversion services. Chapter 5 explains how the Property Knowledge Package can be incorporated into the data model, that is, the

Mediation Data Model. The discussion shows how the Unit Value mediator can be coordinated within the Mediation Data Model and uses the Unit-type Dynamic Meta-Data to support the Multiple Unit Value concept. Chapter 5 introduces the InfoFED Federated Database System architecture and discusses the role of the Mediation Data Model within the architecture.

As a proof of concept, the implementation of the Unit Value Mediator and the InfoFED Federated Database System prototypes are presented in chapter 6. Finally chapter 8 discusses conclusions and suggestions for the future research.

2. Background and Related Work

The related work can be defined into two major areas which are federated database systems and mediators. A federated database system provides the architecture for integrating information from autonomous, heterogeneous databases. The mediators provide domain specialization services which can be used to support the data integration and the query processing in the multidatabase system, i.e., the federated database system.

2.1 Federated Database Systems

Over the past several years, there have been a number of projects centered on solving the problems of accessing heterogeneous and autonomous databases. The following section briefly describes some of the research systems that use the federated database system approach in dealing with heterogeneous and autonomous database access.

2.1.1 DDTS (Distributed Database Testbed System)

DDTS has a single federated schema called the Global Representation Schema, which is expressed in the relational data model [Dwyer&Larson87]. The External Schema in DDTS is called as the Conceptual Schema represented in the Entity-Category-Relationship (ECR) model. Users formulate requests directly against the conceptual schema in the GORDAS query language.

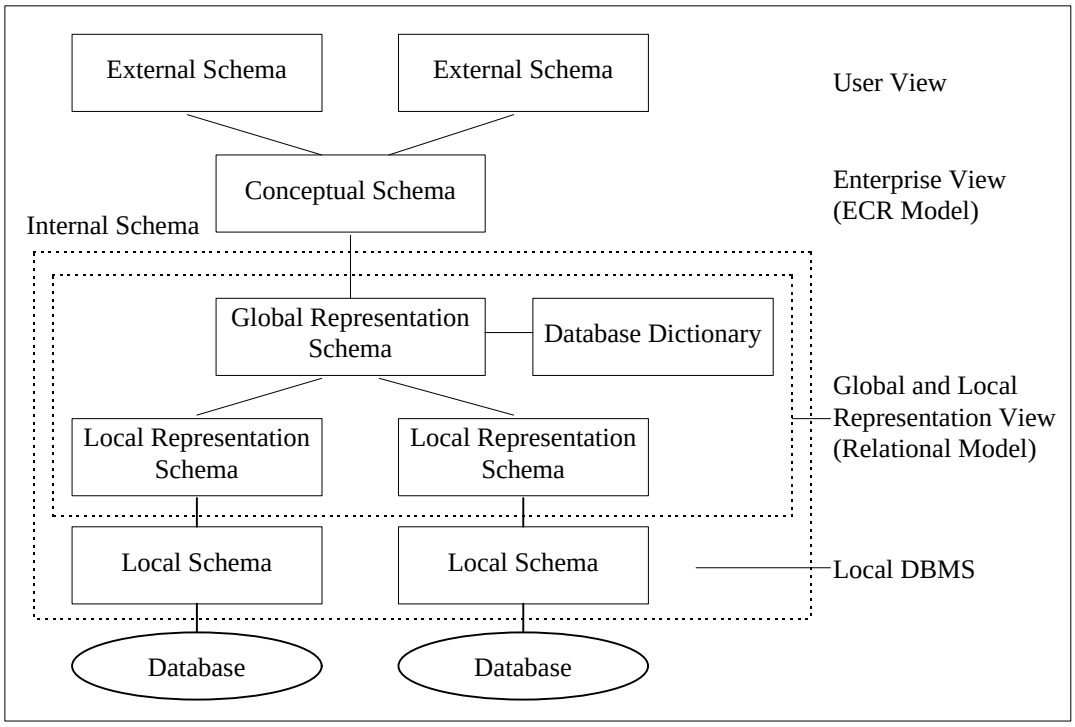


Figure 2-1 DDTS Architecture

2.1.2 MULTIBASE

Multibase [Thomas90] is a system for integrating access to pre-existing, heterogeneous, distributed databases. Users access the database system through a single global schema expressed in DAPLEX [Shipman81] which is a functional data model. Component DBMSs supported by Multibase include both CODASYL (Conference on Data System Language) and relational databases. A user submits a query to the system (with DAPLEX) over the global schema. The query translator translates the global query into a global query over the disjoint union of local schemas using information from the

Auxiliary schemas. The Auxiliary schema holds additional data not stored in any component DBMSs and information needed to resolve inconsistencies.

A Multibase prototype has been implemented in Ada. It executes on a VAX under the VMS operating system.

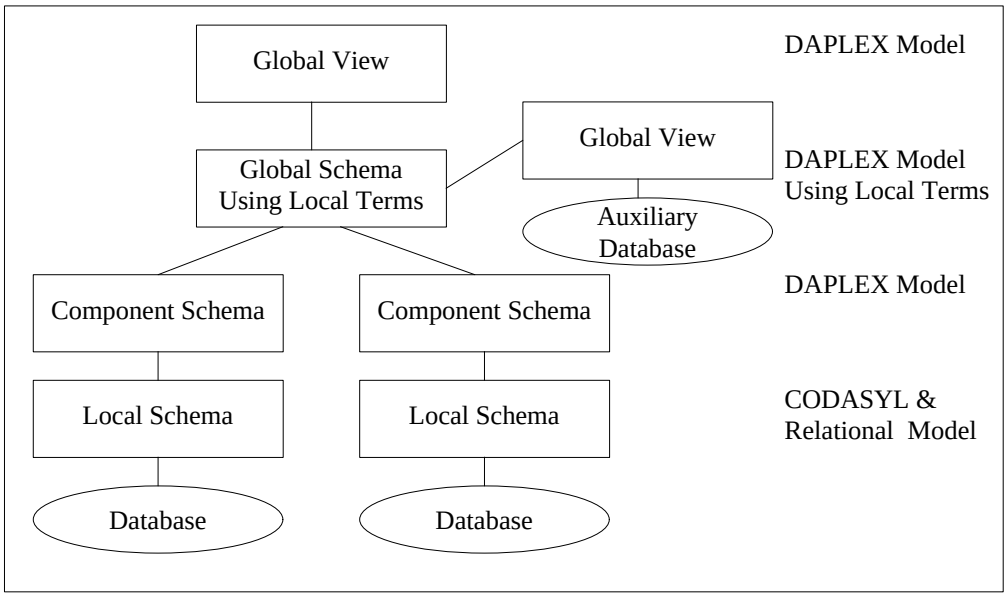


Figure 2-2 Multibase Architecture

2.1.3 MRDSM (Multics Relational Data Store Multidatabase)

The MRDSM [Litwin&Abdellatif86] Multidatabase architecture is a loosely-coupled federated database system and is based on interoperability among heterogeneous databases. All participating database systems retain autonomy and control over their data. MRDSM has no global schema and its general architecture is shown in Figure 2-3.

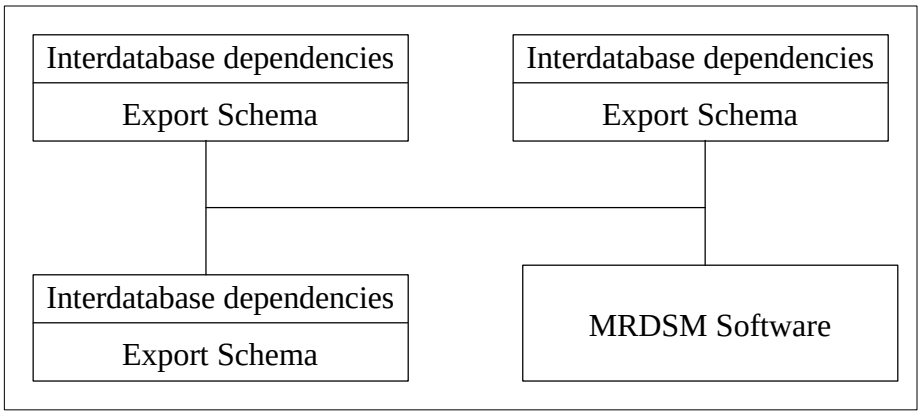


Figure 2-3 MRDSM Architecture

Databases become participants when their export schema is defined to MRDSM. An export schema may be a conceptual schema, a data model, or a database view schema. The goal of MRDSM is to allow users to formulate a query with a single statement by using the MRDSM data manipulation language, MDSL. Both retrieval and update operation are allowed. However, users must know the contents of the participating databases to formulate MDSL queries.

2.1.4 Alberta Land Related Information System (LRIS)

The LRIS [Goodman94] is a project that provides on line, query-only access to several sources of land related data. The data is stored and maintained independently by different agencies and departments throughout the province of Alberta. The LRIS is developed using the federated database system architecture as described in [Sheth&Larson90]. The LRIS consists of a four level schema architecture (Figure 2-4)

which includes the external schema, the federated schema, the component schema, and the local schema.

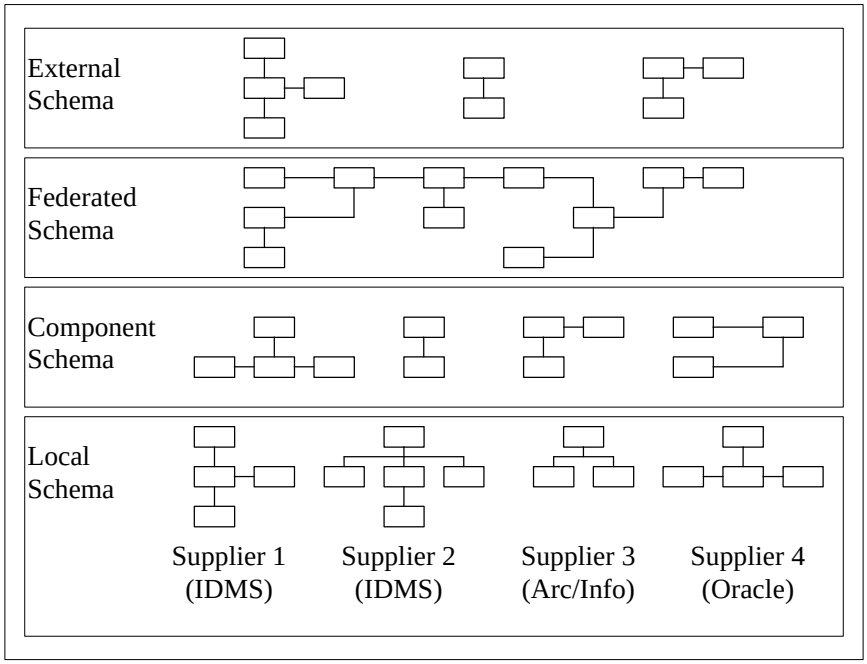


Figure 2-4 LRIS Four Level Schema Architecture

The initial project combines four data sources of land related data into a single federated database; two IDMS databases, one Oracle database and a spatial database implemented using Arc/Info and Oracle. The common data model is based on the relational data model, with extensions to include spatial data types such as POINTS, POLYGONS, and POLYLINES. It supports spatial query operators, such as OVERLAPS, WITHIN, and BUFFER, to allow constraints to be applied against these spatial data types.

2.1.5 FEderated MUltilingual System (FEMUS)

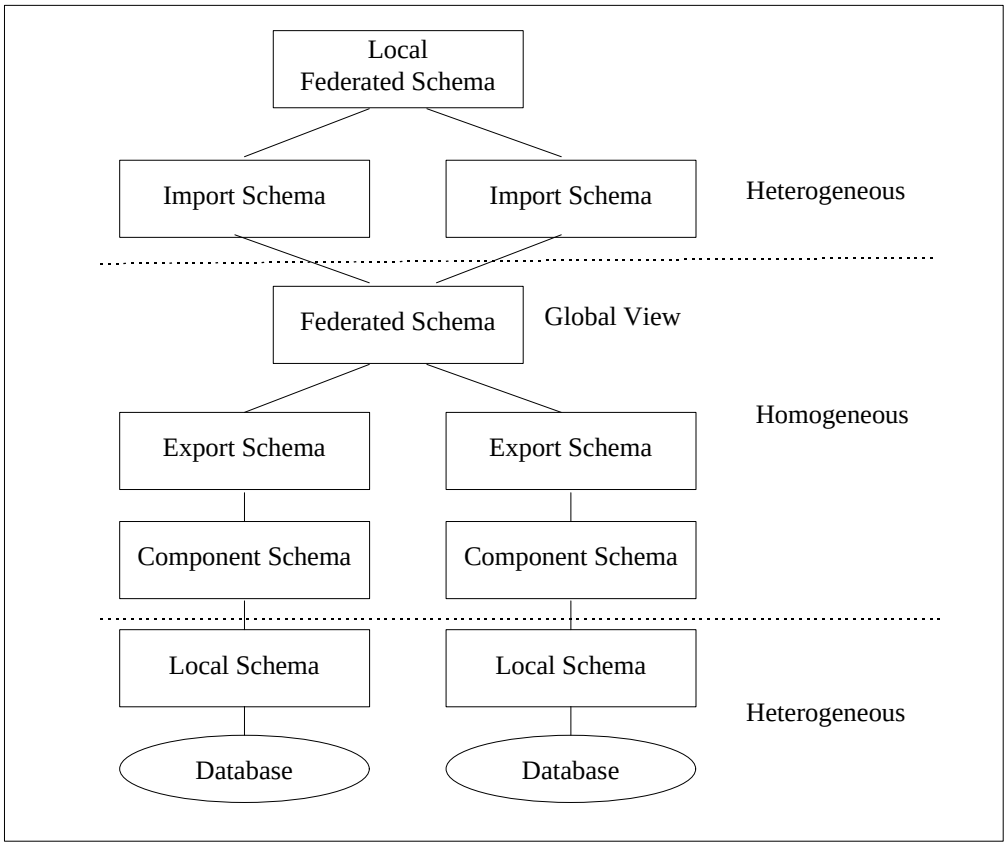


Figure 2-5 The FEMUS Architecture

FEMUS [ADSYTY93] has a six-level architecture as in Figure 2-5 compared to the five-level architecture of Figure 1-1. The additional level is the local integration of federated schema. FEMUS supports two different database approaches: one is based on semantic data model (ERC+), and the other is based on an object-oriented model (COCOON [Scholl&Schek92]).

2.2 Mediator

A mediator is a software module that exploits encoded knowledge about certain sets or subsets of data to create information for a higher layer of application. One example of a mediator is the "context mediator" proposed in [GMS94, and SSR94].

2.2.1 Context Mediator

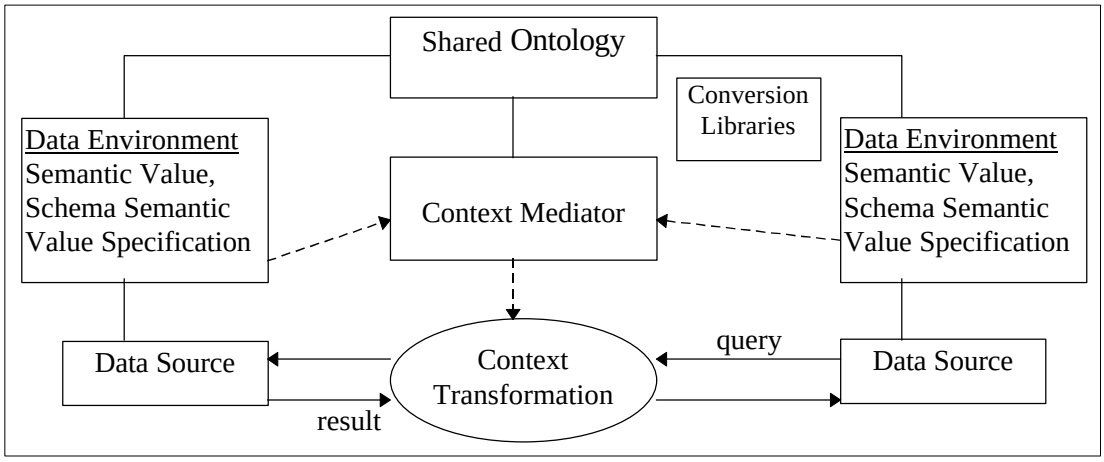


Figure 2-6 An Architecture for the Semantic Interoperability using Semantic Values

Each information system component may have an associated data environment. A data environment contains two parts. The semantic value schema specifies attributes and properties, and the semantic value specification specifies values for some or all of these properties. The context mediator uses the data environments to determine whether a requested data exchange is possible and, if so, to determine necessary conversions. The shared ontology component specifies terminology mappings. These mappings describe

the naming equivalence among the component information systems. The conversion library contains conversion functions.

2.3 Conclusion

The systems reviewed in the preceding sections are all designed to address the problems of integrating existing database systems. The federated database system architecture provides a framework for integrating the information from heterogeneous data sources, where mediators provide semantic conflict resolution services among different data sources. When the number of data sources increases, the semantic incompatibility problems become more difficult to solve. The mediator, which provides semantic conflict resolution services, can enhance the information integration process in the federated database system. However, all of the reviewed federated database systems do not incorporate mediators into their architecture. This dissertation presents such an approach.

Among the problems of integrating mediators to the federated database system architecture is the lack of semantic representation of the data domains in the data model. Mediators require such semantic information as an input, in addition to the actual data elements to process the information [Gattorna96]. The reviewed federated database systems use either the relational data model, a semantic data model or an object oriented data model as their global data model. The semantic data model and the object oriented data model provide a better framework to model the real world enterprise than the

relational data model. However, these data models were not designed specifically to incorporate semantic information about the data domains.

In [SSR94], the semantic information of the data domain is provided by each DBMS as the data environment for processing by the mediators. This approach limits the capabilities of the mediator. The dissertation shows that, with mediators tightly integrated into the data model architecture, they not only provide a basic conflict resolution function but also enhance the data model with additional system capabilities. These extensible system capabilities enhance the data integration process in a federated database system.

3. Meta-Data and Property Knowledge Package

3.1 Definition of Data, Meta-Data, and Property Knowledge Package

This section discusses the definition of data, meta-data, and the Property Knowledge Package, together with their specifications. **Data** is a collection of facts, whereas **meta-data** is the data that describes data, that is, data about data. Meta-data is an abstraction of data and is higher-level data that describes lower-level data [APT96]. The meta-data complements data with knowledge, which provides the information for applications so that they can understand and use the data. Without the meta-data to provide context and usage information, data becomes unusable. In the early days of commercial computing, each application created and handled its own data files. Because the files' meta-data was embedded in the application's data definitions, no application could make sense of another application's files. Since databases were introduced, they have brought the concept of storing the meta-data in the database catalog, not in the individual programs, therefore allowing different applications to access to the same database. A database uses its schema to describe or structure the real word concepts as classes and relationships among classes. This schema information can be considered as the meta-data for instances of classes which are defined within the data model. Two types

of meta-data that are commonly found in a data model are meta-data for properties of classes, and meta-data for instance objects.

Meta-data for classes

Superclass = (Person, Employee)

Subclass = (Employee, Person)

Schema definition (the meta-data for instance objects) for “Person” and “Employee” classes

Person = (Name String[10])

Employee = (Name String[10], Salary Number[5])

Instances (fact)

Employee = (John, 2500)

Employee = (Peter, 3000)

Figure 3-1 Example of Meta-Data

In the data model, the meta-data describes the data of the level below. The meta-data for classes describes the information about classes. From the example in Figure 3-1, the meta-data for classes contains the information about superclass and subclass relationships between the class ‘Person’ and the class ‘Employee’. Thus, ‘Person’ is a superclass of ‘Employee’, and conversely, ‘Employee’ is a subclass of ‘Person’. In the same way, the meta-data for instance objects describes the information about instance objects. From the example, there are two instance objects of the class ‘Employee’ which has two attributes, i.e. ‘Name’ and ‘Salary’. The first instance object has the name as ‘John’ and has the salary as ‘2500’ whereas another instance object has the name as

‘Peter’ and has the salary as ‘3000’. Most of existing works emphasize the extension of the meta-data for classes. Different kinds of the meta-data for classes, such as a class thesaurus, a class antonym etc., are described in [Weishar93]. The class thesaurus supports different names which refer to the same class. This dissertation emphasizes on the meta-data for instance objects which can be categorized as static or dynamic.

Static Meta-Data (SMD) is the meta-data that is fixed for all instances of the same class. An example of the Static Meta-Data is a traditional schema definition, i.e., class names, attribute names, data types, etc. From the example in Figure 3-2, the attribute information is part of the schema definition and is static meta-data. For all data elements, i.e, ‘2500’ and ‘3000’, of the same class, the attribute names are defined as ‘Salary’.

All of existing data models support static meta-data. Most of them, however, do not support the dynamic meta-data.

Dynamic Meta-Data (DMD) is the meta-data that can be veried for each instance of the same class. Examples of the Dynamic Meta-Data are the attributes unit type (e.g., a unit type of the attribute ‘Salary’), security (e.g., classify, unclassify, etc.), data source location, etc. This DMD assists users in understanding the actual context of data and provides the necessary information that is needed by mediators for their processes.

Property Knowledge Package is a package of differentiated and specialized data descriptions, i.e., it contains a list of well defined Dynamic Meta-Data. The Property Knowledge Package provides the additional meta-data which augments the schema

information of each data element. It describes the meaning of the data element to which the knowledge is attached, in addition to the information defined by the schema.

Static Meta-Data Schema	**Data** Data Element
Attribute Salary	2500
Attribute Salary	3000

a. Information which is described by schema and data element.

Static Meta-Data Schema	**Data** Data Element (Attribute.Value)	**Dynamic Meta-Data** Property Knowledge Package (Attribute.Property)
Attribute Salary	2500	: **Unit-type** Currency USD 1 Time-range Month -1 : **Source** URL http://gmu.edu/db1.cgi : : **Security** Class Unclassify :
Attribute Salary	3000	: **Unit-type** Currency CND 1 Time-range Week -1 : **Source** URL http://gmu.edu/db2.cgi : : **Security** Class Classify :

b. Information which is described by schema, data element, and property knowledge package.

Figure 3-2 Knowledge Representation by the Data Model

Existing data models describe real world concepts through the schema (static meta-data) and data elements as shown in Figure 3-2a. This knowledge alone is sometimes insufficient to represent the actual meaning of the real world concepts, especially in an interoperable environment. As the example in Figure 3-2a shows, the attribute salary has the value of ‘2500’. It is unclear what the unit value of this ‘2500’ is

(what are the currency and periodicity of the salary). Users must have some background knowledge about the underlying meaning (e.g., attribute unit value, data source, etc.) of the data's schema in order to understand the actual meaning of the data element. Without this meta-data, automated conflict resolution becomes more difficult in multidatabase environment, where the data integration is required for the data from different schemas that may have different semantic meaning (i.e., unit value). Users also require the explanation information about the data more in the multidatabase environment than in a single database system. For instance, users may want to know which source provides the data in addition to the data definition.

In Figure 3-2b, the proposed concept describes the real world through the schema definition (static meta-data), the data element, and the Property Knowledge Package (dynamic meta-data). The Property Knowledge Package contains necessary information that makes the data element explicit in the interoperable environment. The Property Knowledge Package contains different types of the Dynamic Meta-Data (DMD) such as attribute unit type, data source, etc. From the example in Figure 3-2b, the attribute 'Salary' has the value of '2500'. The unit value of this 2500 is 'US_Dollar/month' ($USD^1 * Month^{-1}$) and the data value (2500) is obtained from the URL address of 'http://gmu.edu/db1.cgi'. The security property of the data is 'unclassify' meaning that the data is accessible to unclassified or higher users in a security lattice.

The DMD also allows the data schema to describe the domain of class attribute in a broader meaning. For example, in defining an attribute salary, if the schema supports only the Static Meta-Data, one fixed unit value such as 'US_Dollar/Month' must be

assigned to the attribute salary unit type. With the DMD, the attribute salary unit type can be defined in a general meaning such as in ‘Currency per Time-range’ and supports all different unit values (ex. ‘US_Dollar/Month’, ‘Canadian_Dollar/Week’, Thai_Baht/Day, etc.). This is preferable in the real world environment.

3.2 Knowledge Specifications for Property Knowledge Package and Dynamic Meta-data

A Property Knowledge Package is a package of differentiated and specialized data descriptions, i.e., it contains a list of well-defined Dynamic Meta-Data (DMD). Each DMD describes a specific type of data description (meta-data). The following sections explain their specifications and components.

3.2.1 Property Knowledge Package specification

The syntax for the Property Knowledge Package is as follows:

Syntax:

Property Knowledge Package ::= : <DMD_1>: <DMD_2>: ... : <DMD_i>:

Where <DMD_i> is a Dynamic Meta-Data specification type *i*.

The Property Knowledge Package specification contains a list of different types of the DMD. In the package, the colon symbol (‘:’) denotes the start and the end of the expression and the separation of the DMD in the list.

