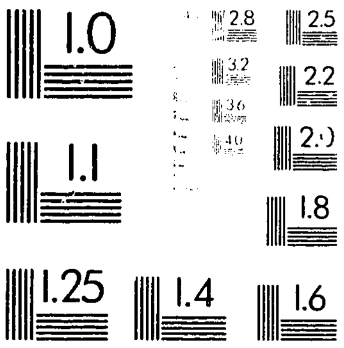
1



University of Alberta

Rainbow: Prototyping the DIOM Interoperable System

by

Yoo-Shin Lee ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 1996

ISBN 0-612-18287-8

University of Alberta

Library Release Form

Name of Author: Yoo-Shin Lee

Title of Thesis: Rainbow: Prototyping the DIOM Interoperable System

Degree: Master of Science

Year this Degree Granted: 1996

[illegible]

Yoo-Shin Lee
4016-77 Street NW
Edmonton, Alberta
Canada, T6K 0X6

Date: July. 30. '96

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Rainbow: Prototyping the DIOM Interoperable System** submitted by Yoo-Shin Lee in partial fulfillment of the requirements for the degree of **Master of Science**.

[illegible]

Dr. L. Liu (Supervisor)

[illegible]

Dr. M. T. Özsu (Internal)

[illegible]

Dr. Dennis Ward (External)

Date: July 23, 96

To my parents, Yoo-Jin, and my grandmother in memoriam

Y.

Abstract

The goal of the DIOM (Distributed Interoperable Object Model) project is the development of adaptive methodologies and toolkits for the integration and access of heterogeneous information sources in large-scale and rapidly growing network environments.

The Rainbow prototype system aims at developing a toolkit based on the DIOM query mediation method to facilitate the interconnection between information consumers and information producers. The Rainbow system consists of several components which extract and collect data from disparate data repositories or information brokers/mediators and then convert this gathered information into a representation specified by the information consumer.

The main services that the DIOM system provides include producer information source registration, metadata library management, query routing, query decomposition, parallel query plan generation subquery transformation and execution, and result assembly. This thesis presents the design and implementation of Rainbow, a prototype of the DIOM adaptive query mediation methodology.

Acknowledgements

This thesis could not have been produced without the contributions of several special people.

First and foremost, I am deeply indebted to my supervisor, Dr. Ling Liu, for her direction, patience, and funding. Not only has she provided me with a first class education into the world of interoperable database research, but she was also a constant source of positive encouragement and inspiration. I thank Dr. Liu for her invaluable advice on academic related matters and her dedication to my program and future. Dr. Liu's work with Dr. Calton Pu provided the fundamental background material for this thesis.

I thank the members of my examining committee, Dr. Özsu and Dr. Ward, who thoroughly reviewed this work and offered excellent constructive criticisms. I also thank the examining committee chair, Dr. Goebel, who made me feel at ease and was flawless in his duties.

Acknowledgement goes out to past members of the DIOM project group, Karim Makhani and Uwe Thiemann, who both offered excellent technical support and ideas. I also appreciate the caring support I received from the support staff and administrative staff in the department of Computing Science.

Contents

List of Figures

Chapter 1

Introduction

1.1 Thesis Motivation

The explosion of networking technology has led to the ubiquitous Internet and the large-scale availability of diverse information sources. Although there is much debate on the accuracy of Internet measurements, a study [55] done in 1995 conservatively estimates that there are ten million hosts [1] and twenty million users on the Internet with numbers doubling roughly every year. In addition to these growing numbers, the loose reign on Internet development has led to many protocols for navigating and obtaining information from the Internet. The current situation of numerous incompatible Internet protocols, the large number of users seeking information, and a wide variety of hosts providing infor nation has resulted in user frustration and the inefficient utilization of resources. These issues have not been adequately resolved by current Internet service tools.

Information consumers, those seeking information, are left with the daunting tasks of contending with data overload, resolving information heterogeneity, learning different information access methods, and compiling the information obtained. The development of the World Wide Web and the proliferation of easy-to-use navigational tools (e.g., Netscape, Internet Explorer, Mosaic, etc.) have gone far to facilitate the availability of online information but still leave much of the searching and assembly of information up to the user. Users are spending approximately 79% of online time browsing [21], therefore it is estimated that much bandwidth today is wasted due to inefficient searching and ineffective assembly of result data. Without more intelligent information tools, the great potential to information that this new networking technology offers is wasted.

In terms of object-oriented terminology, *interoperability* refers to the ability to exchange requests between objects and to enable objects to request services of other objects, regardless of the internal object differences (e.g., hardware platforms, operating systems, data models). *Distributed interoperable objects* are objects that support a level of interoperability beyond the traditional object computing boundaries imposed by programming languages, data models, process address space, and network interfaces [7].

One way to support information gathering across heterogeneous and autonomous information sources is to approach the problems from an interoperability standpoint. Therefore, the challenge is to build cooperative systems for large-scale environments which are able to connect information consumers with information producers while providing transparent and

[1] the data on these hosts will only be useful for specific domains and not all hosts will offer public or even useful data

customizable information access across multiple heterogeneous and autonomous information sources (including databases, knowledge bases, flat files, or programs). For an interoperable system to be effective in large-scale computing environments requires attention to the USECA properties [31] which are summarized here:

Uniform access

Uniform access pertains to the differences in accessing heterogeneous information sources. We can categorize these differences as network access protocol differences and information language differences. When attempting to access an information source in a large scale environment, a user must first contend with choosing and obtaining the tools for utilizing the correct Internet protocol. In the Internet world there are dozens of different application protocols (e.g. WAIS, FTP, GOPHER, TELNET, HTTP, STMP, Vendor Specific etc.) and multiple data access languages (e.g., SQL, Vendor Specific APIs, HTTP FORM, etc.). Thus, obtaining information can become quite cumbersome with results being obtained in different and possibly incompatible formats. In a large scale environment the efficient exchange of information is greatly hindered by the differences in repository access. An interoperable system without uniform access would require queries to contain a list of relevant sites, methods with which to access these sites, and the terminology required to invoke local queries.

Scalability

Scalability is a necessary requirement for the Internet services. As millions of new hosts can come online within a space of months, effective queries in this large-scale environment must be able to utilize all sources that are relevant. Query processing facilities should be able to incorporate new sources as they are added, without rewriting the code and redesigning the system infrastructure.

Evolution

Evolution refers to ability of a system to adapt to the dynamic nature of the Internet growth. In the Internet environment, individual components may temporarily cease to operate or become obsolete. Furthermore, even when the changes to individual component repositories are infrequent, the large number of databases may add up to a surprisingly large number of change events in the system. For example, assuming on average there is only one change in two years for any component database schema, an interoperable system with two hundred databases will have to contend with one hundred changes a year, which on average corresponds to two changes every week.

Composibility

System composibility refers to the modularity and reusability of the system design and implementation, and