Figure 3-3 is an example of the Property Knowledge Package which contains two DMD types: 1) unit-type DMD and 2) data source DMD. From the example, the Property

Knowledge Package indicates that the data has the unit type of ‘Currency per Time-range’, has the unit value of ‘Canadian Dollar per Month’, and is obtained from the URL address at ‘http://isse.gmu.edu/db.cgi’. The details on how to interpret each DMD will be discussed in the next section.

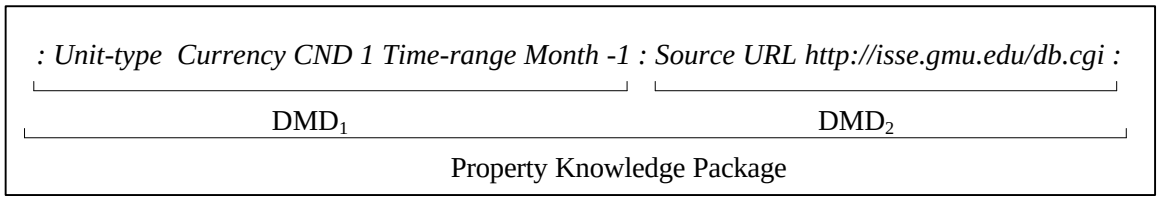


Figure 3-3 Example of Property Knowledge Package

3.2.2 Dynamic Meta-Data Specification.

The syntax for the Dynamic Meta-Data is as follows:

Syntax:

DMD ::= <DMD-Header> <DMD-Specification>

Where <DMD-Header> ∈ Supported-DMD-type
Where Supported-DMD-type is a set of dynamic meta-data types that are supported in the Property Knowledge Package
<DMD-Specification> is a data specification.

Each DMD is composed of two sections: the header section (<DMD-Header>) and the data specification section (<DMD-Specification>). The header section is a string that represents the type of the DMD, ex. Unit-type and Source. The <DMD-Header> must be a member in the *Supported-DMD-type* set. The Supported-DMD-type set contains a

finite set of string members, each of which represents the DMD type that is supported in the Property Knowledge Package. The data specification section contains the information that describes the DMD type that is stated in the header section. Each DMD type has a different specification that is specialized for its type. Unit-type DMD and Data-Source DMD are described here as a proof of concept. Therefore,

Supported-DMD-type = {Unit-type, Source}

Where the first member 'Unit-type' represents Unit-type DMD and the 'Source' represents Data-Source DMD.

Property Knowledge Package is designed to be extensible to support new DMD types. Each DMD type may play a different role. Some DMD types, such as the Data-source DMD, are intended only for the data interpretation. Others, such as the Unit-type DMD, are intended for interpreting the data and supporting a new system capability in the data model (Multiple Unit Value concept). Different DMD types such as security, temporal, data quality, etc. may be added to the Property Knowledge Package. This additional knowledge will augment the data model to represent more complex types of information and support new system capabilities.

3.2.2.1 Unit-Type Dynamic Meta-data

The Unit-type DMD has been implemented in the prototype since the Unit-type DMD is the most basic semantic information in the Property Knowledge Package. The Unit-type DMD provides the information about the unit type and the unit value of the data that the knowledge is attached to. Examples of unit type are 'Currency', 'Weight',

'Time-range', etc. Examples of unit value of 'Time-range' unit type are 'Hour', 'Day', 'Week', 'Month', etc. Unit type can be a combination of more than one unit types. For example, the salary unit type is a combination of 'Currency' unit type and 'Time-range' unit type, i.e., '$Currency^{1}$*$Time\text{-}range^{-1}$'. With the Unit-type DMD, the data model can be extended to support the Multiple Unit Value concept (Discussed in Chapter 3). The Multiple Unit Value concept allows the instance of each attribute to be stored, retrieved, and manipulated in more than one unit value which is preferable in most applications.

The syntax of the Unit-type DMD contains a list of at least one unit type and a conversion information set at the end of the list. The syntax for the Unit-type DMD can be stated as follows:

Syntax:

Unit-type DMD ::= <Unit-type-DMD-Header> <Unit-type-DMD-Specification>

Where

<Unit-type-DMD-Header> ::= Unit-type

<Unit-type-DMD-Specification> ::= <Unit-set>+ [<Conversion-info-set>] | None

<Unit-set> ::= *t v* ±1

where *t* is a type of the unit.

$t \in$ Supported-Unit-type

where Supported-Unit-type is a set of unit types that are supported by the system.

v is a value of the unit.

$v_i \in$ Supported-Unit-value

where Supported-Unit-value is a set of unit values for a specific unit type that are supported by the system.

±1 indicates the power of *t* (+1 means t^{1} where -1 means t^{-1}) and *v* (+1 means v^{1} where -1 means v^{-1})

<Conversion-info-set> ::= **by** <oid>+

where <Conversion-info-set> is a special section that contains the information on how the data is converted form its original unit value.

<oid> is the object identification of the object that is involved in converting the data.

A full format of Unit-type DMD can be expressed as follows:

Syntax:

Unit-type DMD = **Unit-type** t_1 v_1 ±1 t_2 v_2 ±1 ... t_n v_n ±1 [**by** oid_1 … oid_n]

The Unit-type DMD specification contains a list of different types of the <Unit-set>. The <Unit-Set> contains the information about the data unit type and the data unit value. Each <Unit-set> is composed of three sections: the *unit type (t)*, *unit value (v)*, and the power indicator (±1).

The *unit type* is a string that represents the type of the unit, ex. Currency, and Weight. The *unit type* must be a member of the *Supported-Unit-type* set. The *Supported-Unit-type* set contains a finite set of string members, each of which represents the type of the unit that is supported by the system.

The *unit value* is a string that represents the value of the unit for a type of the unit that is stated in the *unit type* section, ex. the unit value ‘Kilogram’, ‘Gram’, and ‘Pound’ for the unit type ‘Weight’. The *unit value* must be a member of the *Supported-Unit-value* set of the unit type that is stated in the *unit type* section. For each unit type, there is a *Supported-Unit-value* set that contains a finite set of string members each of which represents the value of the unit that supports that unit type.

For example, let

Supported-Unit-type[1] = {Currency, Time-range, Length}

From the example, there are three unit types that are supported by the representation: the currency ('Currency'), the time-range ('Time-range'), and the length ('Length'). For each supported unit type, there must be a *Supported-Unit-value* set that defines all the supported unit values for that unit type.

From the example, the *Supported-Unit-value* sets for the currency unit type, the time-range unit type, and the length unit type are respectively presented below:

Currency-Supported-Unit-value = {usd, cnd, frf, jpy, dem, thb}
Time-range-Supported-Unit-value = {second, minute, hour, day, week, month, year}
Length-Supported-Unit-value = {millimeter, centimeter, meter, kilometer, inch, foot, yard, rod, mile}

The power value, '±1', of the unit type and the unit value in the *Unit-set* indicates the multiplication relation among the unit types or the unit values in each of *Unit-set*s. The value '1' of the power designates the variable as a multiplier whereas the value '-1' of the power designates it as a denominator.

Therefore, the final unit type is the product of the multiplication (or division) of all the unit types from each <Unit-set>. From the syntax of the Unit-type DMD, the interpretation for the unit type is:

$$t_1^{\pm 1} * t_2^{\pm 1} * \ldots t_n^{\pm 1}$$

[1] Four unit types which are 'Currency', 'Length', 'Time-range', and 'Weight' are supported and implemented in the study (See Appendix A).

Likewise, the final unit value is the product of the multiplication (or divide) of all the unit values from each <Unit-set>. From the syntax of the Unit-type DMD, the interpretation for the unit value is:

$$v_1^{\pm 1} * v_2^{\pm 1} * \ldots v_n^{\pm 1}$$

The <Conversion-info-set> is a special section that contains the information on how the data is converted from its original unit value. This information contains the list of objects that are involved in the data conversion process. The converted data is converted by using following objects.

$$oid_1, oid_2, \ldots oid_n$$

The following is an example of a unit-type DMD.

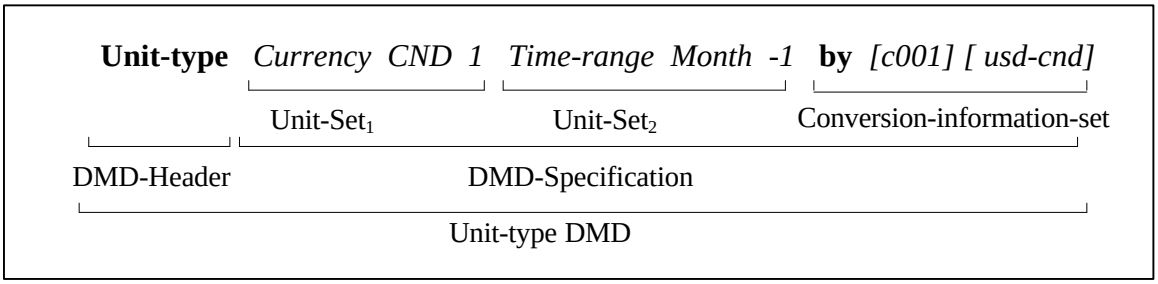


Figure 3-4 Example of Unit-type Dynamic Meta-Data

In this example, the final unit type expressed in the Unit-type DMD is a compound of two basic unit types: ‘Currency’ from <Unit-Set$_1$> and ‘Time-range’ from

<Unit-Set$_2$>. The unit type of this example is 'Currency/Time-range' ('Currency^{1}*Time-range^{-1}') and the unit value is 'Canadian_Dollar/Month' ('CND^{1}*Month^{-1}'). The information also indicates that the data has been converted to this unit type value by using '[C001]' and '[usd-cnd]' objects (discussed in section 5.1.1.2).

3.2.2.2 Data Source Dynamic Meta-data

The Data Source DMD contains the information of the sources of the data. The syntax for the Data Source DMD can be written as follows:

Syntax:

Data-source DMD ::= <Source-DMD-Header> <Source-DMD-Specification>

Where

<Source-DMD-Header> ::= Source.
<Source-DMD-Specification> ::= URL <url>
<url> is an Uniform Resource Locator.

Uniform Resource Locators (URLs) are a scheme for specifying Internet resources on the World Wide Web using a single line of printable ASCII characters [Graham95]. A typical URL composes of three different parts as shown in the below example:

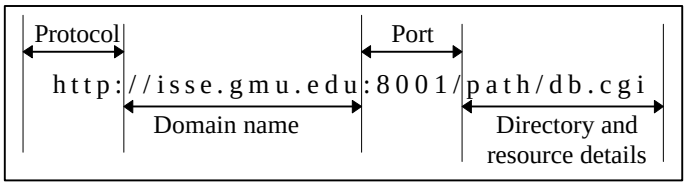


Figure 3-5 Example of the URL

1.) Protocol. The first string in the URL specifies the Internet protocol to use in accessing the resource. URL schemes are defined for most Internet protocols, including FTP, Gopher, HTTP, etc.

2.) Address and Port Number. The second part of this URL is the Internet address of the server; this information lies between the double forward slash (//) and a terminating forward slash (/). This example gives the domain name of the server and the port number to contact.

3.) Resource Location. The last section is the path information required to locate the resource on the server. Often, this resembles a directory path leading down to a file or a common gateway interface (CGI) program.

The following is an example of the Data Source DMD.

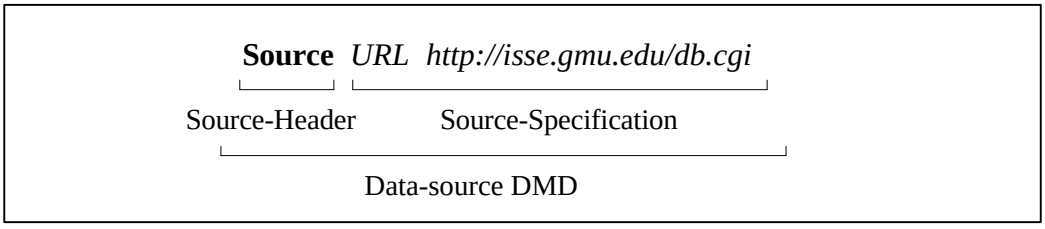


Figure 3-6 Example of Data Source Dynamic Meta-Data Knowledge

Each data source has its unique URL address. The Data Source DMD helps users to distinguish among different sources of the same data in the interoperable environment. The Data Source DMD allows users to filter the retrieved data based on data source

grouping. Data sources could be grouped by the data source domain (e.g., edu, mil, gov, or, etc.) or data quality (e.g., Class A, Class B, etc.) that is graded by creditable agents.

4. Unit Value Mediator

The Unit Value mediator (UV mediator) is a mediator that provides unit conversion services. The architecture and its properties are described in this chapter. The next chapter will discuss how the UV mediator can be incorporated within the data model to support Multiple Unit Value concept which enhances both data processing and data integration in a federated database system.

4.1 Unit Value Mediator Architecture

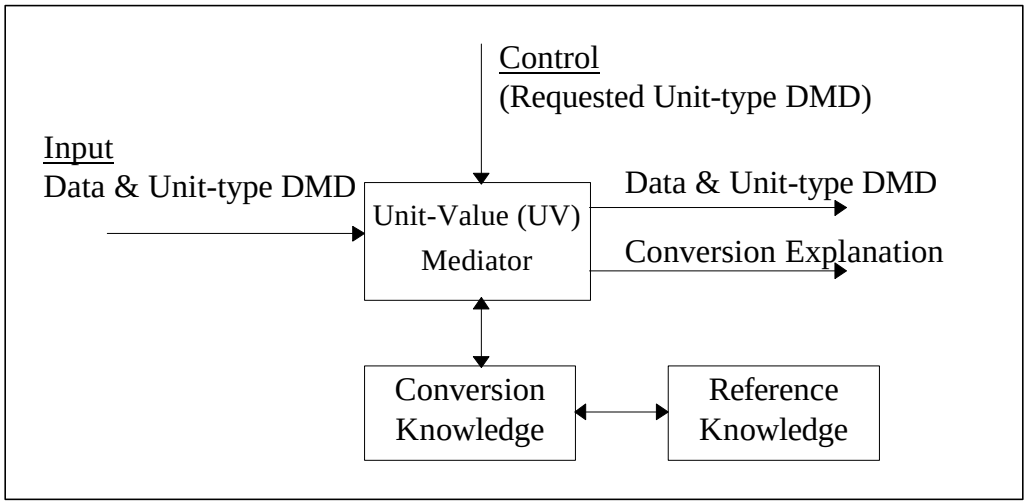


Figure 4-1 Unit Value Mediator Architecture

The Unit Value Mediator is modeled using the Service Description Diagram which was proposed in the referenced architecture for the intelligent integration of information [AHKS95]. The Unit Value Mediator architecture consists of the following components:

1.) Data & Unit-type DMD. Data is a fact, whereas Unit-type DMD contains information about the unit type and unit value of the data (explained in the previous section).

2.) Unit Value Mediator. The Unit Value Mediator (UV Mediator) is a special agent that is responsible for exchanging the unit values.

UV-Mediator(a, u, u′) = (a′, u″)

The UV Mediator takes the data (a) and data's Unit-type DMD (u), and a requested Unit-type DMD (u′) as its input. Users control the output by assigning the desired unit value to the requested Unit-type DMD (u′). The UV Mediator converts the data from its unit value to the requested unit value and returns a converted data (a′) and a converted Unit type DMD (u″) as its output.

The UV mediator itself does not contain the knowledge about how to convert the data. To support conversions among different unit values within the unit type, the system must provide the conversion knowledge and the necessary reference knowledge to support that unit type. When the UV mediator detects the unit value conflict, it assigns an appropriate conversion function, as the Conversion Knowledge object, and a conversion rate, as the Reference Knowledge object, to solve the conflict. For example, to support

conversions among different unit values (usd, cnd, thb, jpy, etc.) within the currency unit type, the system provides a currency unit type conversion function and necessary currency exchange rates.

3.) Unit-type Conversion Knowledge and Reference Knowledge. The UV Mediator requires two kinds of knowledge, Unit-type Conversion Knowledge and Reference Knowledge, to support the unit value conversion. The Unit-type Conversion Knowledge is a collection of conversion objects. Each conversion object contains the meta-data of the conversion function. The conversion function contains the knowledge of how the data can be converted to a specific unit type. Examples of the Unit-type Conversion Knowledge are the currency unit type conversion and the weight unit type conversion.

To support each conversion, the conversion function requests conversion rate information from the Reference Knowledge. The Reference knowledge is a collection of reference objects, each of which contains the fact (i.e., conversion rate, etc.) that will be used by conversion objects to perform a unit conversion. An example of a reference object that is returned from the Reference Knowledge is the currency exchange rate. The knowledge specification for the Conversion Knowledge and the Reference Knowledge will be discussed in Chapter 5.

4.2 Unit Value Conversion Characteristics.

Let

Unit-type Conversion function = f(a,v,v′,±1) = a′

where f is a unit-type conversion function converting the value ‘a’ of unit value ‘v’ to the value ‘a′’ of unit value ‘v′’. The ‘±1’ guides the conversion function to perform a multiplication (+1) or division (-1) operation which reflects the role of the unit in the unit type. The ‘±1’ allows the functions to perform the inverse calculation which allows the function to convert the data back to its original value (i.e., a = f(f(a,v,v′,1),v,v′,-1).

For example,

Weight-unit-type-f(10, kg, pound,1) = 22

is a weight unit type conversion function that converts a value ‘10’ of the unit value ‘kg’ to a value ‘22’ of the unit value ‘pound’. The value can be inverted back to the original value as:

Weight-unit-type-f(22, kg, pound,-1) = 10

There are certain conversion properties that are required for the unit-type conversion functions. These properties ensure that data can be converted to any unit value in a given unit type domain. They also ensure that the result of relational operations (e.g., >, ≥, <, ≤, =, etc.) in the data manipulation language that performed on any two converted data will yield the same result for any unit value. These conversion properties are total, lossless and order-preserving [SSR94].

A total conversion is one in which all conversion functions are defined for all arguments. This restriction ensures that the data value can be converted to any unit value

in a given supported unit-value domain. An example of a non-total property is a currency conversion since some of exchange rate values may not be available to support converting data to some currency unit value. But within a given currency unit type domain, we can force the currency conversion to have a total property.

A lossless conversion is one to which it makes no difference whether a value is converted from one property value to another directly or sequentially. Formally, f is lossless if $f(a,v_1,v_1,\pm1) = a$, and $f(a,v_1,v_3,\pm1) = f(f(a,v_1,v_2,\pm1),v_2,v_3,\pm1)$ for any values of a, v_1, v_2, and u_3. In the case where $v_1 = v_3$, the conversion function converts values from one unit type to another and back again, resulting in the original value. For example in the 'Length' unit type, the data value 'a' of the unit value 'meter' is converted to the unit value 'mile' and then is converted back to the unit value 'meter' must return the same value as 'a'. An example of non-lossless conversion is a document type conversion function. Consider converting a document from a document type 'Word 7' to 'Text' and converting it back to a document type 'Word 7'. The conversion from document type 'Word 7' to 'Text' will lose its original text format supported by 'Word 7', and it is obvious that converting back to a document type 'Word 7' will not obtain the same original document, i.e., $a \neq f(f(a,v_1,v_2,\pm1),v_2,v_1,\pm1)$.

An order-preserving conversion is one that does not change the order of any two values after converting them. Formally, f is order-preserving if $a > a'$ then $f(a,v,v',\pm1) > f(a',v,v',\pm1)$ for any values of v, a′, v, and v′. This restriction ensures that when the unit values are converted, they do not change their relative order. For example, if the data

value ‘a’ is greater than the data value ‘a′’ in the unit value ‘mile’, the data value ‘a’ will be greater than the data value ‘a′’ in any unit value within the unit type ‘Length, i.e. yard, mile, kilometer, meter, etc.

Examples of total, lossless, and order-preserving characteristics are a weight unit type conversion and a length unit type conversion.

4.3 Conversion Knowledge and Reference Knowledge Characteristics

Knowledge can be characterized as static and dynamic. Static knowledge is based on current human knowledge or strong belief that has been true for a long period of time. Examples are length conversion ratio, time range conversion ratio, etc. ‘Washington DC is the capital of the USA’ can be considered static since the fact is based on a human strong belief. Dynamic knowledge, on the other hand, changes regularly. An example is a currency exchange rate.

The Unit-type Conversion Knowledge is considered static. Although the system allows different conversion objects to perform on the same conversion type, these different conversion objects will yield the same conversion results in order to maintain conversion properties despite the differences in the algorithm they use. The Reference Knowledge, however, contains both dynamic and static knowledge. Examples of static Reference Knowledge are length conversion ratio, weight conversion ratio, etc. These facts do not change. An example of Reference Knowledge that is dynamic is the currency exchange rate, which changes overtime.

4.4 Scenario for the Unit Value Mediator

The system works in the following manner. The UV Mediator is an agent that is responsible for checking the data and its unit value against the requested unit value. If there are any conflicts or unit conversions are required, the UV Mediator requests appropriate conversion objects from the Unit-type Conversion Knowledge to solve each conflict. Conversion objects, in turn, may request additional information such as conversion rate from the Reference Knowledge. The UV Mediator performs the unit conversion and returns the converted data and its unit type along with the explanation of the conversion, the <Conversion-info-set> section in the Unit-type DMD. The <Conversion-info-set> contains a list of pairs of the Unit-type Conversion Knowledge object and the Reference Knowledge object that are used for converting the data.

The conversion explanation in the <Conversion-info-set> becomes more important especially when Reference Knowledge objects that are dynamic are involved. If the conversion involves any dynamic objects, the validity of the converted data will depend on these dynamic objects. For example, currency unit conversion may require the currency exchange rate reference knowledge for its conversion. Since currency exchange rate objects are dynamic and usually change daily, the converted data using these currency exchange rate objects will be invalid in the next day as these currency exchange rate objects change their values. The <Conversion-info-set> informs users about the change of the data and allows users to convert the data back to it original unit value.

For example, let

a = 36000
u = ‘Unit-type Currency usd 1 Time-range year -1’
u′ = ‘Unit-type Currency thb 1 Time-range month -1’

UV-Mediator (36000, ‘Unit-type Currency usd 1 Time-range year -1’,
‘Unit-type Currency thb 1 Time-range month -1’)
= (76110, ‘Unit-type Currency thb 1 Time-range month -1 by [c001] [usd-thb] [t001] [year-month]’)

Note: usd = US Dollar
thb = Thai Baht

The UV Mediator converts data from the data values ‘36000’ with the Unit-type DMD of ‘Unit-type Currency usd 1 Time-range year -1’ to the requested Unit-type DMD of ‘Unit-type Currency thb 1 Time-range month -1’.

The UV mediator detects two unit type conflicts: the ‘Currency’ unit type conflict and the ‘Time-range’ unit type conflict. The appropriate Conversion Knowledge objects and Reference Knowledge objects are called to solve these conflicts. The <conversion-info-set> in the output Unit-type DMD provides the information on how the data value is converted. Two pairs of the Conversion Knowledge object and the Reference Knowledge object, ([c001], [usd-thb]) and ([t001], [year-month]), are used to solve the ‘Currency’ unit type conflict and the ‘Time-range’ unit type conflict respectively. The Conversion Knowledge object ([c001], [t001]) contains the meta-data about the conversion function where the Reference Knowledge object ([usd-thb], [year-month]) provides the conversion rate that is used by the conversion function. From the knowledge specification in Figure 4-2, the function ‘c-convert’, from [c001]’s ‘Program.value’ attribute, and conversion rate

‘25.37’, from usd-thb’s ‘Rate.value’ attribute, are used to solve the ‘Currency’ unit type conflict. The function ‘t-convert’, from [t001]’s ‘Program.value’ attribute, and conversion rate ’12’, from year-month’s ‘Rate.value’ attribute, are used to solve the ‘Time-range’ unit type conflict.

Conversion Knowledge objects and Reference Knowledge objects provide not only the conversion function and the conversion rate, but also other useful information about the conversion function and the conversion rate. From the example, both the currency conversion function (c-convert) and the time-range conversion function (t-convert) are specified by the programmer named ‘Joe Smith’. The currency exchange rate ([usd-thb]) has a version dated ‘12/18/96’. The currency exchange rate is the type of Reference Knowledge that is dynamic. The converted data value ‘76110’ will become invalid as soon as the currency exchange rate ([usd-thb]) version dated ‘12/18/96’ expires; that is, the new version dated of the currency exchange rate value is defined as a new day begins. Although the converted data has become invalid, Conversion and Reference Knowledge objects, in the <conversion-info-set>, provide the necessary information for converting the data back to its original value so that it becomes valid again.

Unit-type DMD = 'Unit-type Currency thb 1 time-range month -1 by [c001] [usd-thb] [t001] [year-month]'
<conversion-info-set> = 'by [c001] [usd-thb] [t001] [year-month]'

Conversion Knowledge object 'c001'

([**c001**] of Currency
(Program.value **c-convert**)
(Program.prop ": Unit-type None :
Source URL http://isse.gmu.edu/wiput/query.cgi :")
(Designer.value Joe Smith)
(Designer.prop ": Unit-type None :
Source URL http://isse.gmu.edu/wiput/query.cgi:")
(Version.value "1/29/96")
(Version.prop ": Unit-type None :
Source URL http://isse.gmu.edu/wiput/query.cgi:")
)

Reference Knowledge object 'usd-thb'

([**usd-thb**] of Exchange-rate
(From.value usd)
(From.prop ": Unit-type None :
Source URL gopher://una.hh.lib.umich.edu/00/ebb/monetary/noonfx.frb:")
(To.value thb)
(To.prop ": Unit-type None :
Source URL gopher://una.hh.lib.umich.edu/00/ebb/monetary/noonfx.frb:")
(Rate.value **25.370000**)
(Rate.prop ": Unit-type None :
Source URL gopher://una.hh.lib.umich.edu/00/ebb/monetary/noonfx.frb:")
(Valid.value Y)
(Valid.prop ": Unit-type None :
Source URL gopher://una.hh.lib.umich.edu/00/ebb/monetary/noonfx.frb:")
(Version.value 12/18/96)
(Version.prop ": Unit-type None :
Source URL gopher://una.hh.lib.umich.edu/00/ebb/monetary/noonfx.frb:")
)

Conversion Knowledge object 't001'

([**t001**] of Time-range
(Program.value **t-convert**)
(Program.prop ": Unit-type None :
Source URL http://isse.gmu.edu/wiput/query.cgi:")
(Designer.value Jame Smith)
(Designer.prop ": Unit-type None :
Source URL http://isse.gmu.edu/wiput/query.cgi:")
(Version.value "1/29/96")
(Version.prop ": Unit-type None :
Source URL http://isse.gmu.edu/wiput/query.cgi:")
)

Reference Knowledge object 'year-month'

([**year-month**] of Time-range-info
(From.value year)
(From.prop ": Unit-type None : Source URL http://isse.gmu.edu/wiput/query.cgi:")
(To.value month)
(To.prop ": Unit-type None : Source URL http://isse.gmu.edu/wiput/query.cgi:")
(Rate.value **12**)
(Rate.prop ": Unit-type None : Source URL http://isse.gmu.edu/wiput/query.cgi:")
(Valid.value Y)
(Valid.prop ": Unit-type None : Source URL http://isse.gmu.edu/wiput/query.cgi:")
)

Figure 4-2 Examples of Conversion knowledge objects and Reference Knowledge objects

5. Mediation Database System

This section discusses the Mediation Database System components together with their data specification and the extended query facility that supports the system.

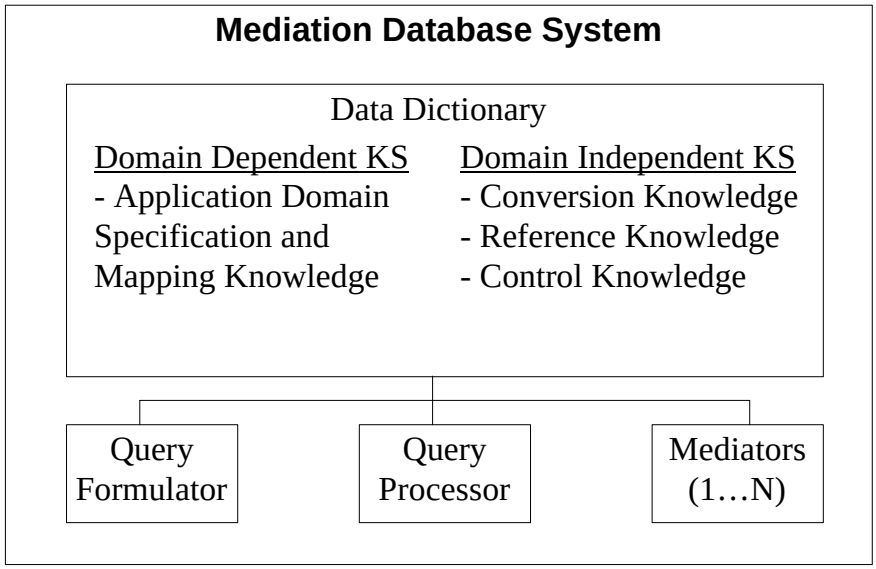


Figure 5-1 Mediation Database System Components

The Mediation Database System is a database system that uses the Mediation Data Model. The Mediation Data Model is an object-oriented data model that extends the schema to include dynamic meta-data as well as static meta-data. The Mediation Data Model incorporates mediator services to make use of dynamic meta-data to enhance the data model capabilities. It supports the basic object-oriented concepts such as the

classification (instance-of) relation, the generalization/specialization (is-a) relation, and the aggregation relation. The Mediation Data Model provides services and capabilities for defining the data, structuring the data, accessing the data, allowing reorganization (i.e., changes in data content, structure, and size), and supporting programming interfaces.

5.1 Mediation Database System Components

5.1.1 Data Dictionary

Data dictionary is a specialized type of database containing meta-data, which is managed by a data dictionary system. It is a repository of information describing the characteristics of data that is used to design, monitor, document, protect, and control data in information systems and databases. Data dictionary describes data through a schema definition. Normally the attribute definition, which is a part of the schema definition, stores only the data value. To improve the ability to represent the real world, the DMD is required to interpret the data. The Mediation Data Model incorporates the DMD information into the schema definition. The schema definition is extended by adding the property attribute to each object's value attribute. This property attribute contains the Property Knowledge Package, which in turn contains the DMD information. The expression for the attribute can be written as:

Attribute = (Attribute.value, Attribute.prop)

For each pair of value attribute and property attribute, the value attribute stores the actual data whereas the property attribute stores the Property Knowledge Package, that is, the DMD of the data element in its pair attribute. When users request a value from the Mediation Database System, a query of the Mediation Data Model is executed. The value of the object's attribute that the system returns to the users include both the value and its Property Knowledge Package, i.e., ‘Attribute.value’ and ‘Attribute.prop’.

Knowledge in the Data Dictionary can be classified into two groups, Domain Dependent Knowledge and Domain Independent Knowledge [Weisher93]. Domain-Dependent Knowledge represents the knowledge about application domains while Domain Independent Knowledge represents the control knowledge associated with unit conversion, general query formulation, query processing, etc.

5.1.1.1 Domain Dependent Knowledge

Domain Dependent Knowledge is used to define application domains. An ontology may be used to represent an application domain. Ontology [Gruber93, and Gruber&OlsenG94] is a shared specification of a conceptualization for a particular subject domain. It is important to recall that the Property Knowledge Package, in the ‘Attribute.prop’, is not intended to describe the full semantic definition of the real world concept. Each concept meaning must be clearly stated in the ontology definition. In the extensible data schema where the ontology is defined in the Domain Dependent Knowledge, its concept is enhanced by the information within the Property Knowledge Package. A concept definition (Class) does not have to be specific but can be stated with

more general definition. For example, the salary definition can be defined in a general unit type (‘Currency/Time-range’) instead of in one specific unit value (e.g. ‘US_Dollar/Month’).

An application domain is codified using the specification for an ‘Application’ class. This knowledge specification provides semantic information such as superclass, subclass, etc. [Weisher93]. The knowledge specification for ‘Application’ class contains the following information:

Application-object-class ::=
CLASS <name> **HAS**
SUPERCLASS (<Superclass-name>+)
SUBCLASS (<Subclass-name>+)
ATTRIBUTE (<value-property-attribute>+)
ATTRIBUTE-UNIT (: <attribute-unit-type :>+)
SYNONYM (<name>+)
DESCRIPTION (<description>)

Where
<value-property-attribute>::= <attribute-name.value> <domain>
<attribute-name.prop> <domain>
<domain> ::= string | number
<attribute-unit-type> ::= <attribute-name> <defined-unit-type>
<defined-unit-type> ::= Unit-type <unit>+ | NONE
<unit> ::= <unit-type> ±1
<unit-type> ∈ Supported-Unit-type set

The ‘Superclass’ attribute stores the parent class from which the current class inherits properties. Formally, a superclass C is a proper superset of a given class B (i.e., $B \subset C \Leftrightarrow \forall x\,[x \in B \Rightarrow x \in C]$ where $B \neq C$).

The ‘Subclass’ attribute stores the child classes which receive all properties from the current class. Again, formally, a subclass A is a proper subset of a given class B (i.e., $A \subset B \Leftrightarrow \forall x\ [x \in A \Rightarrow x \in B]$ where $A \neq B$).

The ‘Attribute’ attribute stores all value-property attributes that the class has. Each of the value-property attributes is composed of the value attribute and the property attribute (i.e., Value-Property attribute = (Attribute.value, Attribute.prop)). The value attribute (Attribute.value) contains the actual data where the property attribute (Attribute.prop) contains the Property Knowledge Package, i.e., the list of Dynamic DMDs, that describes the semantic information of the actual data in its pair value attribute.

The ‘Attribute-unit’ attribute stores the unit type information of each attribute in the class. The unit type information is a basic type of semantic information that describes properties of a class. However in most data models, it is not possible for a class to define unit type information. Characteristically, unit type information is associated with attributes that have domains that are numeric. Attributes with domain in the string expression normally do not require a unit type to be associated with them. Examples are the attribute ‘Salary’ of a class ‘Person’ and the attribute ‘Weight’ of a class ‘Aircraft’. Both have the domain in ‘Number’ and therefore have unit types as ‘Currency/Time-range’ and ‘Weight’ respectively. The attribute ‘Name’ and the attribute ‘Address’ of a class ‘Employee’ have a domain in ‘String’ and do not have a unit type. Not all attributes defined on numeric domains will have unit type information. For example, the attribute

'Number-of employee' of the class 'Company' has a domain in 'Number' but does not have a unit type. If the attribute has no unit-type, the value 'None' is assigned.

The 'Synonym' attribute defines the set of terms that are similar to the term that can be used to refer to the class. Formally, a term A is a synonym to a given term B (i.e., $A = B \Leftrightarrow \forall x\ [x \in A \Rightarrow x \in B]$). The synonym term concept allows the system to understand the different names or appearances that are equivalent semantically.

The 'Description' attribute defines the description of the class in free text which is useful to users who are browsing the application domain.

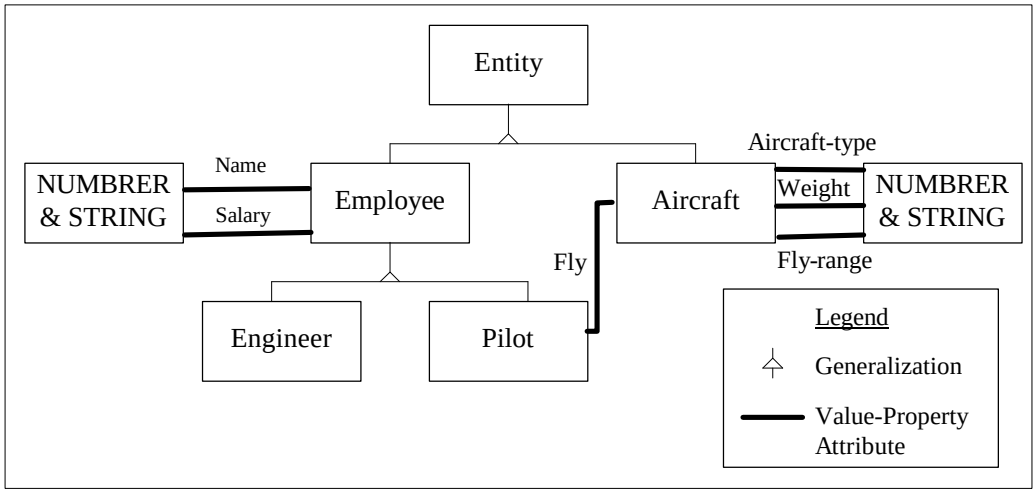


Figure 5-2 Example of Class Diagram for the Application Domain 'Airline'

Figure 5-2 shows the class diagram for an application domain 'Airline'. The specification for the 'Employee' class is shown as follows:

CLASS Employee **HAS**
SUPERCLASS (Entity)

SUBCLASS (Pilot Engineer)
ATTRIBUTE
Name.value string
Name.prop string
Salary.value number
Salary.prop string
ATTRIBUTE-UNIT (: Name Unit-type None :
Salary Unit-type Currency 1 Time-range -1 :)
SYNONYM (Worker)
DESCRIPTION (‘A person who works for another person’)

From the example, the ‘Employee’ class has an ‘Entity’ class as its superclass and has two subclasses, ‘Pilot’ and ‘Engineer’ classes. The ‘Employee’ class has two pairs of the value-property attribute which are (Name.value, Name.prop), and (Salary.value, Salary.prop). The ‘Name’ attribute has no unit type, whereas the ‘Salary’ attribute has a unit type as ‘Currency/Time-range’ ($Currency^{1}$* $Time\text{-}range^{-1}$). From the ‘Synonym’ attribute, the term ‘Worker’ is similar to the term ‘Employee’ and is also understood by the system.

The example instances for the ‘Employee’ class are presented in table format (Figure 5-3). Each attribute is expressed by a pair of value attribute and property attribute. The Property Knowledge Package provides additional information for interpreting the data with which it is associated. From the example, the employee named ‘Peter’ has the salary unit value in ‘US Dollar/Month’, whereas the employee named ‘Bob’ has the salary unit value in ‘Canadian Dollar/Month’.

ObjectID	Name.value	Name.prop	Salary.value	Salary.prop

[gen1]	Peter	: Source URL http://gmu.edu/db-1.cgi : Unit-type None :	2500	: Source URL http://gmu.edu/db-1.cgi : Unit-type Currency usd 1 Time-range month -1 :
[gen2]	Bob	: Source URL http://isse.gmu.edu/isse-db.cgi : Unit-type None :	2500	: Source URL http://isse.gmu.edu/isse-db.cgi : Unit-type Currency cnd 1 Time-range month -1 :
[gen3]	Bob	: Source URL http://gmu.edu/main-db.cgi : Unit-type None :	2200	: Source URL http://gmu.edu/main-db.cgi : Unit-type Currency cnd 1 Time-range month -1 :
[gen4]	Susan	: Source URL http://abc.com/db.cgi : Unit-type None :	50000	: Source URL http://abc.com/db.cgi : Unit-type Currency jpy 1 Time-range week -1 :
[gen5]	Martin	: Source URL http://abc.com/db.cgi : Unit-type None :	2000	: Source URL http://abc.com/db.cgi : Unit-type Currency thb 1 Time-range day -1 :

Figure 5-3 Example of Instances for the 'Employee' Class

The information in the property attribute also provides a simple method for resolving potential data conflicts. From the example, there are two employees who have the same name as ‘Bob’ but have different salary values, ‘2500’ and ‘2200’. Without knowing where the data came from, users hardly decides which data should be trusted. The Data-source DMD provides the data source information that helps users to distinguish among different sources of the same data. From the example, the first Bob’s salary (2500) has come from the data source’s URL address ‘http://isse.gmu.edu/isse-db.cgi’ and the second Bob’s salary (2200) has come from the data source’s URL address ‘http://gmu.edu/main-db.cgi’. Users then may decide to trust the first data, ‘2500’, since it is obtained from the department database (isse) which usually has more updated information than the main database of the university.

5.1.1.2 Domain Independent Knowledge

Domain Independent Knowledge represents the knowledge associated with unit conversion, general query formulation, query processing, etc. The knowledge is reusable by all applications. As mentioned in the previous chapter, two types of the knowledge that are associated with unit conversions and required by the Unit Value mediator are the Conversion Knowledge and the Reference Knowledge.

5.1.1.2.1 Conversion Knowledge

The Conversion Knowledge contains the knowledge about the unit type conversion. The Conversion Knowledge is organized as a class hierarchy (Figure 5-4) with a ‘Conversion’ class as a root and different subclasses for each supported unit conversion type; ex. ‘Currency’ class, ‘Time-range class, etc.

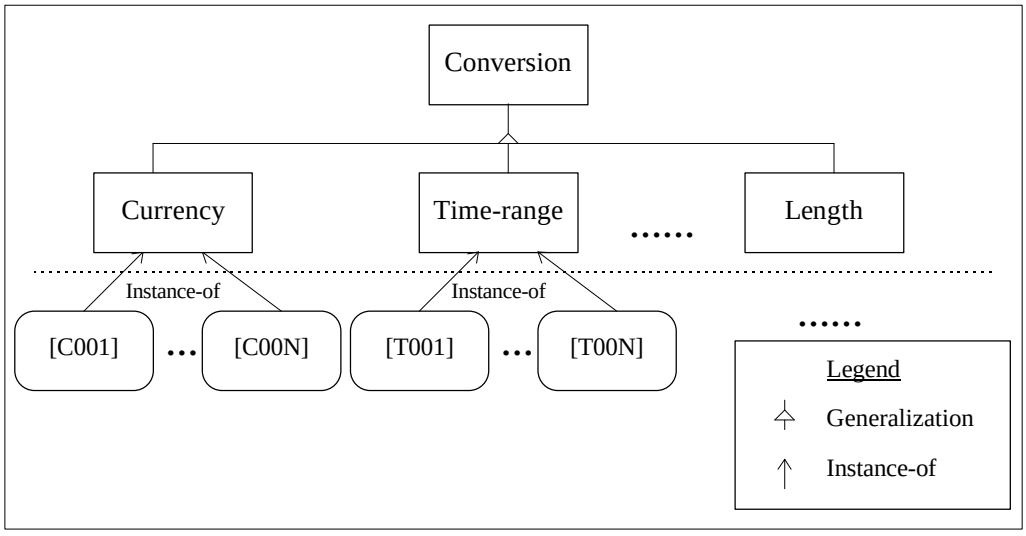


Figure 5-4 The Organization of the Conversion Knowledge

Each subclass contains a collection of unit-type conversion knowledge objects, each of which contains meta-data of a conversion function to support a unit type conversion that is referred to by that subclass. For example, the ‘Currency’ class has instance objects, ‘C001’, ‘C002’, …, ‘C00N’. Each instance object contains the meta-data for a specified currency-unit-type conversion function that can be used by the UV mediator if this instance object is called for services.

The Conversion Knowledge classes are codified using the knowledge specification for the ‘Conversion’ class, which contains the following information:

Conversion Class ::=
CLASS <name> **HAS**
SUPERCLASS (<Superclass-name>)
SUBCLASS (<Subclass-name>)
DEFAULT-CALL (<Instance Object’s id>)
DESCRIPTION (<description>)
ATTRIBUTE
Program.value string
Program.prop string
Designer.value string
Designer.prop string
Version.value string
Version.prop string

Where
<Instance Object’s id> is an instance object that will be called automatically to perform the conversion function

The ‘Superclass’ and ‘Subclass’ attributes contain the superclass and the subclass information of the class respectively. The ‘Default-call’ attribute contains the default class’s instance that will be automatically called if the unit type conversion function is requested by the UV mediator.

The following is the ‘Currency’ class which stores currency unit type conversion objects.

CLASS Currency **HAS**
SUPERCLASS (Conversion)
SUBCLASS ( )
DEFAULT-CALL ([c001])
DESCRIPTION (‘Conversion knowledge that is responsible for converting the currency unit type’)
ATTRIBUTE
Program.Value string
Program.prop string
Designer.value string
Designer.prop string
Version.value string
Version.prop string

In a query execution, when the Unit Value mediator detects any conflicts regarding currency units or requires a currency unit conversion, the ‘Currency’ class is called by the Unit Value mediator for the currency exchange service. The ‘Currency’ class will assign the conversion knowledge accordingly. Unless a specific currency Conversion Knowledge object is requested, the ‘Currency’ class will assign its instance object ‘[C001]’ (which is based on the value in the ‘Default-call’ attribute) to perform the duties.

5.1.1.2.2 Reference Knowledge

The Reference Knowledge contains conversion rate information that is needed by conversion functions to perform unit type conversion services. The Reference Knowledge is organized as a class hierarchy (Figure 5-5) with a ‘Reference’ class as a root and

different subclasses for each supported unit conversion type; ex. ‘Exchange-rate’ class, ‘Time-range-rate’, etc. Each subclass contains a collection of Reference Knowledge objects, each of which contains conversion rate information that will be used by a conversion function to support the unit type conversion. For example, the ‘Exchange-rate’ class has instance objects ‘[usd-thb]’, ‘[usd-jpy]’, …, ‘[thb-usd]’. Each of this instance object contains a currency conversion rate that can be used by a currency conversion function if this object is called for services. This conversion rate information is kept separately as an object making it easy for update and maintenance. The exchange rate information is an example of the Reference Knowledge that needs a daily update. The update information can be obtained automatically from on-line information sources on the Internet.

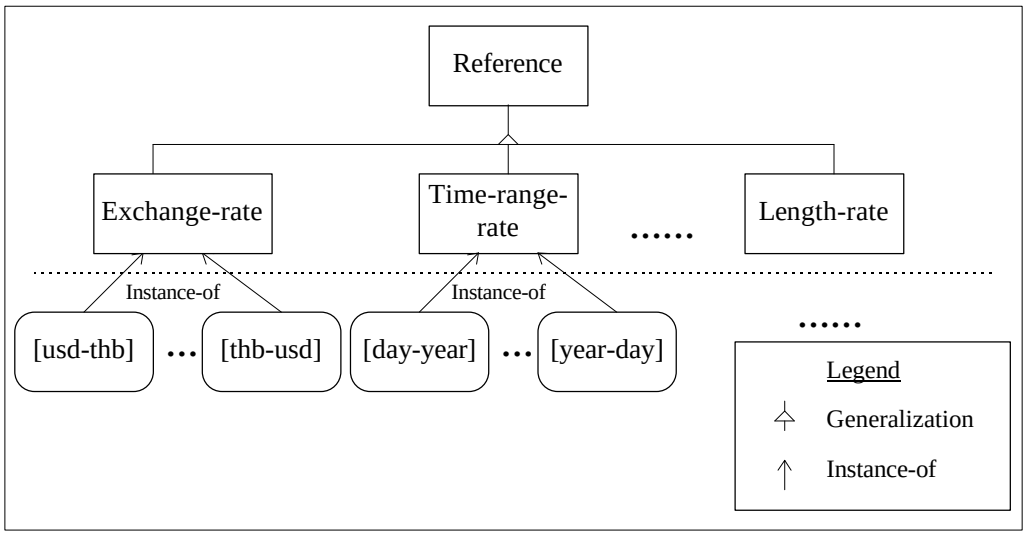


Figure 5-5 The Organization of the Reference Knowledge

The Reference Knowledge classes are codified using the specification for the ‘Reference’ class. The knowledge specification for the ‘Reference’ class contains the following information.

Reference Class ::=
CLASS <class-name> **HAS**
  SUPERCLASS (<superclass-name>)
  SUBCLASS (<subclass-name>+)
  UNIT-TYPE (<unit type>)
  SUPPORTED-UNIT (<unit value>+)
  DESCRIPTION (<description>)
  ATTRIBUTE
    From.value string
    From.prop string
    To.value string
    To.prop string
    Rate.value number
    Rate.prop string
    Version.value string
    Version.prop string
    Active.value string
    Active.prop string

The ‘Superclass’ and ‘Subclass’ attributes contain the superclass and the subclass information of the class respectively. The ‘Supported-unit’ attribute contains a list of supported unit values for supporting unit conversions within the unit type that is defined in the ‘Unit-type’ attribute.

The following is the knowledge specification for the ‘Exchange-Rate’ class which is a reference knowledge .

CLASS Exchange-Rate **HAS**
  SUPERCLASS (Reference)

SUBCLASS ( )
UNIT-TYPE (Currency)
SUPPORTED-UNIT (usd frf jpy thb)
DESCRIPTION (‘Exchange rate knowledge’)
ATTRIBUTE

From.Value	string
From.prop	string
To.Value	string
To.prop	string
Rate.Value	number
Rate.prop	string
Date.Value	string
Date.prop	string
Active.Value	string
Active.prop	string

From the above example, the ‘Exchange-Rate’ class has the exchange rate information for converting the currency unit among ‘usd’ (US Dollar), ‘frf’ (French Franc), ‘jpy’ (Japanese Yen), and ‘thb’ (Thai Baht) unit values.

OID	From. value	From.prop	To. Value	To.prop	Rate. Value	Rate.prop	Date. Value	Date.prop	Active .value	Active.prop
[usd-frf1]	USD	: Source URL http://ny.edu/rate : Unit-type None :	FRF	: Source URL http://una.hh.lib.. : Unit-type None :	5.42	: Source URL http://una.hh.lib.. : Unit-type None :	2/7/97	: Source URL http://una.hh.lib.. : Unit-type None :	N	: Source URL http://una.hh.lib.. : Unit-type None :
[usd-frf2]	USD	: Source URL http://una.hh.lib.. : Unit-type None :	FRF	: Source URL http://una.hh.lib.. : Unit-type None :	5.64	: Source URL http://una.hh.lib.. : Unit-type None :	2/10/97	: Source URL http://una.hh.lib.. : Unit-type None :	Y	: Source URL http://una.hh.lib.. : Unit-type None :
[usd-jpy]	USD	: Source URL http://una.hh.lib.. : Unit-type None :	JPY	: Source URL http://una.hh.lib.. : Unit-type None :	124.5	: Source URL http://una.hh.lib.. : Unit-type None :	2/10/97	: Source URL http://una.hh.lib.. : Unit-type None :	Y	: Source URL http://una.hh.lib.. : Unit-type None :
…. ….										
[jpy-usd]	JPY	: Source URL http://una.hh.lib.. : Unit-type None :	USD	: Source URL http://una.hh.lib.. : Unit-type None :	0.008	: Source URL http://una.hh.lib.. : Unit-type None :	2/10/97	: Source URL http://una.hh.lib.. : Unit-type None :	Y	: Source URL http://una.hh.lib.. : Unit-type None :

Figure 5-6 Example of Instance for the Class 'Exchange-Rate'

Figure 5-6 is the example of 'Exchange-rate' instance objects presented in a table format. There are two objects that contain currency exchange rate information for converting from 'usd' unit value to 'frf' unit value. The instance object '[usd-frf2]' has more recent exchange rate information than the instance object '[usd-frf1]' and is used by default for a currency conversion. The exchange rate information can be automatically obtained from the information sources on the Internet such as the Federal Reserve Bank of New York at URL address 'gopher://una.hh.lib.umich.edu/00/ebb/ monetary/noonfx.frb'.

5.1.2 Query Formulator and Query Processor

The Query Formulator accepts user requests for information and expresses those requests in terms of the knowledge contained in the Ontology (Application Domain). If errors are detected, the Query Formulator automatically attempts to make corrections. In the process, the Query Formulator uses the knowledge such as the synonym meta-data information to formulate a syntactically correct query. Once a query has been formulated, the Query Formulator submits the query to the Query Processor. The Query Processor then proceeds to plan and optimize the query steps and access strategies for subqueries to data sources. The Query Processor may consult Mediators for resolving any unit conflicts that may occur.

5.1.3 Mediators

A mediator, as described in Chapter 4, provides domain specialized services. The mediators are configured to assist the query processing and the data integration in the federated database system which integrates information from multiple data and information sources, having a diversification of data formats, different meanings, different time units, and providing differing levels of information quality [KGJM96]. The UV Mediator is an example of a mediator that provides unit value conversion services.

5.2 Multiple Unit Value Concept

The Multiple Unit Value (MUV) concept is a system capability that extends the schema definition to allow object attribute value to be stored, retrieved, and manipulated in more than one unit value. For example, consider the attribute salary that is defined to have a unit type as ‘Currency/Time-range’. Traditional data models must further define a specific unit value, e.g., ‘US Dollar/Month’, ‘Yen/Week’, etc., for the attribute salary. With MUV concept, the attribute salary does not need to be defined to any specific unit value. It will support all these different unit values. To integrate the information in the multidatabase environment, there is no necessity for data from different sources to be converted. It can be imported and stored in its own actual unit value in order to avoid data invalidity as the time changes.

The Multiple Unit Value (MUV) concept is the result of the cooperation among the Unit-type DMD, the UV mediator, and the extended DML. The Unit-type DMD allows data to be specified in different unit values. As a result, an attribute may contain

multiple conflicting unit values. The data manipulation language (DML) is extended so that it can cope with the differences of unit values and correctly process queries. The UV mediator is configured to be tightly integrated into the data model architecture. The UV mediator will not only provide a basic unit value conflict resolution function for the DML but also enhance the Mediation Data Model to support the MUV concept. In order to use this new system capability, the DML must be extended to support it. The extended query language specification is presented in the next section.

5.3 Extended Query Language Specification

In order to support the extensible schema definition of the Mediation Data Model, the data manipulation language has to be extended. The schema definition defines the class attribute as a pair attribute of the ‘value attribute’ and the ‘property attribute’, i.e., ‘Attribute.value’, and ‘Attribute.prop’. Accordingly, the attribute information tree splits into ‘value’ and ‘property’ branches (see Figure 5-7).

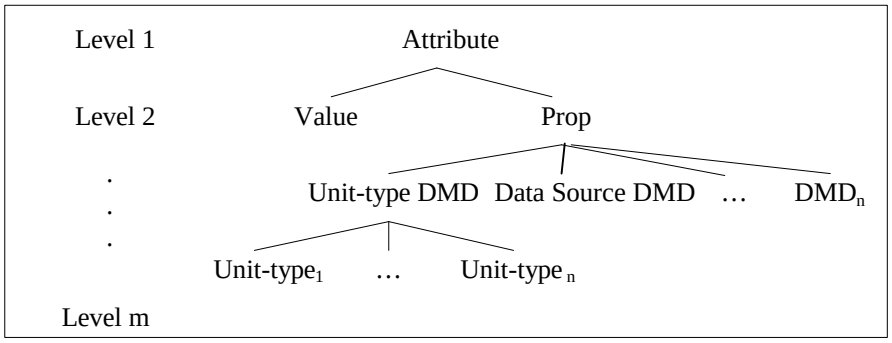


Figure 5-7 Attribute Information Tree

The DML must have the capabilities to retrieve and access both data values and data properties. The query language requires extensible instructions for traversing down the attribute information tree. The dot, '.', notation is used and the syntax is presented as follows:

Syntax:

$Level_1$ (attribute-name) . $Level_2$ (value | prop) . $Level_3$ . . . . $Level_m$

For example, users can retrieve the attribute salary's value by defining the attribute as 'Salary.value' and retrieve the attribute salary's Unit-type DMD by specifying the attribute as 'Salary.prop.unit-type'. The '.' tells the system to move down to a lower level of the attribute information tree having the attribute name at the top of the tree. In the case of 'Salary.prop.unit-type', the system goes down to the attribute's unit-type and returns all information of the unit type, including the information of all sub-level below the unit-type, back to users.

The Mediation Data Model provides a query language for query execution of the data which is stored in the data model. The query language specification is defined in the following format:

SELECT (<attribute>+)
UNIT-AS (<unit-value>+)
FROM (<instance-set>+)
WHERE (<condition>) | TRUE

Where
<attribute>::= attr-name | attr-name.value | attr-name.prop |
attr-name.prop.unit-type | attr-name.prop.source

<unit-value>::= (= attr-name.prop.unit-type <Unit-type DMD>)
<instance-set>::= (?variable <class-name>)
<condition>::= <Boolean-expression>

The ‘Select’ clause defines an output which is composed of a list of attributes as defined in the dot information tree specification. The ‘Unit-as’ clause defines the unit value for the attributes in the ‘Select’ clause. The ‘From’ clause defines an instance set. The instance set is a collection of instances. Each instance set member is an instance of a set of classes that are defined in the application domain. The ‘Where’ clause defines a Boolean expression which is the condition of the query. The result of the query is the set of instances from the instance set that satisfies the Boolean expression. More details of the query language specification are presented in Appendix B.

The Mediation Data Model also uses synonym meta-data to enhance the query formulation and to deal with different terminology in the query.

The query below retrieves a salary value and a salary unit type of a ‘Worker’ named ‘Peter’.

SELECT (?ins:Name.value ?ins:Salary.value ?ins:Salary.prop.unit-type)
FROM ((?ins **Worker**))
WHERE (eq ?ins:Name.value ‘Peter’)

The query is firstly processed by Query Formulator, which detects that the term ‘Worker’ is not defined as any class in the application domain ‘Airline’. However, from the synonym meta-data, the term ‘Worker’ is synonymous with the term ‘Employee’

which is a class in the application domain ‘Airline’. The Query Formulator then replaces the term ‘Worker’ in the query with the term ‘Employee’ as in the query below before it sends the query to the Query Processor for evaluation.

```
SELECT  (?ins:Name.value ?ins:Salary.value ?ins:Salary.prop.unit-type)
FROM    ((?ins Employee))
WHERE   (eq  ?ins:Name.value  ‘Peter’)
```

From the instance table in Figure 5-3, the results return as follows:

Name.value	Salary.value	Salary.prop.unit-type
Peter	2500	Unit-type Currency usd 1 Time-range month -1

The DML also needs to be extended to support the Multiple Unit Value concept of the data model. In the MUV concept, the data model can represent data in different unit values on an attribute. The DML must have the capability to handle data in different unit values to maintain the correct state of the data and to return correct results for the operations that are performed on these data items. For example, let the first data item be (1000, ‘Unit-type Length meter 1’) and the second data item be (1, ‘Unit-type Length kilometer 1’). Although both data items have different values, they are actually the same when scaled.

The Mediation Data Model provides a query language specification that supports attributes of different unit type as shown in Figure 5-8. The ‘UNIT-AS’ clause is introduced in addition to the traditional ‘SELECT-FROM-WHERE’ SQL query format.

The ‘UNIT-AS’ clause allows users to define a unit value for attributes that are stated in the ‘Select’ clause. The query language allows users to include data unit value as parameters in the query, which normally is not allowed by other query languages.

The Query Processor requires services from the Unit Value (UV) mediator when unit value conflicts occur. There are two phases in the query evaluation process that require services from the UV mediator: The pre-condition phase and The post-condition phase.

The pre-condition phase involves the evaluation of the Boolean expression in the ‘WHERE’ clause against the instance set. The Boolean expression must return the correct result when dealing with data in different unit values. The Boolean expression is composed of well-formed predicate functions, math functions, etc. Since unit type information is associated with attributes that have numeric domains, those functions must be extended to handle data with unit values. The predicate functions such as greater than function (>), less than function (<), equal function (=), etc., and the math functions such as addition (+), subtraction (-), etc. are extended to deal with the unit value data. The Query Processor calls the UV Mediator when using these functions to compare and convert the data to the same unit value before any of these functions can be processed.

The post-condition phase considers the final returned instances that satisfy the condition in the ‘WHERE’ clause. If unit values need any conversion to satisfy the user’s request in the ‘UNIT-AS’ clause, Query Processor then calls the UV Mediator to convert the data accordingly.

The example query in Figure 5-8 finds employee who have a salary greater than 2600 in ‘Canadian Dollars per month’ and presents the salary results in ‘Canadian Dollar per month’ unit value.

```
SELECT  (?ins:Name.value ?ins:Salary)
UNIT-AS ((= ?ins:Salary.prop.unit-type ‘Unit-type Currency cnd 1
        Time-range month -1’))
FROM    ((?ins Employee))
WHERE   (> ?ins:Salary.value ?ins:Salary.prop.unit-type
            2600 ‘Unit-type Currency cnd 1 Time-range month -1’)
```

Figure 5-8 Example of a query that involves Unit-type DMD

The salary value of each instance (?ins:Salary.value ?ins:Salary.prop.unit-type) and the condition salary value (2600 ‘Unit-type Currency cnd 1 Time month -1’) must be converted to the same unit value before the predicate function, ‘>’, can be processed. Then, all satisfied results are returned and converted to the requested unit type (‘Unit-type Currency cnd 1 Time-range month -1’). The Unit Value mediator is responsible for all unit conversions that are required in the process. From the same instance table in Figure 5-3, the query returns the following results to the user.

Name.value	Salary.value	Salary.prop
Peter	3372.5	: Source URL http://gmu.edu/main-db.cgi : Unit-type Currency cnd 1 Time-range month -1 by [c001] [usd-cnd] :

Martin	3166.227	: Source URL http://abc.com/db.cgi : Unit-type Currency cnd 1 Time-range month -1 by [c001] [thb-cnd] [t001] [day-month] :

From the results shown in the above table, both salary values have been converted from their original unit values. The Unit-type DMD of the attribute 'Salary' also provides the conversion explanation information which tells how each data has been converted. Let's look at the first value '3372.5'. The objects '[c001]' and '[usd-cnd]', which are a unit-type conversion knowledge object and a reference knowledge object respectively, are used to derive this converted data. Therefore, the salary value of '3372.5' is dependent on these objects ('[c001]', '[usd-cnd]'). If these objects are updated or become invalid, the salary value '3372.5' will become invalid. The '[usd-cnd]' is an instance of the 'Exchange-rate' class which is a kind of dynamic reference knowledge that has its value updated daily. Therefore the converted data '3372.5' may become invalid in the next day. However, the conversion explanation information allows the converted data to be converted back to its original unit value using the conversion values in effected at the time of conversion. At the same time, users can track the history of how the data is derived and thus feel more confident about the converted data. The conversion explanation information becomes more important when two converted data are compared. The system checks for any conflict in the explanation section of each object before comparing them. For example, the converted data may use the currency exchange rate from different reference knowledge objects (e.g., different dated versions). This results in incompatibility between two converted data; hence the data can not be compared.

6. The InfoFED Federated Database System

The InfoFED Federated Database System is a virtual database system that integrates data from autonomous and heterogeneous data sources. InfoFED operates within the Internet environment. The 'Internet', described by the Federal Networking Council (FNC), refers to the global information system that is logically linked together by globally address space based on the Internet Protocol (IP) and is able to support communications using the Transmission Control Protocol/Internet Protocol (TCP/IP). The Internet allows interoperability among different platforms by providing a common communication protocol connection. The Internet allows the designer to focus more on the data representation conflict problems rather than hardware and operating system conflict problems.

A federated database system provides a basic architecture for integrating information from heterogeneous databases. InfoFED adopts the federated database system as its referenced architecture. However, there is a major difference between this and the federated database system architecture. The global view in the approach is not the result of the integration from export schemas from local databases. The global view uses a well-developed ontology as the shared specification of a conceptualization for all export schemas. Since an ontology is intended to be created as a standard definition, the global

view that is created with an ontology will be more stable than one that is created as a result of the schema integration from export schemas.

As mentioned earlier, the federated database system requires data from local schemas to be converted and integrated into the common data model representation format. InfoFED uses the Mediation Data Model as the common data model. The Mediation Data Model supports the Multiple Unit Value Concept. The MUV concept reduces meta-data conflicts by allowing heterogeneous databases to export their data in their own unit values without any unit conversion. The Mediation Data Model also provides information, e.g., the unit type and the data source, for describing a data element. This information is very useful in the interoperable environment. For example, the data source information, when presented together with the data element, helps users to distinguish among different sources of the same data.

6.1 The InfoFED Federated Database System Architecture

The following discusses the four components of the InfoFED federated database system as presented in Figure 6-1: 1) Federation Manager, 2) Data Sources, 3) Information Gatherer, and 4) Users.

1.) Federation Manager. The Federation Manager manages the data that is gathered from data sources on the Internet and provides users with access to data. The Federation Manager is composed of four components: the Controller; the Mediation Database System; the Data Source Communication Agent; and the Web Server.

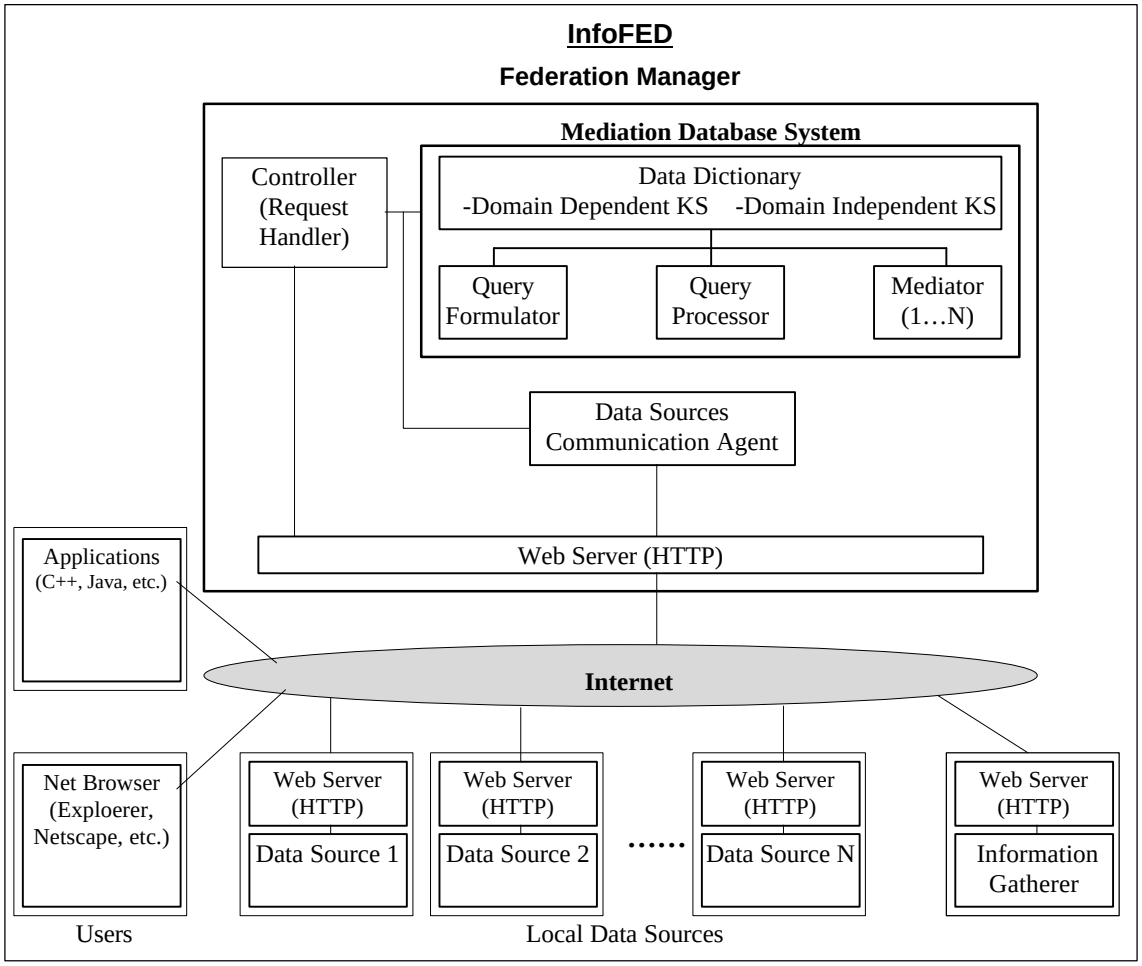


Figure 6-1 The InfoFED Federated Database System Architecture

The Controller provides an interface between the Federation Manager and users. It controls and monitors the activities in the Federation Manager. The Mediation Database System is a database management system, an extended data model that supports an extensible schema, and uses mediators to support data integration and query processing. The Data Source Communication Agent manages the connection to, and the data retrieval from, data sources. The Web Server, using Hypertext Transfer Protocol (HTTP), provides

the network connection among the Federation Manager, data sources, Information Gatherer, and users within the Internet environment. HTTP is the Web's Remote Procedure Call (RPC) on top of TCP/IP. HTTP allows the Federation Manager to effectively access resources that are defined in the URL format and connected within the Internet environment [Orfali&Harkey97].

2.) Data Sources. Data sources are the sources of data which information providers own and export to the Federation Manager (global level). Data sources are site autonomous. They retain control of information to be shared with the Federation Manager and decide on which global requests they will service. There is no data source modification required upon joining the Federation Manager. Data in the data sources is also independent from changes at the global level.

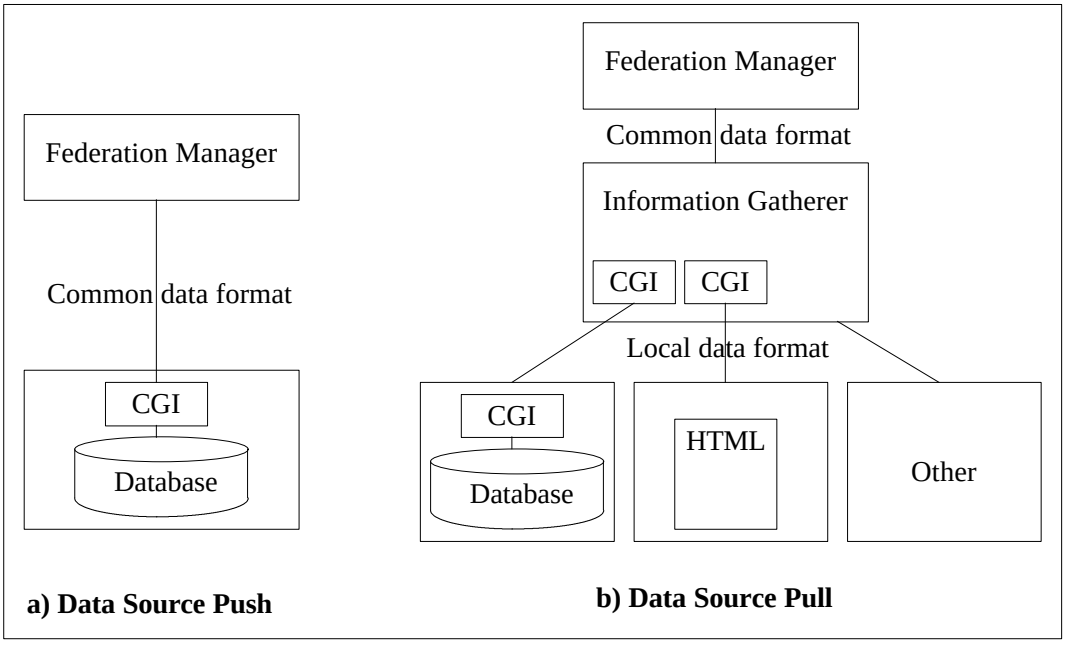


Figure 6-2 Data Obtaining Process

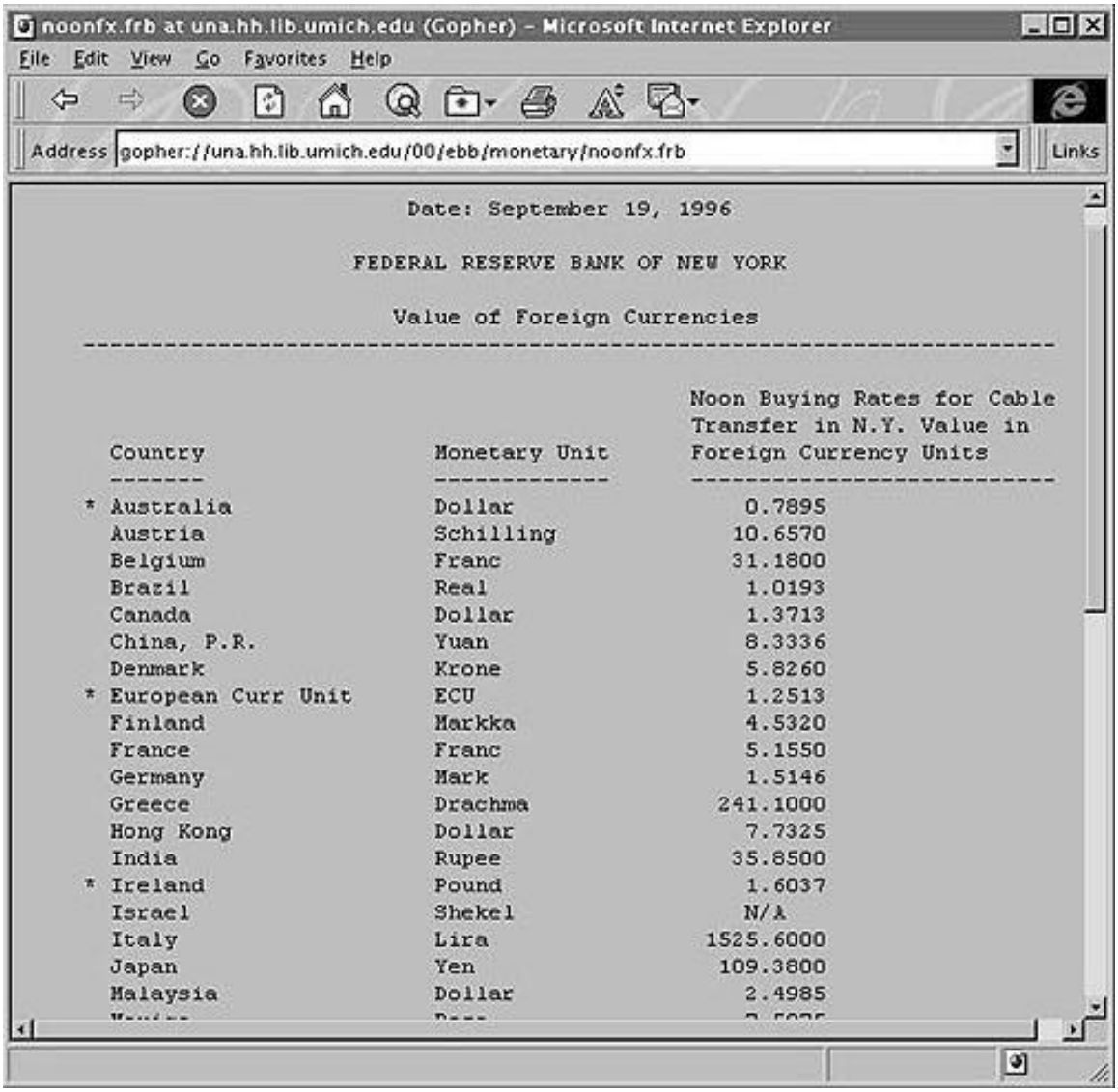
Common Gateway Interface programs (CGI programs) are required for the data export processes. The Internet address of CGI programs which can be defined within the URL specification allows the Federation Manager to locate and execute these CGI programs over the Internet. The CGI programs provide wrapping services which make data sources comply to an internal or external standard. The CGI programs modify queries from the Federation Manager into the data source format before passing it to the data sources. After that, the CGI programs modify data output from the data sources to the Federation Manager format before exporting it.

InfoFED obtains information in two ways as shown in Figure 6-2: Data Source Push and Data Source Pull. Data Source Push is the concept in which owners of data sources would like to export (push) their data to the Federation Manager. In this scenario, information providers provide the CGI programs that allow the Federation Manager to access their data (Data Source Push). The data sources retain their autonomy and export only the data portion they are willing to share. On the other hand, Data Source Pull is the concept in which the Federation Manager goes out, through the Information Gatherer, to get information from data sources on the Internet and imports (pull) information back to the Federation Manager. In this scenario, the Federation Manager uses the CGI programs that have been prepared by the Information Gatherer to access the data from data sources (Data Source Pull). In this case, the meta-data of the data sources must be known to the Information Gatherer.

Data sources can be represented in such formats as ORACLE, SYSBASE, CLIPS, HTML, etc. Figure 6-3 shows an example of data source in HTML format. The example

shows the exchange rate information source that is provided by the Federal Reserve Bank of New York. The data is presented in tabular form. The Information Gatherer provides the CGI program that performs the wrapping service by restructuring the data into a common data format which is understood by the Federation Manager. Although the exchange rate information is updated daily, the data presentation form, i.e., meta-data, does not change. This allows the Information Gatherer to routinely extract information and bring it back to the Federation Manager.

3.) Information Gatherer. Information Gatherer is an agent that is responsible for searching, retrieving, and extracting data (Data Source Pull) from data sources in the World Wide Web. This information can help a corporation to reduce the cost of obtaining and maintaining the information. For example, a corporation that needs daily currency exchange rate information may obtain it from a bank which provides the information through a fax or a delivery service. The corporation then has to pay for the bank services. However, the daily currency exchange rate information is provided by several sites in the World Wide Web. The Information Gatherer can go out to the World Wide Web and then search and retrieve data back to the Federation Manager. Internet search engines [Eichmann95] such as 'Harvest' [HSW95, and HSWBDM95], 'Excite', 'Webcrawler', 'InfoSeek', etc. can be used to assist the Information Gatherer to find the information in the desired topic. The currency exchange rate information in Figure 6-3 example is a result from these search engines.



Note: The exchange rate information from the Federal Reserve Bank of New York at the URL addresses “gopher://una.hh.lib.umich.edu/00/ebb/monetary/noonfx.frb”

Figure 6-3 Example of Data Source in the HTML Format

From the results, Information Gatherer uses a specified script program (a CGI program) to extract and restructure the data from the selected data source and import the information to the Federation Manager. The Information Gatherer is also responsible for

monitoring any changes in the data structure of the data sources and accordingly readjusting the data extracting strategy to match the changes. Also, Information Gatherer can be located at the same address as the Federation Manager or at any third party agents that are responsible for locating and collecting data for the Federation Manager.

4.) Users. Users are application programs or human beings who have an access to Internet by means of a Web Browser, such as 'Netscape Navigator' and 'Microsoft Internet Explorer', and request services from the Federation Manager.

6.2 Scenario for the InfoFED Federated Database System

This section describes each step annotated and the operations performed (or represented) and information sent (transmitted) in the InfoFED federated database system (Figure 6-4).

The process starts (step 1) when a user sends a request for information to the Federation Manager over the Internet. The request is received by the Controller (Request Handler). From the request, the Controller extracts the query to be processed and passes it to the Mediation Database System. The Mediation Database System determines from the query whether or not data is required from external sources. If the query requires no data from external data sources, the Mediation Database System executes the query and returns data to users (step 4). The Controller formats the output into an HTML document and sends it over the Internet to the user.

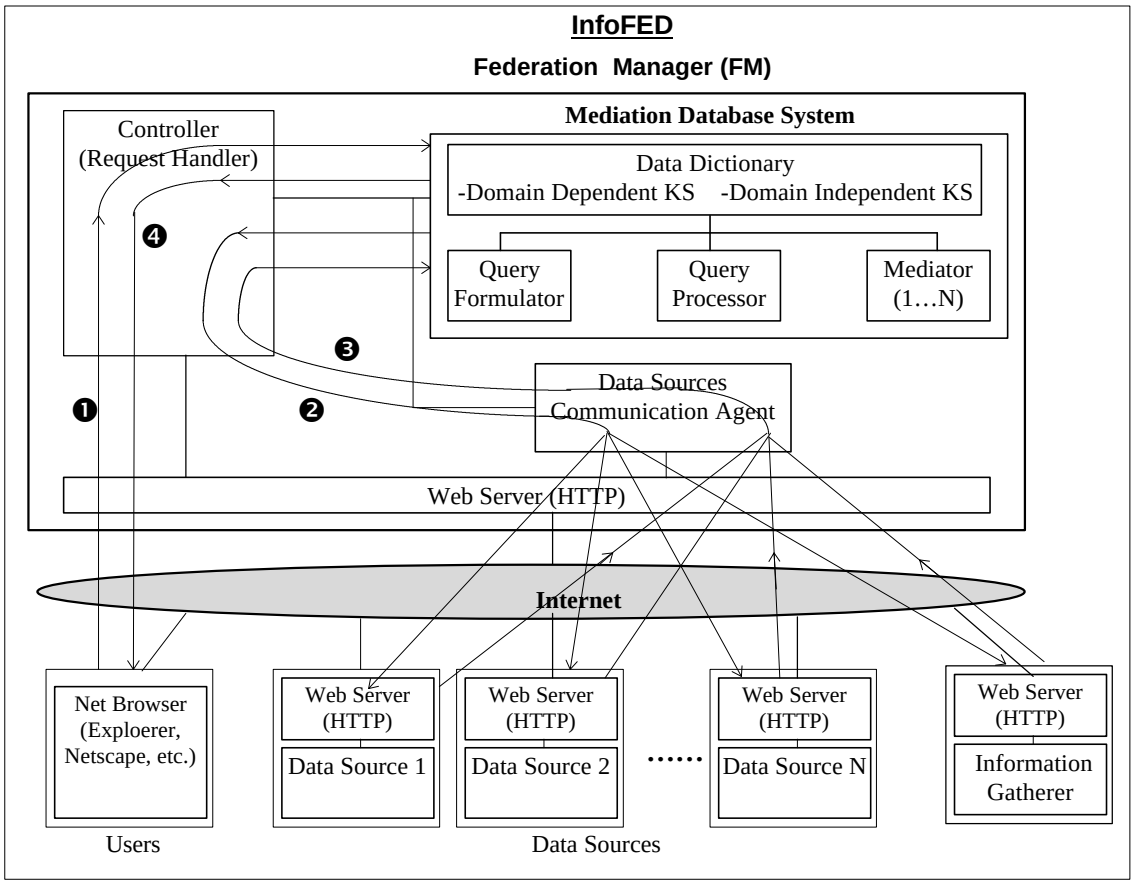


Figure 6-4 Information Flow Diagram in the InfoFED Federated Database System

However, if the query requires external data sources (step 2), the Mediation Database System sends a request along with a list of data source URLs and necessary information for contacting local data sources to the Controller. The Controller calls the Data Source Communication Agent to retrieve data from multiple data sources based on the information given by the Mediation Database System. Each data source returns the requested data to the Data Source Communication Agent which will return all retrieved

data, in a common data format, to the Mediation Database System through the Controller (step 3).

After the Mediation Database System receives all external data, it executes the users' query against all external data and returns the result to the Controller (step 4). Then, the Controller formats the output into HTML format and sends the final results to the user's browser over the Internet.

6.3 Information Integration

This section discusses how the data from heterogeneous data sources can be integrated and imported into InfoFED which uses the Mediation Data Model representation. The Mediation Data Model is based on an object-oriented data model which allows the entire federated database to be modeled as a single distributed complex object. In InfoFED, different databases (data sources) are used as one shared data source. As mentioned earlier, an ontology is specified at the Federation Manager (global level) and used as a global schema for the exported knowledge from local databases.

To submit the data to the Federation Manager, each local database maps its export schema to the global schema. Part of the integration responsibility is now shifted from the Federation Manager to the local level. As a result, the data integration is based on the willingness of the information providers to submit their data to the Federation Manager. InfoFED supports mapping the knowledge from different data representation knowledge, such as the relational data model, semantic data model, and the object oriented data model, to the Mediation Database System. The schema mapping knowledge of InfoFED

maps each object class view from the global view with the views of local data sources. This schema mapping contains information about who submitted the data for each object in the global schema, where to obtain the data, and how to get and map the data to the global schema. The schema mapping knowledge is stored as part of the domain dependent knowledge in the Data Dictionary of the Federation Manager.

The schema mapping knowledge contains two object classes: the 'Object-source' class and the 'Object-mapping' class. The 'Object-source' class contains objects each of which contains the list of the data sources that are willing to submit the data for an object class in the global schema. Each data source for an object class in the global schema is represented by an 'Object-mapping' object's identification (id). The 'Object-Mapping' class contains object indicating for each data source, the location and access method, and mapping of the data to the object class in the global schema.

Below is the knowledge specification for the 'Object-source' class

CLASS Object-source **HAS**
  ATTRIBUTE
    Term String
    Source-list String

and the knowledge specification for defining instance objects of the 'Object-source' class.

(<object id> **of Object-source**
    (Term <class-name>**)**
    (Source-list <Object-Mapping objects' ids>+ **)**
)

The following is the knowledge specification for the 'Object-mapping' class

CLASS Object-mapping **HAS**
  ATTRIBUTE
    Term string

Url-source string
Map-query string
Map-attribute string

and the knowledge specification for defining instance objects of the ‘Object-mapping’ class.

(<object id> **of Object-source**
(Term <class-name>**)**
(Url-source <url>**)**
(Map-query <mapped-query>**)**
(Map-attribute : <mapped-attribute :>+**)**
)

Where
<url> is an URL of a CGI program to connect to a local data source
<mapped-query> is a query, in a local database DML format, for accessing the local database
<mapped-attribute>::= <global-attribute> Attr <local-attribute> |
<global-attribute> Default <default-value> |
<global-attribute> NA
<local-attribute> ::= column-1 | column-2 | .. | column-N

The ‘Object-source’ class has two attributes: ‘Term’ and ‘Source-list’. The ‘Term’ attribute contains the name of the object class that is defined in the global schema, and the ‘Source-list’ attribute contains a list of Object-Mapping objects’ ids. Each of the Object-Mapping objects’ ids represents the source that submits data to the class that is stated in the ‘Term’ attribute.

The ‘Object-mapping’ class has four attributes: ‘Term’, ‘Url-source’, ‘Map-query’, and ‘Map-attribute’. The ‘Term’ attribute contains the name of the object class in the global schema. The ‘Url-source’ attribute contains an URL address of the data source CGI program.

The ‘Map-query’ attribute contains a mapped query that will be passed as a parameter to the local database to create an exported object view for exporting the data back to the Federation Manager. The ‘Map-attribute’ attribute contains information on how the global attribute names are associated with the exported object view. The ‘Attr’ parameter indicates that the global attribute name is mapped to an exported object view’s attribute name. If the exported object view does not provide a value for any global attribute name, either the ‘Default’ or ‘NA’ parameters is used. The ‘Default’ parameter indicates that the default value has been assigned to the global attribute name of all instances. The default value allows the Federation Manager to integrate data from traditional data models that do not support the property attribute. Although traditional data models do not support the property attribute, it is possible to assign one specific value to the property attribute. The ‘NA’ parameter indicates that there is no data to assign to that attribute name.

The mapped query resolves different kinds of the schema integration conflicts, i.e., the identity conflict, the semantic conflict and the structural conflict. The mapped query creates an exported object view from the local database schema. Data from each attribute of the exported object view can be mapped to the corresponding attribute of the object class in the global schema by using the attribute mapping information from the attribute ‘Map-attribute’.

```
Object-source
(Term01 of Object-source
        (Term Pilot)
        (Source-list Pi001 Pi002)
)

Object-mapping
(Pi001 of Object-mapping
        (Term Pilot)
        (Url-source  "http://site.gmu.edu/wiput/oracle.cgi")
        (Map-attribute
        : Ssn.value Attr Column-1
        : Ssn.property Default ": Source URL http://site.gmu.edu/wiput/oracle.cgi : Unit-
        type None :"
        : Last-name.value Attr Column-2
        : Last-name.property Default ": Source URL http://site.gmu.edu/wiput/oracle.cgi :
        Unit-type None :"
        : First-name.value Attr Column-3
        : First-name.property Default ": Source URL http://site.gmu.edu/wiput/oracle.cgi :
        Unit-type None :"
        : Salary.value Attr Column-4
        : Salary.property Default ": Source URL http://site.gmu.edu/wiput/oracle.cgi :
        Unit-type Currency usd 1 Time-range month -1 :"
        : Fly.value Attr Column-5
        : Fly.property Default ": Source URL http://site.gmu.edu/wiput/oracle.cgi : Unit-
        type None :" : )
        (Map-query     "SELECT   e.Ssn, e.Last-name, e.First-name, e.Salary, p.Fly
                        FROM     Employee e, Pilot p
                        WHERE    e.Ssn = p.Ssn;" )
)
```

Figure 6-5 Example of schema mapping knowledge specification

The global view is defined using the Mediation Data Model representation. The Mediation Data Model supports the Multiple Unit Value concept. The Multiple Unit Value concept reduces the complexity of the mapped query if the unit value conflict is

involved. The data sources can export their data in their own unit value without any conversions.

The example of instance objects for an ‘Object-source’ class and an ‘Object-mapping’ class which describe the schema mapping knowledge for the object class ‘Pilot’ is shown in Figure 6-5.

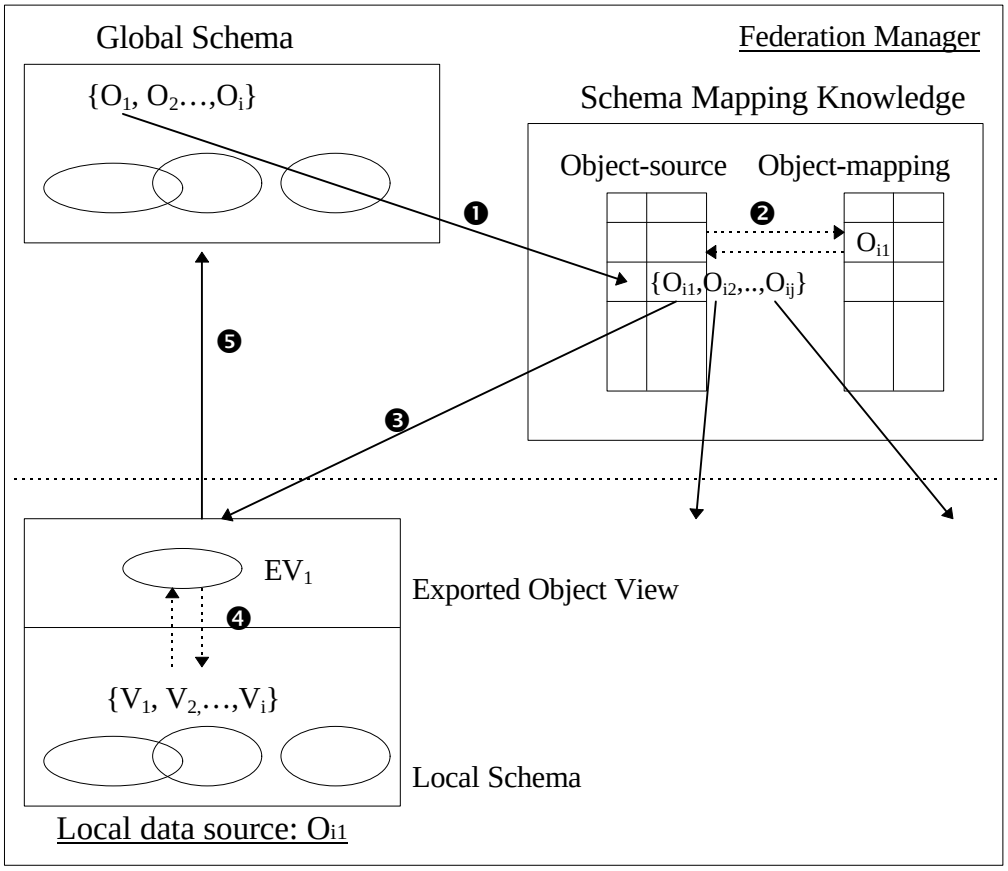


Figure 6-6 Local Data Source Accessing Process

Figure 6-6 shows how the Federation Manager retrieves data from local data sources. The global schema consists of a set of object classes, $\{O_1,O_2\ldots, O_i\}$. When users

request information from the system, the Federation Manager checks which of the object classes requires data retrievals. For each object class O_i that requires a data retrieval, the Federation Manager queries the schema mapping knowledge (Step 1) for the list of data sources, $\{O_{i1}, O_{i2}, \ldots, O_{ij}\}$, that provide data for object class O_i.

In Step 2, the schema mapping knowledge prepares, for each data source O_{ij}, the information on which sources to contact (the URL address of the CGI program), how to retrieve the data (the map query) and how to map the data (the map attribute information) to the object class O_i. Step 3, the Federation Manager connects to each data source and executes the local CGI program to access the local data source.

Step 4, the mapped query, which is sent as a CGI parameter, is passed as a query to the local database so as to create an exported object view, EV_1. This exported object view is recomputed whenever the local database is queried. Step 5, the CGI program converts the data into the common import data format (a tabular format) such as in Figure 6-7 and returns it to the Federation Manager for the integration. As the data is returned, the attribute mapping information is used for mapping the imported data to the common data model representation at the Federation Manager. The system also automatically generates implicit object identifiers with which to reference all imported instances objects. An object identifier is a computer artifact and has no meaning other than to identify and distinguish among imported instance objects.

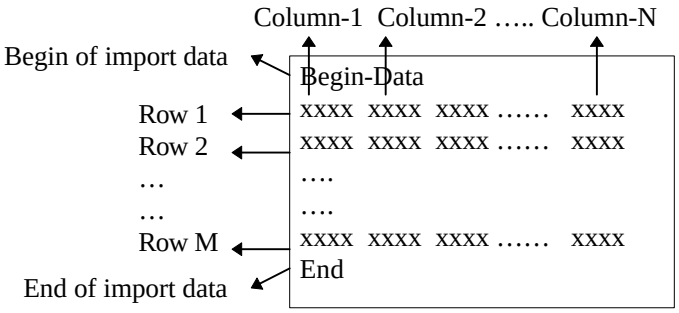


Figure 6-7 Import Data Format in a Tabular Format

For example, the object class ‘Pilot’ needs to retrieve the data from external data sources. From the schema mapping knowledge in Figure 6-5, there are two data sources that export data to the object class ‘Pilot’: ‘Pi001’ and ‘Pi002’ (Only the ‘Pi001’ is explained here and a similar scenario applies to the ‘Pi002’).

The Federation Manager connects to the data source ‘Pi001’ through the URL address ‘http://site.gmu.edu.edu/wiput/oracle.cgi’. The local CGI program, ‘oracle.cgi’, is executed and passes the mapped query, from Pi001’s attribute ‘Map-query’, to the local database. As the data is returned from the data sources, the attribute mapping information, from Pi001’s attribute ‘Map-attribute’, is used to map the imported data to the common data model representation (Mediation Data Model Representation) at the Federation Manager. Although the data model at the data source ‘Pi001’ does not support the property attribute, it is possible to assign a default value to it. The property attribute contains the Property Knowledge Package which is composed of Data Source DMD and Unit type DMD. The value of the Data Source DMD of all property attributes can be assigned as ‘http://site.gmu.edu.edu/wiput/oracle.cgi’. For the Unit type DMD, only the

attribute ‘Salary’ has the unit type which has a value as ‘Unit-type Currency usd 1 Time-range month -1’.

The Federation Manager also provides facilities for potential data sources to register their sites by providing the appropriate URLs, the query that may be executed locally, and the mapping of local data to global objects. The Federation Manager can then task the Information Gatherer to obtain the data for analysis as to relevance, quality, and overall site reliability before that site is formally incorporated into the federation.

7. Prototype

This chapter describes the prototypes that are implemented in this dissertation: the Unit Value Mediator and the InfoFED federated database system. The Unit Value Mediator is a stand alone agent that provides unit conversion services over the Internet. InfoFED is a federated database system that uses the Mediation Data Model as a common data model. The Mediation Database System provides a framework to incorporate mediators, such as, the Unit Value Mediator, which enhances the query processing and the data integration process within InfoFED.

7.1 The Unit Value Mediator Prototype

The Unit Value (UV) mediator is a stand alone agent that provides unit conversion services over the Internet. Users use the UV mediator services by providing the data, its Unit-type DMD, and the requested Unit type DMD. Then the UV mediator converts data to the requested unit value.

The prototype is implemented in CLIPS (C Language Integrated Production System) [Giarratano&Riley93, Giarratano93, NASA93a, NASA93b, and NASA93c]. The UV mediator requires conversion knowledge and reference knowledge to support the unit conversion. Both conversion knowledge and the reference knowledge are implemented in

the CLIPS Object-Oriented Language (COOL). COOL is an extension part of CLIPS to support an object-oriented knowledge representation model (Appendix A.1).

The UV mediator provides the Graphic User Interface (GUI) and the programming interface to users. Users can access the UV mediator's GUI, Figure 7-1, through the Internet. From the example, the user inputs values '3500', 'Unit-type Currency usd 1 Time-range month -1', and 'Unit-type Currency thb 1 Time-range week -1' which represent the data, the data Unit-type DMD, and the requested Unit-type-DMD respectively. Then, the user clicks the 'submit' button to request conversion services. The example shows the conversion of the data '3500' from the unit value 'US_Dollar/month' to 'Thai_Baht/week'. The converted data and its Unit-type DMD is displayed in the output section at the bottom of the screen. The 'HELP' page is also provided to assist users. It provides the information such as the Unit-type DMD specification, the supported unit types, and the supported unit values for each unit type.

The UV mediator also provides the programming interface which allows users or application programs to connect and execute the UV mediator program over the Internet using the standard URL format (HTTP/CGI). This allows the UV mediator to be embedded in any application program that supports the built in network capability such as the JAVA programming language [Flanagan96, Lemay&Perkins96, and Symantec96]. With some minor modification, the UV mediator can act as a middleware between users and any data sources. It provides the unit conversion services by preparing the data from the data sources to the users according to their requests.

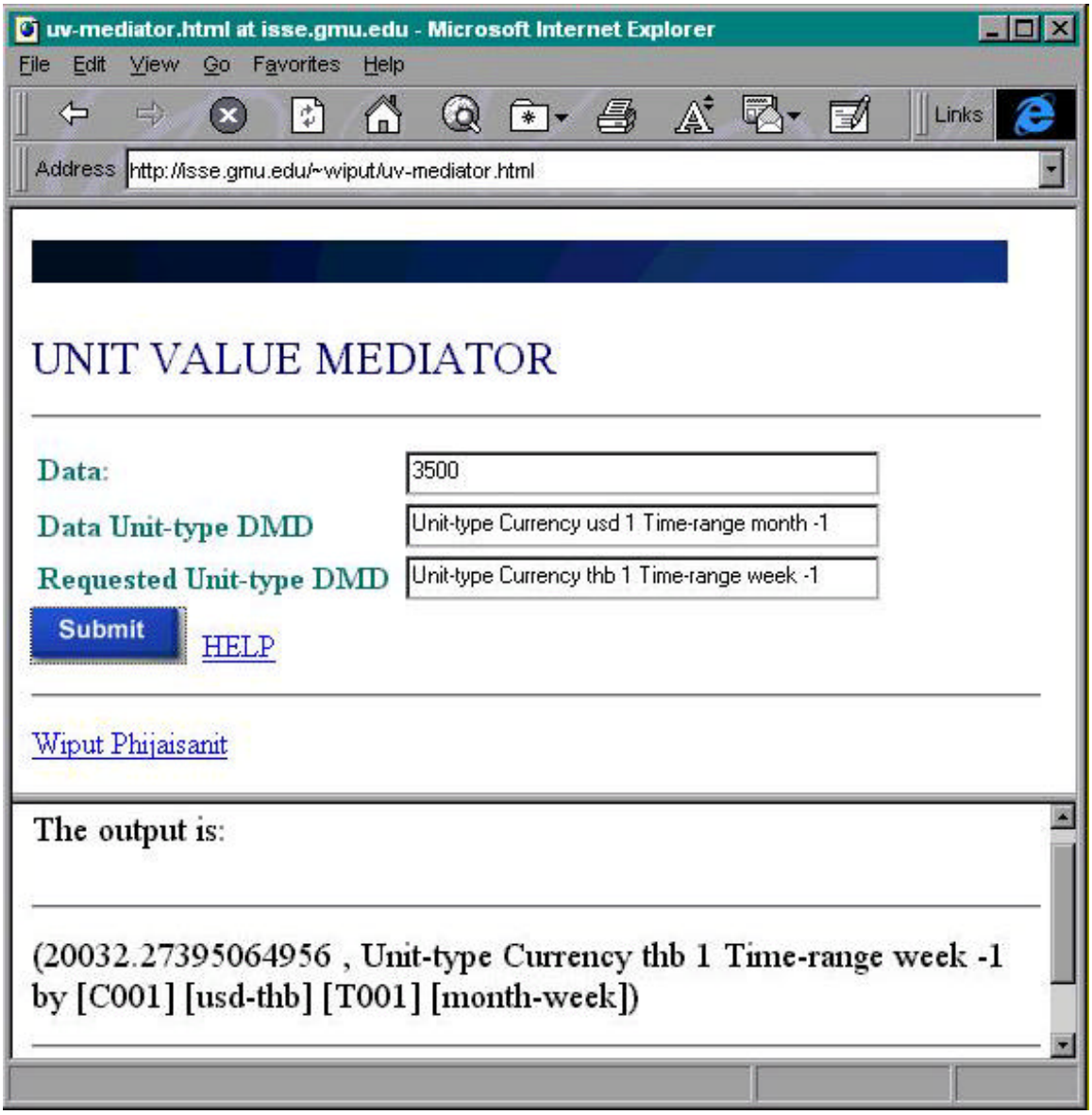


Figure 7-1 The Unit Value Mediator

7.2 The InfoFED Prototype

InfoFED is a federated database system that integrates data from multiple heterogeneous data sources. InfoFED uses the Mediation Data Model as a common data

model. The Mediation Data Model is developed in COOL. COOL supports classes, attributes, and instances. It supports the classification concept, the generalization/specialization concept [Smith&Smith77], and the aggregation concept. The Mediation Data Model extends COOL to support the property attribute for storing dynamic meta-data in addition to the traditional attribute values. InfoFED allows users to browse both data and meta-data, and query data stored in multiple, heterogeneous (ORACLE, CLIPS, SYSBASE, etc.) of database and file systems. In the initial prototype, InfoFED connects to two types of data representation formats, an Oracle database and CLIPS knowledge representation. In this prototype, the Data Source Communication Agent was developed using JAVA language for accessing multiple data sources over the Internet. JAVA language supports multithread programming which allows InfoFED to concurrently connect to multiple data sources. Users can retrieve and manipulate data in multiple unit values. The Unit Value Mediator, extended query language, and domain independent knowledge sources (conversion and reference knowledge source) have been developed. In this prototype, the Mediation Data Model serves as a common data model to integrate multiple databases.

The InfoFED prototype also demonstrates how information integration concepts from section 6.3 can be used to integrate data from heterogeneous database schemas to the global view, via the Mediation Data Model. When users connect to InfoFED, it provides them with choices of application domains that are supported by the system as shown in Figure 7-2. InfoFED then loads the application domain in accordance with the users' selection. From the example, the application domain 'Airline' is selected. InfoFED

then displays the application domain main menu (Figure 7-3). From this screen, InfoFED gives users three options for interacting with the application domain ‘Airline’; they are Browser mode, Query mode, and Register New Source mode.

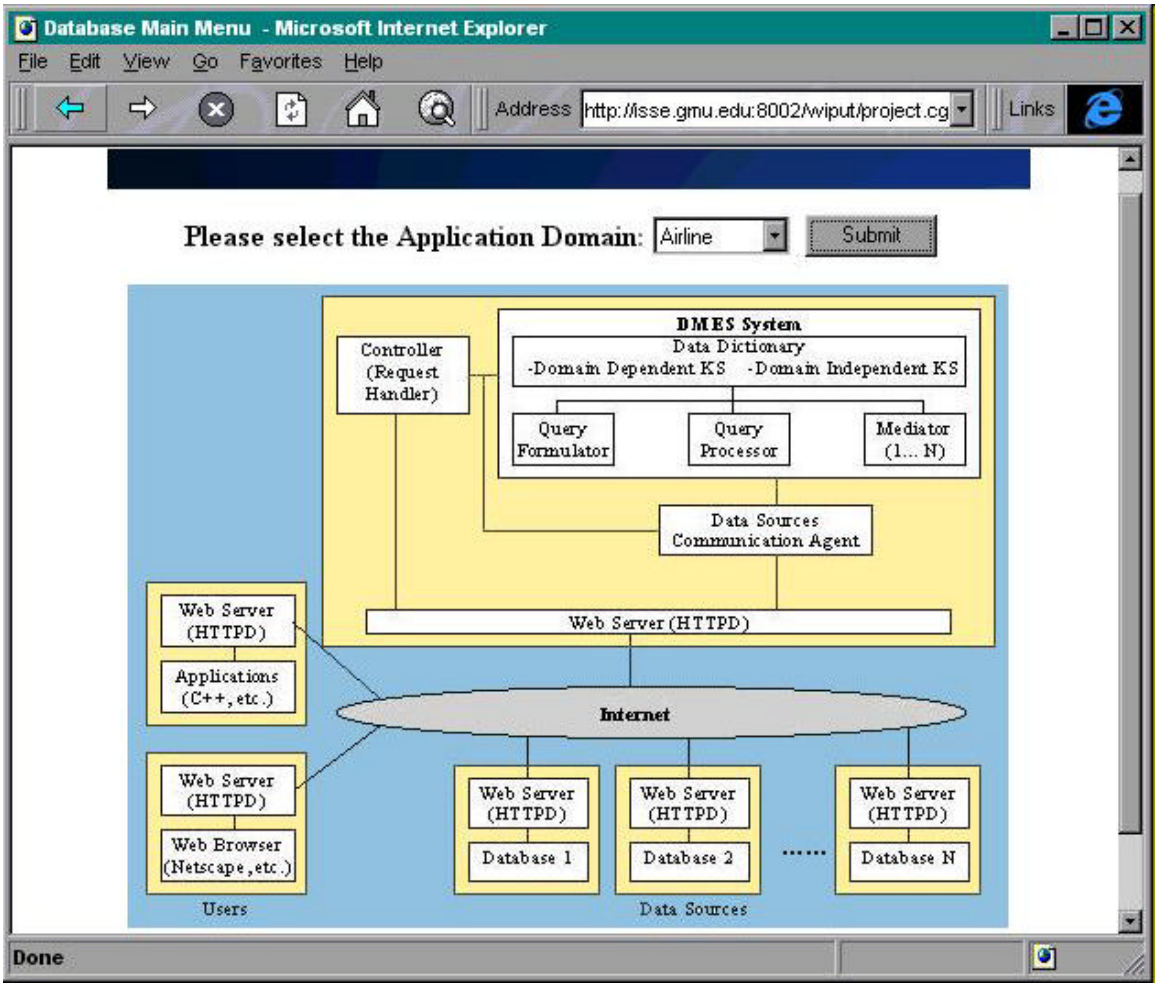


Figure 7-2 InfoFED Main Menu

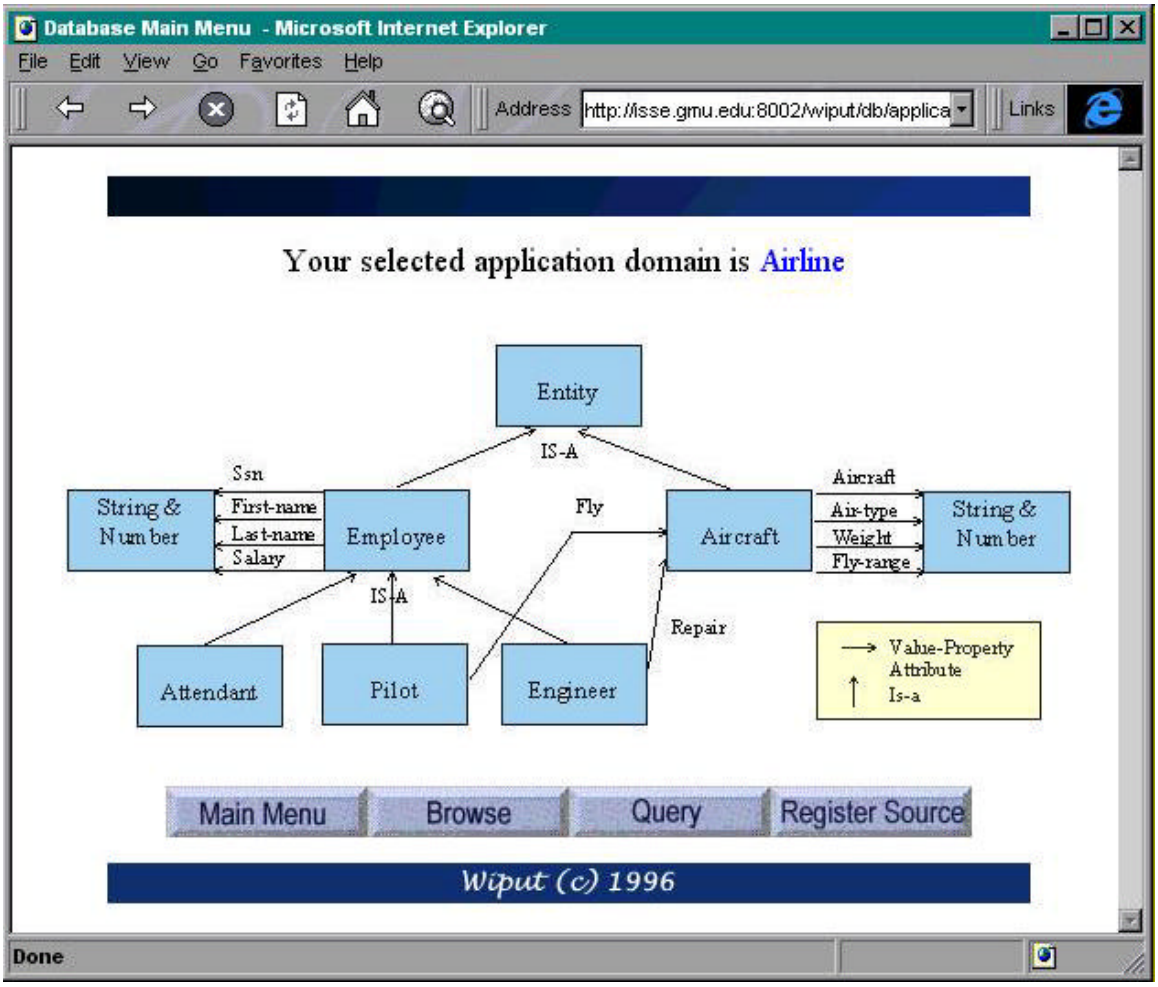


Figure 7-3 InfoFED Application Domain 'Airline' Main Menu

In the Browser mode (Figure 7-4), users can select a class and move from one class to another class in the information network and examine it. Users can browse through both domain dependent and domain independent knowledge sources.

From the example, the class 'Pilot' is selected. The Browser Frame provides different kinds of information about the class 'Pilot'. The frame shows the meta-data information such as superclasses, subclasses, and attributes which can be further selected to move to other classes in the active application domain. For example, users can move to

investigate the class ‘Employee’ by selecting the term ‘Employee’ in the superclass slot. The data source slot shows a list of data sources that provide data for InfoFED in the data source slot. Users can select data sources from the list and the Federation Manager will connect and retrieve data from these data sources accordingly so that users can further examine the data.

The information from the browser frame assists users in formulating a complex query in the query frame (Figure 7-5). This is very useful in the multidatabase environment where users may not know all the information about the large complex schema. In the query frame, users can query the Federation Manager through the extended COOL query language as explained in the section 5.3. Users can retrieve data in any of the unit values supported by the domain independent knowledge source.

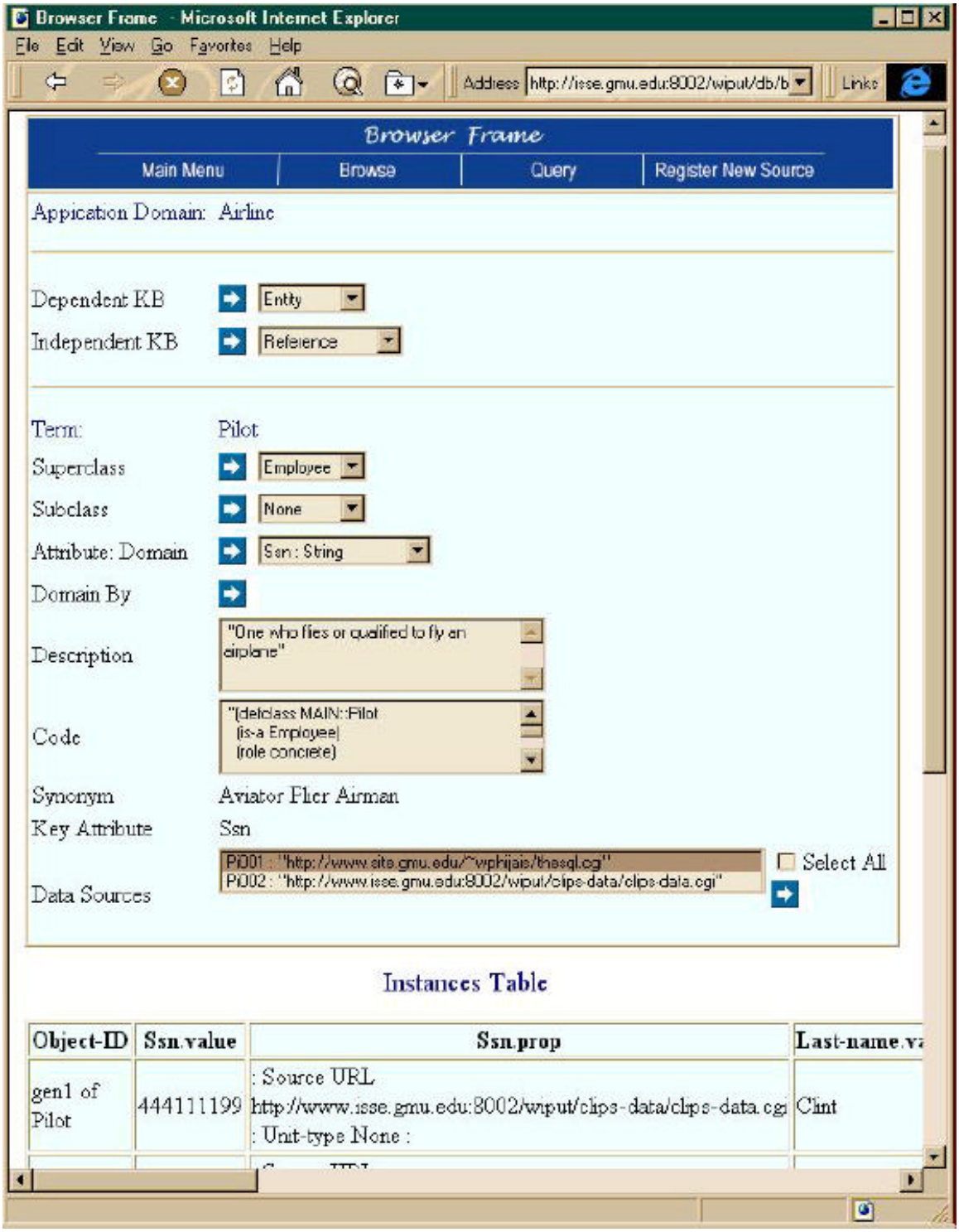


Figure 7-4 InfoFED Browser Mode

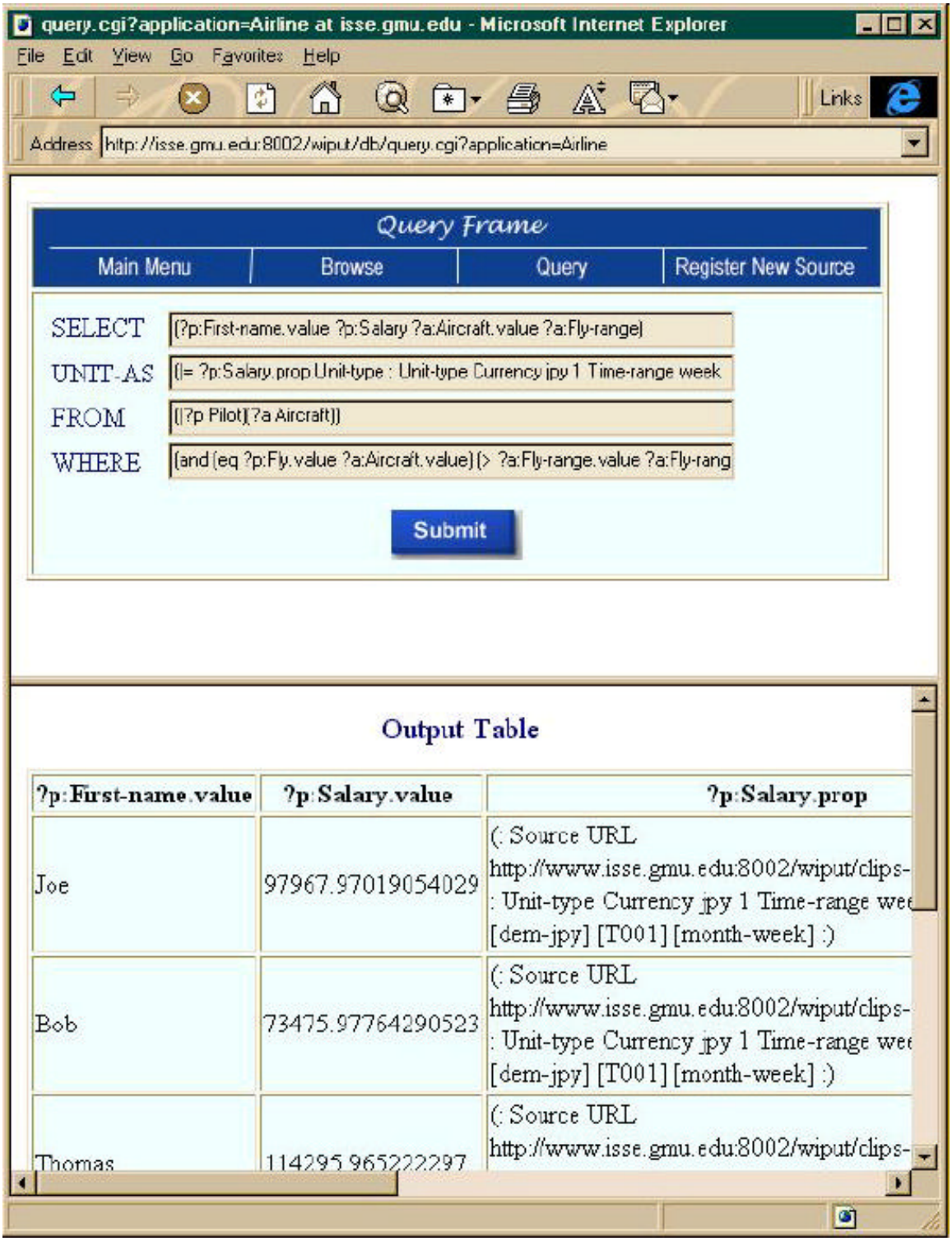


Figure 7-5 Query Frame for InfoFED

The Query Frame allows users to formulate complex queries for InfoFED. From the query in the Query Frame, the user requests the Federation Manager to find the pilot who flies the aircraft which has a fly-range greater than 3500 miles and reports the pilot's name, how many Yens he earns per week, the aircraft type and the fly-range of the aircraft. The query is defined as follows.

```
SELECT  (?p:Name.value ?p:Salary ?a:Aircraft.value ?a:Fly-range )
UNIT-AS ((= ?p:Salary.prop.unit-type 'Unit-type Currency jpy 1
        Time-range week -1'))
FROM    ((?p Pilot) (?a Aircraft))
WHERE   (and (eq ?p:Fly.value ?:Aircraft.value)
        (> ?a:Fly-range.value ?a:Fly-range.prop.unit-type
            3500 'Unit-type Distance mile 1'))
```

At the Federation Manager, the user's query is firstly received by Query Formulator. Query Formulator checks for errors and tries to correct them. Then the query is sent to Query Processor for evaluation. From the query, external data sources for the class 'Pilot' and the class 'Aircraft' are required. The Communication Agent is responsible for connecting to these data sources and retrieving data back to the Federation Manager. After all data is returned, the federal query is processed and the instances that satisfy all constraints are presented to the user.

InfoFED allows data providers to link their data sources to the Federation Manager (Figure 7-6) by themselves. Each data provider can add, edit, or remove their

data from the Federation Manager. The information in the browser frame assists the data providers to understand the global schema definition. Although the concept allows the data providers to do the integration by themselves, it is necessary to have a closed supervision on each integration of each data source. Automated or manual checking is required to ensure the correctness of the data integration before allowing the data to be accessed by other users.

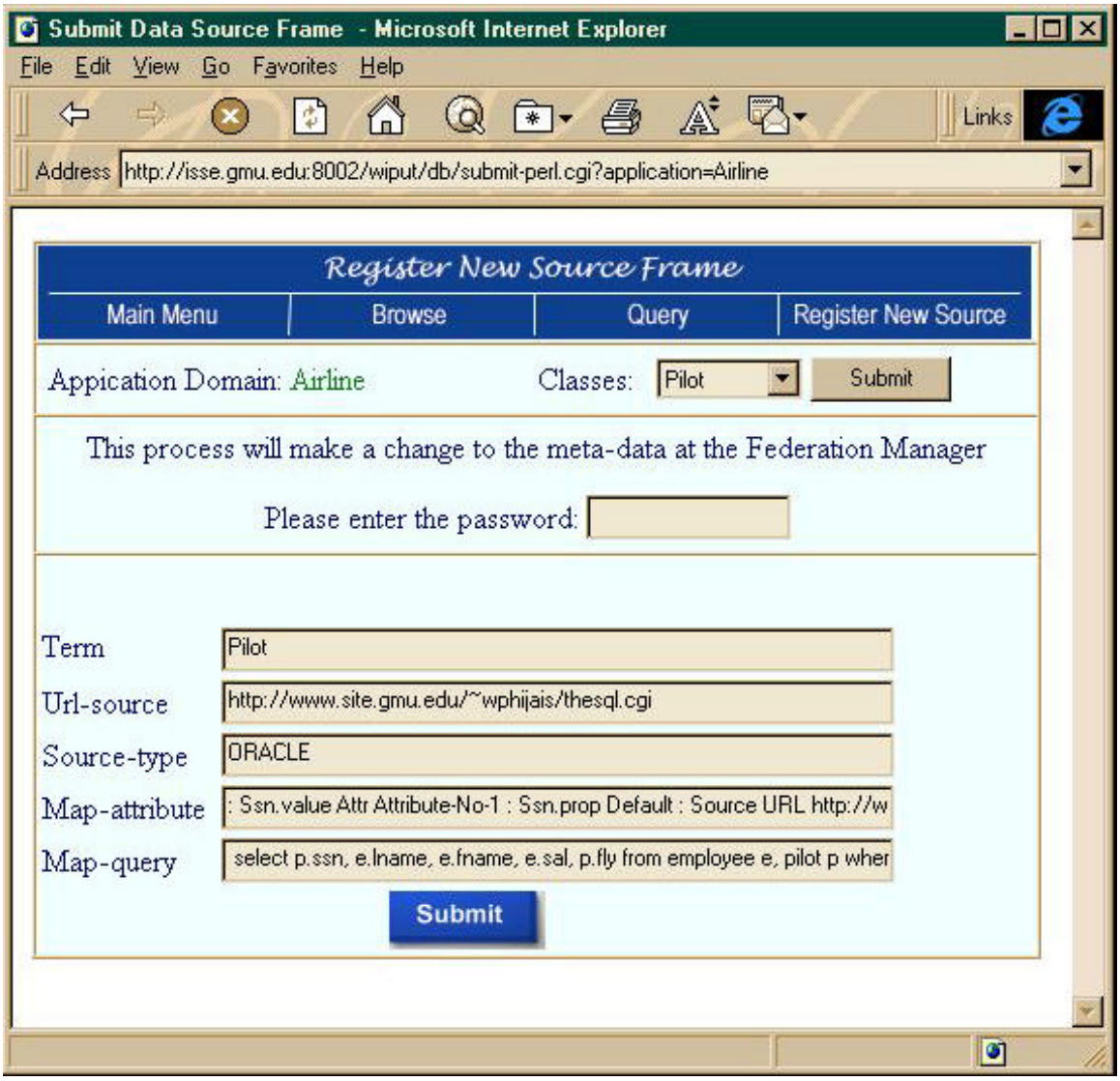


Figure 7-6 New Register Data Source Frame for InfoFED

8. Conclusion

8.1 Goals Achieved

In heterogeneous database systems where data are constantly exchanged, meta-data plays an increasingly important role than in a single database environment. The federated database system accesses and manipulates data from several data sources. The Mediation Data Model is introduced and developed. The Mediation Data Model provides an extensible schema that supports dynamic meta-data as well as static meta-data, and a framework for specifying mediator services within a data model. The dynamic meta-data describes different types of data semantic information, in addition to the traditional static meta-data. The dynamic meta-data allows each instance object of a class to have dynamic meta-data associated as properties of attribute, whereas the static meta-data assigns the same meta-data information common to all instance objects of a class. Dynamic Meta-Data refers to properties of an attribute such as the data quality of an attribute value, the reliability of the source provides the data, the units for the data value, and other dynamically changing properties of attribute.

The Mediation Data Model demonstrates that the Dynamic Meta-Data (DMD) can be used to enhance the semantics of a database schema, support data integration and mediation services, and facilitate query processing in a federated database system. With

DMD attached to the data element, the Unit-Type DMD enables the Multiple Unit Value (MUV) capability which allows object attributes to store, retrieve, and manipulate data in convertible unit values. The Data Source DMD supplies to users the origin from which the data is imported and helps users to distinguish among different sources of the same data in a multidatabase environment.

The Mediation Data Model provides a framework to incorporate mediators into the data model which, in turn, extends the data model capability. The Multiple Unit Value (MUV) concept is implemented by means of Unit-Type DMD, Unit Value Mediator, and extended Data Manipulation Language.

As the proof-of-concept, the Unit Value Mediator and the InfoFED federated database system have been developed. InfoFED makes use of the Mediation Data Model as a common data model to provide self-describing data as well as an intelligent data manipulation language to users, and to improve data integration in a federated database system. The Mediation Data Model provides self-describing data by supplying additional information such as unit types and data sources to help users to interpret data. The Mediation Data Model provides an intelligent data manipulation language by allowing users to include unit type (dynamic meta-data) as parameters in a query. The Mediation Data Model improves data integration by reducing the semantic conflicts among heterogeneous data sources. The schema integration conflicts are reduced because the Mediation Data Model supports the MUV concept allowing heterogeneous databases to export their data in their own unit values while the Unit Value Mediator provides translation and conversion services at the Federal level. InfoFED also provides a browser

for both data and meta-data, an object-oriented federated schema, and the data source registration and management.

8.2 Future Directions

Mediators will be more commonly used as systems provide for domain specialized services to support interoperability among heterogeneous data sources. The research in this dissertation provides a framework for defining dynamic meta-data and incorporating mediators into the data model, the Mediation Data Model. The framework extends the data model capabilities to support more complex schema representation and query processing, and enhances the data integration process in the multidatabase systems.

Additional types of dynamic meta-data and mediators should be investigated to support new features in the data model. For example, the Media-type DMD and the Media-type mediator should be investigated on how it can extend the data model to support multimedia objects. The Data-source DMD and the Data-source mediator should also be investigated on how it can extend the data model to support data filtering services. Users can select the data based on the corporate type of data sources (ex., government, education, etc.) or data quality of data sources in InfoFED.

Another area is the performance and the query optimization strategy. The Mediation Data Model requires services from mediators in the query processing. These mediators may incorporate functions that may perform complex calculations. The cost of these functions may vary considerably. The processing costs will directly impact the

performance of the query processing. Different query optimization strategies and quality-of-service models should be investigated for dealing with services involving mediators.

References

References

[ADSYTY93] M. Andersson, Y. Dupont, S. Spaccapietra, K. Y'etongnon, M. Tresch, and H. Ye, "The FEMUS Experience in Building a Federated Multilingual Database," In *Proc. 3rd Int'l Workshop on Research Issues on Data Engineering: Interoperability in Multidatabase Systems*, IEEE Computer Society Press, April 1993, pp. 65-68.

[AHKS95] Yigal Arens, Richard Hull, Roger King, and Michael Siegel, *Reference Architecture for the Intelligent Integration of Information*, Version 2.0 (Draft), August 22, 1995.

[APT96] "Briefing Paper: What is Metadata", APT Software Tools Bulletins, Available at: http://www.computerwire.com/bulletinsuk/ 212e_1a6.htm, March 1996.

[Bertino91] E. Bertino, "Integration of Heterogeneous Data Repositories by using Object-Oriented Views," *IEEE First International Workshop on Inteoperability in Multidatabase Systems*, Kyoto, Japan, 1991.

[BLN86] C. Batini, M. Lenzerini, and S. Navathe, "A Comparative Analysis of Methodologies for Database Schema Integration," *ACM Computing Surveys*, Vol. 18, Dec. 4, 1986.

[Brathwaite92] Kenmore S. Brathwaite, *Object-Oriented Database Design: Concepts and Application*, Academic Press, 1992.

[CPM84] S. Ceri, G. Pelagatti, and P. Milano, *Distributed Databases Principle and Systems*, McGraw-Hill, 1984.

[Dwyer&Larson87] P. A. Dwyer, and J. A. Larson, "Some Experiences with a Distributed Database Testbed System," *Proceedings of the IEEE*, Vol. 75, No. 5, May 1987.

[Edward90] J. M. Edward, "Object-Oriented," *Communication of ACM*, Vol. 33, No. 9, Sept. 1990.

[Eichmann95] David Eichmann, “Advances in Network Information Discovery and Retrieval,” University of Houston, 1995.

[Flanagan96] David Flanagan, *Java in a Nutshell*, O’Reilly & Associates, Inc., 1996.

[FFMM94a] Tim Finin, Rich Fritzson, Don Mckay and Robin McEntire, “KQML as an Agent Communication Language”, *Proceedings of the Third International Conference on Information and Knowledge Management*, 1994.

[FFMM94b] Tim Finin, Rich Fritzson, Don Mckay and Robin McEntire, “KQML - A Language and Protocal for Knowledge and Information Exchange”, Available at: “http://www.cs.umbc.edu/kqml/papers/kqml94.ps”.

[Gattorna96] Giacomo Gattorna, “Advanced Database Topics: Chapter 23 Mediation in DB Systems”, Available at: “http://www.cs.rpi.edu/~gattorng/ db1.html”, 1996.

[Giarratano93] Joseph C. Giarratano, *CLIPS User’s Guide*, NASA Lyndon B. Johnson Space Center, May 1993.

[Goodman94] Jerome N. Goodman, “Alberta Land Related Information System, A Federated Database Case Study”, Available from: “http://wwwsgi.ursus.maine.edu/gisweb/spatdb/urisa/ur94037.html”, 1994.

[Graham95] Ian S. Graham, *The HTML Source Book*, John Wiley & Sons, Inc, 1995.

[Gruber93] Thomas R. Gruber, “Toward Principles for the Design of Ontologies Used for Knowledge Sharing,” *International Workshop on Formal Ontology*, March 1993.

[Gruber&Olsen94] Thomas R. Gruber, and Gregory R. Olsen, “An Ontology for Engineering Mathematics,” *Fourth International Conference on Principles of Knowledge Representation and Reasoning*, 1994.

[GMS94] C. H. Goh, S. E. Madnick, and M. D. Siegal, “Context Interchange: Overcoming the Challenges of Large-Scale Interoperable Database Systems in a Dynamic Environment,” *Proceedings of the Third International Conference on Information and Knowledge Management*, 1994.

[Giarratano&Riley93] Joseph Giarratano, and Gary Riley, *Expert Systems: Principles and Programming*, PWS Publishing Company, 1993.

[HBP92] A.R. Hurson, M.W. Bright, and S.H. Pakzad, "A taxonomy and Currency Issues in Multidatabase Systems," *Computer*, Vol.25, No. 3, Mar. 1992, pp. 50-60.

[HBP94] A.R. Hurson, M.W. Bright, and S.H. Pakzad, *Multidatabase Systems: an Advanced Solution for Global Information sharing*, IEEE Computer Society Press, 1994.

[Hayne&Ram90] S. Hayne and S. Ram, "Multi-User View Integration System (MUVIS): An Expert System for View Integration", *Proceedings of the 6th International Conference on Data Engineering* (Feb.), 1990.

[Howard&Rehak89] H. Craig Howard, and Daniel R. Rehak, "KADBASE Interfacing Expert Systems with Database," *IEEE Expert*, Fall 1989.

[Heiler&Siegel91] Sandra Heiler, and Michael Siegel, "Heterogeneous Information Systems: Understanding Integration," *IEEE First International Workshop on Interoperability in Multidatabase Systems*, Kyoto, Japan, 1991.

[HSW95] Darren R. Hardy, Michael F. Schwartz, and Duane Wessels, *Harvest User's Manual*, University of Colorado at Boulder, Sept. 1995.

[HSWBDM95] Darren R. Hardy, Michael F. Schwartz, Duane Wessels, C. M. Bowman, Peter. B. Danzig, and Udi Manber, *Harvest: A Scaleable, Customizable Discovery and Access System*, University of Colorado at Boulder, March. 1995.

[Jarke&Koch84] Matthias Jarke, and Jurgen Koch, "Query Optimization in Database Systems", *ACM Computing Surveys*. Vol. 16, No. 2, June 1984.

[KGJM96] L. Kerschberg, H. Gomaa, S. Jajodia, and A. Motro, *Knowledge Rovers: A Family of Intelligent Software Agents for Logistics for the Warrior Volume 1 : Technical Proposal.* Department of Information and Software Systems Engineering, George Mason University, 1996.

[Litwin88] W. Litwin, “From Database Systems to Multidatabase systems: Why and How,” *Proc. Sixth British National Conference On Databases,* Cambridge Univ. Press, New York, N.Y., 1988, pp. 161-188.

[Litwin&Abdellatif86] W. Litwin, and A. Abdellatif, "Multidatabase Interoperability," *IEEE Computer*, 1986.

[LMR90] W. Litwin, L. Mark, and N. Roussopoulos, "Interoperability of Multiple Autonomous Databases," *ACM Computing Surveys,* Vol. 22, No. 3, Sept. 1990, pp. 267-293.

[Lemay&Perkins96] Laura Lemay, and , Charles L. Perkins, *Teach Yourself JAVA in 21 Days*, Sams.net, 1996.

[Motro87] Motro, A., "*Superviews: Virtual Integration of Multiple Database",* IEEE Transactions on Software Engineering, Vol SE-13, No 7 July, 1987.

[NASA93a] *CLIPS Reference Manual Volume I: Basic Programming Guide,* NASA Lyndon B. Johnson Space Center, June, 1993.

[NASA93b] *CLIPS Reference Manual Volume II: Advanced Programming Guide,* NASA Lyndon B. Johnson Space Center, June, 1993.

[NASA93c] *CLIPS Reference Manual Volume III: Interfaces Guide,* NASA Lyndon B. Johnson Space Center, June, 1993.

[NEL86] S. Navathe, R. Elmasri, and J. Larsin, "Integrating User Views in Database Design," *IEEE Comput.,* Vol. 19 Jan 1, 1986.

[Orfali&Harky97] Robert Orfali and Dan Harky, Client/Server Programming with JAVA and CORBA, John Wiley & Sons, Inc., 1997.

[PCKW89] K. Parsaya, M. Chignell, S. Khoshafian, and H. Wong, *Intelligent Databases.,* Wiley, 1989.

[PBE95] Evaggelia Pitoura, Omran Bukhres, and Ahmed Elmagarmid, “Object Orientation in Multidatabase Systems,” *ACM Computing Surveys,* Vol. 27, No. 2, June 1995, pp 141-195.

[RBPEL91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*. Prentice Hall, NJ, 1991.

[Shipman81] D. Shipman, "The Functional Data Model and the Data Language DAPLEX," *ACM Transaction on Database Systems* 6, No. 1, March 1981, pp. 140-173.

[Symantec96] *Café Companion*, Symantec Corporation. 1996.

[Sheth&Larson90] A. Sheth, and J. Larson, "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases," *ACM Computing Surveys*, Vol. 22, No. 3, Sept. 1990.

[Smith&Smith77] J. M. Smith, and D. Smith, "Data Abstraction: Aggregation and Generalization," *Communications of the ACM* 20, No. 6, June 1977, pp. 568-579.

[Scholl&Schek92] M.H. Scholl and H. J. Schek, "Survey of COCOON Project," In R. Bayer, T. Harder, and P.C. Lockemann, editors, *Objektbanken Fur Experten*, Informatik Aktuell, October 1992, pp. 243-253. Also available from *http://www.informatik.uni-konstanz.de/~scholl/COCOON-Pub/spp92.ps*.

[SSR94] Michael Siegel, Edward Sciore, and Arnon Rosenthal, "Using Semantic Values to Facilitate Interoperability Among Heterogeneous Information Systems," *ACM Transactions on Database Systems*, June, 1994.

[Thomas90] G. Thomas, "Heterogeneous Distributed Database Systems for Production Use," *ACM Computing Surveys*, Vol. 22, No. 3, Sept., 1990, pp. 237-266.

[Yu&Chang84] C. T. Yu, and C. C. Chang, "Distributed Query Processing," *ACM Computing Surveys*. Vol. 16, No. 4, June 1984

[Weishar93] D. J. Weishar, *A Knowledge-Base Architecture for Query Formulation and Processing in Federated Heterogeneous Database*, Doctoral Dissertation, George Mason University, 1993.

[Wiederhold92] Gio Wiederhold, "The Roles of Artificial Intelligence in Information Systems," *Journal of Intelligent Information Systems 1(1)*, 1992, pp. 35-36.

[Wiederhold95] Gio Wiederhold, "Mediation in Information Systems," *ACM Computing Surveys*, Vol. 27, No. 2, June 1995, pp 265-267.

[Winston&Horn89] Patrick H. Winston, and Berthold K. P. Horn, *LISP 3rd Edition*, Addison-Wesley Publishing Company, 1989.

[W&K91] D. J. Weishar and L. Kerschberg, "An Intelligent Heterogeneous Autonomous Database Architecture for Semantic Heterogeneity Support," *IEEE First International Workshop on Interoperability in Multidatabase Systems*, Kyoto, Japan, April 1991, pp 152-155.

[Zhu&Maier88] J. Zhu and D. Maier, "Abstract Objects in an Object-Oriented Data Model," *Expert Database Systems*, April 1988, pp. 3-16.

Appendices

Appendix A: Prototype High Level Data and Knowledge Structure

The prototype high level data and knowledge structure is presented in this section. First, overview of COOL, the language i which the prototype is implemented, is presented. The second and the third sections present the high level data and knowledge structure for defining the Domain Dependent Knowledge, Application Domain, and the Domain Independent Knowledge, Conversion Knowledge and Reference Knowledge.

A.1 Overview of COOL

COOL is an acronym for 'CLIPS Object-Oriented Language'. COOL is the extension of CLIPS (C Language Integrated Production System) to support object oriented programming language. CLIPS is an expert system tool developed by the Software Technology Branch (STB), NASA/Lyndon B. Johnson Space Center. It was first released in 1986. CLIPS is designed to facilitate the development of software to model human knowledge or expertise. CLIPS has also been designed for full integration with other languages such as C and Ada.

COOL is an object-oriented programming language which supports five generally accepted features of object-oriented programming: classes, message-handlers, abstraction, encapsulation, inheritance, and polymorphism. COOL is a hybrid of features from many different Object Oriented Programming (OOP) systems as well as new ideas. For example, object encapsulation concepts are similar to those in Smalltalk, and the

Common Lisp Object System (CLOS) [Winston&Horn89] provides the basis for multiple inheritance rules. A mixture of ideas from Smalltalk, CLOS and other systems form the foundation of messages.

COOL provides a command for creating classes as well as a query language to query instances of these classes. COOL defines classes through a ‘defclass’ command. A ‘defclass’ is a construct for specifying the properties (slots) of a class of objects. A ‘defclass’ consists of four elements: 1) a name, 2) a list of superclasses from which the new class inherits slots and message-handlers, 3) a specifier saying whether or not the creation of direct instances of the new class is allowed and 4) a list of slots specific to the new class. All user-defined classes must inherit from at least one class, and to this end COOL provides predefined system classes for use as a base in the derivation of new classes. Any slots explicitly given in the ‘defclass’ override those gotten from inheritance. COOL applies rules to the list of superclasses to generate a class precedence list for the new class. Facets further describe slots. Some examples of facets include: default value, cardinality, and types of access allowed.

The syntax of the ‘defclass’ construct is:

(defclass <name> [<comment>]
  (is-a <superclass-name>+)
  [<role>]
  [<pattern-match-role>]
  <slot>*
  <handler-documentation>*)

Where
<role> ::= (role concrete | abstract)
<pattern-match-role> ::= (pattern-match reactive | non-reactive)
<slot> ::= (slot <name> <facet>*) |

(single-slot <name> <facet>*) |
(multislot <name> <facet>*)
<facet> ::= <default-facet> | <storage-facet> |
<access-facet> | <propagation-facet> |
<source-facet> | <pattern-match-facet> |
<visibility-facet> | <create-accessor-facet>
<override-message-facet> | <constraint-attributes>
<default-facet> ::=
(default ?DERIVE | ?NONE | <expression>*) |
(default-dynamic <expression>*)
<storage-facet> ::= (storage local | shared)
<access-facet> ::= (access read-write | read-only | initialize-only)
<propagation-facet> ::= (propagation inherit | no-inherit)
<source-facet> ::= (source exclusive | composite)
<pattern-match-facet> ::= (pattern-match reactive | non-reactive)
<visibility-facet> ::= (visibility private | public)
<create-accessor-facet> ::= (create-accessor ?NONE | read | write | read-write)
<override-message-facet> ::= (override-message ?DEFAULT | <message-name>)
<handler-documentation> ::= (message-handler <name> [<handler-type>])
<handler-type> ::= primary | around | before | after
<constraint-attribute>::=<type-attribute> | <allowed-constant-attribute>|
<range-attribute> |<cardinality-attribute> | <default-attribute>
<type-attribute> ::= (type <type-specification>)
<type-specification> ::= <allowed-type>+ | ?VARIABLE
<allowed-type> ::= SYMBOL | STRING | LEXEME | INTEGER | FLOAT |
NUMBER | INSTANCE | INSTANCE-NAME | INSTANCE-ADDRESS |
EXTERNAL-ADDRESS | FACT-ADDRESS

A.2 High Level Data and Knowledge Structure for Domain Dependent Knowledge

Application domains are classified as a part of Domain Dependent Knowledge. InfoFED supports multiple application domains. The knowledge specification view for the application domain object class is defined as follows:

Application-object-class ::=
CLASS <name> **HAS**
SUPERCLASS (<Superclass-name>+)
SUBCLASS (<Subclass-name>+)
ATTRIBUTE (<value-property-attribute>+)
ATTRIBUTE-UNIT (: <attribute-unit-type :> +)

SYNONYM (<name>+)
DESCRIPTION (<description>)

COOL is the programming language that the system prototype is built on. COOL supports object oriented data representation and provides the Data Definition Language (DDL) for defining object classes. However, to specify the meta-data of classes, the metaclass concept is used. The metaclass concept considers classes as instances of classes called metaclasses. These metaclasses store the meta-data of the class in addition to the class definition. Therefore the knowledge specification view for the above class (Application-object-class) is implemented and is defined into two sections: the class definition and the meta-data definition.

```
; Section I: The Class Definition
(defclass <class-name> [<comment]
  (is-a   <superclass-name>+)
  [<role>]
  <slot>*)
```

Where the detail is described in section A.1

```
; Section II: The Meta-Data Definition
(<class-name> of Application
  (Superclass  <superclass-name>+)
  (Subclass    <subclass-name>+)
  (Attribute   <value-property-attribute-name>+)
  (Attribute-unit       : <attribute-unit-type :> + )
  (Key-attribute        <attribute-name>)
  (Attribute-domain   : <attribute-domain :>+)
  (Synonym   <name>+)
  (Description          <description>)
)
```

Where

```
  <attribute-unit-type> ::= <attribute-name> <defined-unit-type>
  <defined-unit-type> ::= Unit-type <unit>+
  <unit> ::=  <unit-type> ±1
```

<unit-type> ∈ Supported-Unit-type set
<attribute-domain> ::= <attribute-name> <allowed-type>
<allowed-type> ::= <class-name> | string | number

A.2.1 Data Definition for the Application Domain 'AIRLINE'.

The following is the knowledge specification which describes the application domain 'Airline' (Figure 5-2) referring in chapter 5.

Classes Definition Knowledge

The knowledge specification view for the 'Entity' object class is:

```
CLASS  Entity HAS
        SUPERCLASS ( )
        SUBCLASS (Employee Aircraft)
        ATTRIBUTE ( )
        ATTRIBUTE-UNIT ( )
        SYNONYM (Begin Thing Individual Existence)
        DESCRIPTION ("Something having concrete existence")
```

The 'Entity' object class is implemented and defined as:

```
; Section I: The Class Definition
(defclass Entity
        (is-a USER)
        (role abstract)
)
; Section II: The Meta-Data Definition
(Entity of Application
        (Superclass )
        (Subclass Employee Aircraft)
        (Attribute )
        (Attribute-unit )
        (Key-attribute )
        (Attribute-domain )
        (Synonym Begin Thing Individual Existence)
        (Description "Something having concrete existence")
)
```

The knowledge specification view for the 'Employee' object class is:

```
CLASS  Employee HAS
     SUPERCLASS    (Entity )
     SUBCLASS      (Pilot Engineer)
     ATTRIBUTE
        Ssn.value          string
        Ssn.prop           string
        Last-name.value    string
        Last-name.prop     string
        First-name.value   string
        First-name.prop    string
        Salary.value       number
        Salary.prop        string
    ATTRIBUTE-UNIT (: Ssn Unit-type None : Last-name Unit-type None : First-name
        Unit-type None : Salary Unit-type Currency 1 Time-range -1 :)
    SYNONYM       (Worker)
    DESCRIPTION   ("a person who works for another")
```

The ‘Employee’ object class is implemented and defined as:

```
; Section I: The Class Definition
 (defclass Employee
        (is-a Entity)
        (role concrete)
        (multislot Ssn.value  (type STRING) (create-accessor read-write))
        (multislot Ssn.property (type STRING)(create-accessor read-write))
        (multislot F-name.value (type STRING)(create-accessor read-write))
        (multislot F-name.property (type STRING)(create-accessor read-write))
        (multislot L-name.value (type STRING)(create-accessor read-write))
        (multislot L-name.property (type STRING)(create-accessor read-write))
        (multislot Salary.value (type NUMBER)(create-accessor read-write))
        (multislot Salary.property (type STRING)(create-accessor read-write))
)

; Section II: The Meta-Data Definition
(Employee of Application
        (Superclass Entity)
        (Subclass  Pilot Engineer)
        (Attribute Ssn Last-name First-name Salary)
        (Attribute-unit : Ssn Unit-type None : Last-name Unit-type None : First-name
Unit-type None : Salary Unit-type Currency 1 Time-range -1 :)
        (Key-attribute Ssn)
        (Attribute-domain : Ssn Number : Last-name String : First-name String : Salary
Number :)
```

```
        (Synonym Worker)
        (Description "a person who works for another")
)
```

The knowledge specification view for the ‘Pilot’ object class is:

```
CLASS  Pilot HAS
   SUPERCLASS      (Employee)
   SUBCLASS        ()
   ATTRIBUTE
        Ssn.value          string
        Ssn.prop           string
        Last-name.value    string
        Last-name.prop     string
        First-name.value   string
        First-name.prop    string
        Salary.value       number
        Salary.prop        string
        Fly.value          string
        Fly.prop           string
   ATTRIBUTE-UNIT (: Ssn Unit-type None : Last-name Unit-type None : First-name
        Unit-type None : Salary Unit-type Currency 1 Time-range -1 : Fly Unit-type None
        :)
   SYNONYM         (Aviator Flier Airman)
   DESCRIPTION     ("One who flies or qualified to fly an airplane ")
```

The ‘Pilot’ object class is implemented and defined as:

```
; Section I: The Class Definition
 (defclass Pilot
        (is-a Employee)
        (role concrete)
        (multislot Fly.value (type STRING)(create-accessor read-write))
        (multislot Fly.property (type STRING)(create-accessor read-write))
)
;Section II: The Meta-Data Definition
(Pilot of Application
        (Superclass Employee)
        (Subclass  )
        (Attribute Ssn First-name Last-name Salary Fly)
        (Attribute-unit : Ssn Unit-type None : First-name Unit-type None : Last-name
Unit-type None : Salary Unit-type Currency 1 Time-range -1 : Fly Unit-type None :)
        (Key-attribute Ssn)
```

```
        (Attribute-domain : Ssn Number : First-name String : Last-name String : Salary
Number : Repair Aircraft :)
        (Synonym Aviator Flier Airman)
        (Description "One who flies or qualified to fly an airplane")
)
```

The knowledge specification view for the ‘Engineer’ object class is:

```
CLASS  Engineer HAS
   SUPERCLASS      (Employee)
   SUBCLASS        ()
   ATTRIBUTE
        Ssn.value          string
        Ssn.prop           string
        Last-name.value    string
        Last-name.prop     string
        First-name.value   string
        First-name.prop    string
        Salary.value       number
        Salary.prop        string
        Repair.value       string
        Repair.prop        string
   ATTRIBUTE-UNIT (: Ssn Unit-type None : Last-name Unit-type None : First-name
        Unit-type None : Salary Unit-type Currency 1 Time-range -1 : Repair Unit-type
        None :)
   SYNONYM         ()
   DESCRIPTION     ("A designer or a builder of engines")
```

The ‘Engineer’ object class is implemented and defined as:

```
; Section I: The Class Definition
 (defclass Engineer
        (is-a Employee)
        (role concrete)
        (multislot Repair.value (type STRING)(create-accessor read-write))
        (multislot Repair.property (type STRING)(create-accessor read-write))
)
; Section II: The Meta-Data Definition
(Pilot of Application
        (Superclass Employee)
```

```
        (Subclass  )
        (Attribute Ssn First-name Last-name Salary Repair)
        (Attribute-unit : Ssn Unit-type None : First-name Unit-type None : Last-name
Unit-type None : Salary Unit-type Currency 1 Time-range -1 : Repair Unit-type None :)
        (Key-attribute Ssn)
        (Attribute-domain : Ssn Number : First-name String : Last-name String : Salary
Number : Repair Aircraft :)
        (Synonym )
        (Description "A designer or a builder of engines")
)
```

The knowledge specification view for the ‘Aircraft’ object class is:

```
CLASS  Aircraft HAS
   SUPERCLASS      (Entity)
   SUBCLASS        ()
   ATTRIBUTE
        Aircraft.value      string
        Aircraft.prop       string
        Air-type.value      string
        Air-type.prop       string
        Weight.value        number
        Weight.prop         string
        Fly-range.value     number
        Fly-range.prop      string
   ATTRIBUTE-UNIT (: Aircraft Unit-type None : Air-type Unit-type None : Weight
        Unit-type Weight 1 : Fly-range Unit-type Length 1 :)
   SYNONYM         (Plane)
   DESCRIPTION     ("A machine that can fly")
```

The ‘Aircraft’ object class is implemented and defined as:

```
; Section I: The Class Definition
 (defclass Aircraft
        (is-a Entity)
        (role concrete)
        (multislot Aircraft.value (type STRING)(create-accessor read-write))
        (multislot Aircraft.property (type STRING)(create-accessor read-write))
        (multislot Air-type.value (type STRING)(create-accessor read-write))
        (multislot Air-type.property (type STRING)(create-accessor read-write))
        (multislot Weight.value (type NUMBER)(create-accessor read-write))
        (multislot Weight.property (type STRING)(create-accessor read-write))
        (multislot Fly-range.value (type NUMBER)(create-accessor read-write))
```

```
        (multislot Fly-range.property (type STRING)(create-accessor read-write))
)
; Section II: The Meta-Data Definition
(Aircraft of Application
        (Superclass Entity)
        (Subclass )
        (Attribute Aircraft Air-type Weight Fly-range)
        (Attribute-unit : Aircraft Unit-type None : Air-type Unit-type None : Weight Unit-
type Weight 1 : Fly-range Unit-type Length 1 :)
        (Key-attribute Aircraft)
        (Attribute-domain : Aircraft String : Air-type String : Weight Number : Fly-range
Number :)
        (Synonym Plane)
        (Description "A machine that can fly")
)
```

A.3 High Level Data and Knowledge Structure for Domain Independent Knowledge

The Unit Value Mediator requires two types of independent knowledge, conversion knowledge and reference knowledge, to support its conversion services. The four unit types which are implemented and supported are ‘Currency’, ‘Length’, Time-range’, and ‘Weight’.

Supported-unit-type = {Currency, Length, Time-range, Weight}

The supported unit values for each unit type are:

Currency-supported-unit-value = {usd, cnd, jpy, dem, thb}
Length-supported-unit-value = {millimeter, centimeter, meter, kilometer, inch, foot, yard, rod, mile}
Time-range-supported-unit-value = {second, minute, hour, day, week, month, year}
Weight-supported-unit-value = {milligram, gram, kilogram, grain, dram, ounce, pound, short-ton, long-ton, metric-ton}

Both high level knowledge specification of conversion knowledge and reference knowledge are presented in this section.

A.3.1 Conversion Knowledge

The knowledge specification view of the conversion knowledge object class is defined as follows:

Conversion Class ::=
CLASS <name> **HAS**
  SUPERCLASS (<Superclass-name>)
  SUBCLASS (<Subclass-name>)
  DEFAULT-CALL (<Instance Object's id>)
  DESCRIPTION (<description>)
  ATTRIBUTE
    Program.value string
    Program.prop string
    Designer.value string
    Designer.prop string
    Version.value string
    Version.prop string

The knowledge specification view for the above class (Conversion-object-class) is implemented and is defined into two sections: the class definition and the meta-data definition.

; Section I: The Class Definition
(defclass <class-name> [<comment]
  (is-a <superclass-name>+)
  [<role>]
  <slot>*)
Where the details are described in section A.1

; Section II: The Meta-Data Definition
(<class-name> of Conversion-knowledge
  (Superclass <superclass-name>)
  (Subclass <subclass-name>+)
  (Attribute <value-property-attribute-name>+)
  (Attribute-unit : <attribute-unit-type :>+)
  (Default-call <object-id>)
  (Description <description>)

)

Where
<attribute-unit-type> ::= <attribute-name> <defined-unit-type>
<defined-unit-type> ::= Unit-type <unit>+
<unit> ::= <unit-type> ±1
<unit-type> ∈ Supported-Unit-type set
<attribute-domain> ::= <attribute-name> <allowed-type>
<allowed-type> ::= <class-name> | string | number

The knowledge specification view for the ‘Conversion’ object class is:

CLASS Conversion HAS
SUPERCLASS ( )
SUBCLASS (Currency Length Time-range Weight)
(Default-call C123)
ATTRIBUTE
Program.value string
Program.prop string
Designer.value string
Designer.prop string
Version.value string
Version.prop string
ATTRIBUTE-UNIT (: Program Unit-type None : Designer Unit-type None : Version Unit-type None :)
SYNONYM ( )
DESCRIPTION ("Conversion knowledge for different types of conversions")

The ‘Conversion’ object class is implemented and defined as:

```
; Section I: The Class Definition
(defclass Conversion
   (is-a USER)
   (role concrete)
   (multislot Program.value (type STRING)(create-accessor read-write))
   (multislot Program.prop (type STRING)(create-accessor read-write))
   (multislot Designer.value (type STRING)(create-accessor read-write))
   (multislot Designer.prop (type STRING)(create-accessor read-write))
   (multislot Version.value (type STRING)(create-accessor read-write))
   (multislot Version.prop (type STRING)(create-accessor read-write))
)
; Section II: the Meta-Data Definition
(Conversion of Conversion-knowledge
```

```
   (Superclass None)
   (Subclass Currency Weight Time-range Length)
   (Default-call )
   (Attribute Program Designer Version)
   (Attribute-domain : Program String : Designer String : Version Unit-type None :)
   (Description "Conversion knowledge for different types of conversions")
)
```

The knowledge specification view for the ‘Currency’ object class is:

```
CLASS  Currency HAS
   SUPERCLASS      (Conversion)
   SUBCLASS        ()
   DEFAULT-CALL  ([C001])
   ATTRIBUTE
         Program.value      string
         Program.prop       string
         Designer.value     string
         Designer.prop      string
         Version.value      string
         Version.prop       string
   ATTRIBUTE-UNIT (: Program Unit-type None : Designer Unit-type None : Version
         Unit-type None :)
   SYNONYM         ( )
   DESCRIPTION     ("Conversion knowledge for the currency")
```

The ‘Currency’ object class is implemented and defined as:

```
; Section I: The Class Definition
(defclass Currency
   (is-a Conversion)
   (role concrete)
)
; Section II: the Meta-Data Definition
(Currency of Conversion-knowledge
   (Default-call [C001])
   (Superclass Conversion)
   (Subclass None)
   (Attribute Program Designer)
   (Attribute-domain : Program Unit-type String : Designer Unit-type String : Version
Unit-type String :)
   (Description "Conversion knowledge for the currency ")
)
```

The knowledge specification view for the ‘Length’ object class is:

```
CLASS  Length HAS
   SUPERCLASS      (Conversion)
   SUBCLASS        ()
   DEFAULT-CALL  ([D001])
   ATTRIBUTE
         Program.value      string
         Program.prop       string
         Designer.value     string
         Designer.prop      string
         Version.value      string
         Version.prop       string
   ATTRIBUTE-UNIT (: Program Unit-type None : Designer Unit-type None : Version
         Unit-type None :)
   SYNONYM         ( )
   DESCRIPTION     ("- the space between two objects or points
                     - Conversion knowledge for Length unit")
```

The ‘Length’ object class is implemented and defined as:

```
; Section I: The Class Definition
(defclass Length
   (is-a Conversion)
   (role concrete)
)
; Section II: the Meta-Data Definition
(Length of Conversion-knowledge
   (Default-call [D001])
   (Superclass Conversion)
   (Subclass None)
   (Attribute Program Designer)
   (Attribute-domain : Program String : Designer String :)
   (Description "- the space between two objects or points
                  - Conversion knowledge for Length unit")
)
```

The knowledge specification view for the ‘Time-range’ object class is:

```
CLASS  Time-range HAS
   SUPERCLASS      (Conversion)
   SUBCLASS        ()
```

```
   DEFAULT-CALL  ([T001])
   ATTRIBUTE
        Program.value      string
        Program.prop       string
        Designer.value     string
        Designer.prop      string
        Version.value      string
        Version.prop       string
   ATTRIBUTE-UNIT (: Program Unit-type None : Designer Unit-type None : Version
        Unit-type None :)
   SYNONYM          ( )
   DESCRIPTION      ("- Conversion knowledge for time unit ")
```

The ‘Time-range’ object class is implemented and defined as:

```
; Section I: The Class Definition
(defclass Time-range
   (is-a Conversion)
   (role concrete)
)
; Section II: the Meta-Data Definition
(Time-range of Conversion-knowledge
   (Default-call [T001])
   (Superclass Conversion)
   (Subclass None)
   (Attribute Program Designer)
   (Attribute-domain : Program String : Designer String :)
   (Description "- Conversion knowledge for time unit ")
)
```

The knowledge specification view for the ‘Weight’ object class is:

```
CLASS  Weight HAS
   SUPERCLASS      (Conversion)
   SUBCLASS        ()
   DEFAULT-CALL  ([W001])
   ATTRIBUTE
        Program.value      string
        Program.prop       string
        Designer.value     string
        Designer.prop      string
        Version.value      string
        Version.prop       string
```

ATTRIBUTE-UNIT (: Program Unit-type None : Designer Unit-type None : Version Unit-type None :)
SYNONYM ( )
DESCRIPTION ("- a system of units used to express the weight of something
- Conversion Knowledge for weight ")

The 'Weight' object class is implemented and defined as:

```
; Section I: The Class Definition
(defclass Weight
   (is-a Conversion)
   (role concrete)
)
; Section II: the Meta-Data Definition
(Weight of Conversion-knowledge
   (Default-call [W001])
   (Superclass Conversion)
   (Subclass None)
   (Attribute Program Designer)
   (Attribute-domain : Program String : Designer String : Version String :)
   (Description "- a system of units used to express the weight of something
                - Conversion Knowledge for weight ")
)
```

Reference Knowledge

The knowledge specification view of the reference knowledge object class is defined as follows:

Reference Class ::=
CLASS <class-name> **HAS**
SUPERCLASS (<superclass-name>)
SUBCLASS (<subclass-name>+)
UNIT-TYPE (<unit type>)
SUPPORTED-UNIT (<unit value>+)
DESCRIPTION (<description>)
ATTRIBUTE
From.value string
From.prop string
To.value string
To.prop string

Rate.value	number
Rate.prop	string
Version.value	string
Version.prop	string
Active.value	string
Active.prop	string

The knowledge specification view for the above class (Reference-object-class) is implemented and is defined into two sections: the class definition and the meta-data definition.

; Section I: The Class Definition
(defclass <class-name> [<comment]
  (is-a <superclass-name>+)
  [<role>]
  <slot>*)
Where the details are described in section A.1

; Section II: The Meta-Data Definition
(<class-name> of Conversion-knowledge
  (Superclass <superclass-name>)
  (Subclass <subclass-name>+)
  (Attribute <value-property-attribute-name>+)
  (Attribute-unit : <attribute-unit-type :> +)
  (Unit-type <unit-type>)
  (Supported-unit <unit-value>+)
  (Attribute-domain : <attribute-domain :>+)
  (Description <description>)
)

Where
  <attribute-unit-type> ::= <attribute-name> <defined-unit-type>
  <defined-unit-type> ::= Unit-type <unit>+
  <unit> ::= <unit-type> ±1
  <unit-type> ∈ Supported-Unit-type set
  <attribute-domain> ::= <attribute-name> <allowed-type>
  <allowed-type> ::= <class-name> | string | number

The knowledge specification view for the ‘Reference’ object class is:

```
CLASS  Reference HAS
   SUPERCLASS      ()
   SUBCLASS        (Exchange-rate Length-rate Time-range-rate Weight-rate)
   UNIT-TYPE       ()
   SUPPORTED-UNIT  ()
   ATTRIBUTE
        From.value        string
        From.prop         string
        To.value          string
        To.prop           string
        Rate.value        number
        Rate.prop         string
        Valid.value       string
        Valid.prop        string
        Version.value     string
        Version.prop      string
   ATTRIBUTE-UNIT (: From Unit-type None : To Unit-type None : Rate Unit-type
        None : Valid Unit-type None : Version Unit-type None :)
   SYNONYM         ( )
   DESCRIPTION     ("conversion rate for conversion among conversion unit values")
```

The ‘Reference’ object class is implemented and defined as:

```
; Section I: The Class Definition
(defclass Reference
   (is-a USER)
   (role concrete)
   (multislot From.value (type STRING) (create-accessor read-write))
   (multislot From.prop (type STRING)(create-accessor read-write))
   (multislot To.value (type STRING)(create-accessor read-write))
   (multislot To.prop (type STRING)(create-accessor read-write))
   (multislot Rate.value (type NUMBER)(create-accessor read-write))
   (multislot Rate.prop (type STRING)(create-accessor read-write))
   (multislot Valid.value (type STRING)(create-accessor read-write))
   (multislot Valid.prop (type STRING)(create-accessor read-write))
   (multislot Version.value (type STRING)(create-accessor read-write))
   (multislot Version.prop (type STRING)(create-accessor read-write))
)
; Section II: the Meta-Data Definition
(Reference of Reference-knowledge
   (Superclass None)
   (Subclass Exchange-rate Length-rate Time-range-rate Weight-rate)
   (Attribute From To Rate Valid Version)
```

```
   (Attribute-unit : From Unit-type None : To Unit-type None : Rate Unit-type None :
Valid Unit-type None : Version Unit-type None :)
   (Attribute-domain : From String : To String : Rate Number : Valid String : Version
String :)
   (Description "conversion rate for conversion among conversion unit values")
)
```

The knowledge specification view for the ‘Exchange-rate’ object class is:

```
CLASS  Exchange-rate HAS
   SUPERCLASS      (Reference)
   SUBCLASS        ()
   UNIT-TYPE       (Currency)
   SUPPORTED-UNIT  (usd cnd frf dem jpy mxp thb)
   ATTRIBUTE
         From.value        string
         From.prop         string
         To.value          string
         To.prop           string
         Rate.value        number
         Rate.prop         string
         Valid.value       string
         Valid.prop        string
         Version.value     string
         Version.prop      string
   ATTRIBUTE-UNIT (: From Unit-type None : To Unit-type None : Rate Unit-type
         None : Valid Unit-type None : Version Unit-type None :)
   SYNONYM         ( )
   DESCRIPTION     ("conversion rate for conversion among currency unit values")
```

The ‘Exchange-rate’ object class is implemented and defined as:

```
; Section I: The Class Definition
(defclass Exchange-rate
   (is-a Reference)
   (role concrete)
)
; Section II: the Meta-Data Definition
(Exchange-rate of Reference-knowledge
   (Superclass Reference)
   (Subclass None)
   (Attribute From To Rate Valid Version)
```

(Attribute-unit : From Unit-type None : To Unit-type None : Rate Unit-type None : Valid Unit-type None : Version Unit-type None :)
(Attribute-domain : From String : To String : Rate Number : Valid String : Version String :)
(Unit-type Currency)
(Unit-support usd cnd frf dem jpy mxp thb)
(Description "conversion rate for conversion among currency unit values")
)

The knowledge specification for the ‘Length-rate’ object class is:

```
CLASS  Length-rate HAS
   SUPERCLASS      (Reference)
   SUBCLASS        ()
   UNIT-TYPE       (Length)
   SUPPORTED-UNIT(millimeter centimeter meter kilometer inch foot yard rod mile)
   ATTRIBUTE
        From.value        string
        From.prop         string
        To.value          string
        To.prop           string
        Rate.value        number
        Rate.prop         string
        Valid.value       string
        Valid.prop        string
        Version.value     string
        Version.prop      string
   ATTRIBUTE-UNIT (: From Unit-type None : To Unit-type None : Rate Unit-type
        None : Valid Unit-type None : Version Unit-type None :)
   SYNONYM         ( )
   DESCRIPTION     ("conversion rate for conversion among length unit values")
```

The ‘Length-rate’ object class is implemented and defined as:

```
; Section I: The Class Definition
(defclass Length-rate
   (is-a Reference)
   (role concrete)
)
; Section II: the Meta-Data Definition
(Length-rate of Reference-knowledge
   (Superclass Reference)
   (Subclass None)
```

```
   (Attribute From To Rate Valid Version)
   (Attribute-unit : From Unit-type None : To Unit-type None : Rate Unit-type None :
Valid Unit-type None : Version Unit-type None :)
   (Attribute-domain : From String : To String : Rate Number : Valid String : Version
String :)
   (Unit-type  Length)
   (Unit-support millimeter centimeter meter kilometer inch foot yard rod mile)
   (Description "conversion rate for conversion among length unit values")
)
```

The knowledge specification view for the ‘Time-range-rate’ object class is:

```
CLASS  Time-range-rate HAS
   SUPERCLASS      (Reference)
   SUBCLASS        ()
   UNIT-TYPE       (Time-range)
   SUPPORTED-UNIT  (second minute hour day week month year)
   ATTRIBUTE
         From.value        string
         From.prop         string
         To.value          string
         To.prop           string
         Rate.value        number
         Rate.prop         string
         Valid.value       string
         Valid.prop        string
         Version.value     string
         Version.prop      string
   ATTRIBUTE-UNIT (: From Unit-type None : To Unit-type None : Rate Unit-type
         None : Valid Unit-type None : Version Unit-type None :)
   SYNONYM         ( )
   DESCRIPTION     ("conversion rate for conversion among time unit values ")
```

The ‘Time-range-rate’ object class is implemented and defined as:

```
; Section I: The Class Definition
(defclass Time-range-rate
   (is-a Reference)
   (role concrete)
)
```

```
; Section II: the Meta-Data Definition
(Time-range-rate of Reference-knowledge
   (Superclass Reference)
   (Subclass None)
   (Attribute From To Rate Valid Version)
   (Attribute-unit : From Unit-type None : To Unit-type None : Rate Unit-type None :
Valid Unit-type None : Version Unit-type None :)
   (Attribute-domain : From String : To String : Rate Number : Valid String : Version
String :)
   (Unit-type  Time-range)
   (Unit-support second minute hour day week month year)
   (Description "conversion rate for conversion among time unit values")
```

The knowledge specification view for the ‘Weight-rate’ object class is:

```
CLASS  Weight-rate HAS
   SUPERCLASS      (Reference)
   SUBCLASS        ()
   UNIT-TYPE       (Weight)
   SUPPORTED-UNIT (milligram, gram, kilogram, grain, dram, ounce, pound, short-ton,
         long-ton, metric-ton)
   ATTRIBUTE
         From.value        string
         From.prop         string
         To.value          string
         To.prop           string
         Rate.value        number
         Rate.prop         string
         Valid.value       string
         Valid.prop        string
         Version.value     string
         Version.prop      string
   ATTRIBUTE-UNIT (: From Unit-type None : To Unit-type None : Rate Unit-type
         None : Valid Unit-type None : Version Unit-type None :)
   SYNONYM         ( )
   DESCRIPTION     ("conversion rate for conversion among weight unit values")
```

The ‘Weight-rate’ object class is implemented and defined as:

```
; Section I: The Class Definition
(defclass Weight-rate
   (is-a Reference)
   (role concrete)
```

```
)
; Section II: the Meta-Data Definition
(Weight-rate of Reference-knowledge
   (Superclass Reference)
   (Subclass None)
   (Attribute From To Rate Valid Version)
   (Attribute-unit : From Unit-type None : To Unit-type None : Rate Unit-type None : Valid Unit-type None : Version Unit-type None :)
   (Attribute-domain : From String : To String : Rate Number : Valid String : Version String :)
   (Unit-type  Weight)
   (Unit-support milligram, gram, kilogram, grain, dram, ounce, pound, short-ton, long-ton, metric-ton)
   (Description "conversion rate for conversion among weight unit values")
)
```

Appendix B: The Extended Query Language Specification

The Mediation Data Model was developed using COOL language. The Mediation Data Model provides the query language to query data that is stored in the model. A query is a user-defined Boolean expression that is applied to an instance set to determine if the instance set meets user-defined restrictions. The query language specification for retrieving the data is as follows:

SELECT (<attribute>+)
UNIT-AS (<unit-value>+)
FROM (<instance-set>+)
WHERE (<condition>) | TRUE

Where
<attribute>::= attr-name | attr-name.value | attr-name.prop |
attr-name.prop.unit-type | attr-name.prop.source
<unit-value>::= (= attr-name.perop.unit-type <Unit-type DMD>)
<instance-set>::= (?variable <class-name>)
<condition>::= <Boolean-expression>

The Boolean expression is composed of well-formed predicate functions, math functions, etc. The CLIPS function specifications are fully described in [GR93, NASA93a]. As object classes' attributes support the Multiple Unit Value capabilities, these functions must be extended so that they can handle data with the unit values. Since the unit type information is associated with the attributes that have domains in numeric expression, binary functions that deal with variables in the numeric expression must be extended. By using functions' name overload concept, when functions are dealing with

attributes that have no unit type associated with them, only value attributes are used as parameters in the functions. However, when functions are dealing with attributes that have unit types associated with them, value attributes are used together with attribute unit type DMD as parameters in the functions.

Below are some of binary functions that are extended to supported data with unit value.

Predicate functions

(φ <data-value$_1$><data-value$_2$>) (I)
(φ <data-value$_1$> <Unit-type DMD$_1$> <data-value$_2$> <Unit-type DMD$_2$>) (II)

Where

φ ::= > | >= | < | <= | = | <>

> is a greater than operation.
The function returns TRUE if the first data value argument is greater than the second data value argument, otherwise FALSE.

>= is a greater than or equal operation.
The function returns TRUE if the first data value argument is greater than or equal to the second data value argument, otherwise FALSE.

< is a less than operation
The function returns TRUE if the first data value argument is less than the second data value argument, otherwise FALSE.

<= is a less than or equal operation
The function returns TRUE if the first data value argument is less than or equal to the second data value argument, otherwise FALSE.

= is an equal operation
The function returns TRUE if the first data value argument is equal to the second data value argument, otherwise FALSE.

<> is an not equal operation
The function returns TRUE if the first data value argument is not equal to the second data value argument, otherwise FALSE.

<data-value$_1$> ::= <numeric-expression>
<data-value$_2$> ::= <numeric-expression>
<Unit-type DMD$_1$> ::= Unit-type DMD
<Unit-type DMD$_2$> ::= Unit-type DMD

Math functions

(φ <data-value$_1$><data-value$_2$>) (I)
(φ <data-value$_1$> <Unit-type DMD$_1$> <data-value$_2$> <Unit-type DMD$_2$>) (II)
Where

φ ::= + | -

\+ is an add operation.

Case I. The function returns the sum of its data value arguments.

Case II. The function returns the sum of its data value arguments with a unit value of the first data value.

\- is a subtraction operation.

Case I. The function returns the result of the first data value argument minus the second data value argument.

Case II. The function returns the result of the first data value argument minus the second data value argument with a unit value of the first data value.

<data-value$_1$> ::= <numeric-expression>
<data-value$_2$> ::= <numeric-expression>
<Unit-type DMD$_1$> ::= Unit-type DMD
<Unit-type DMD$_2$> ::= Unit-type DMD

Curriculum Vitae

Wiput Phijaisanit was born on July 20, 1966, in Bangkok, Thailand, and is a Thai citizen. He graduated from Triam Udom Suksa High School, Bangkok, Thailand in 1984. He received his Bachelor of Engineering in Electrical Engineering from King Mongkut's Institute of Technology Landkrabang, Bangkok, Thailand in 1988. He worked at Asea Brownboveri in Thailand as a Substation Design Engineer from 1988 to 1990. He received his Master of Business Administration in Management from Oklahoma City University, Oklahoma City, Oklahoma in 1991.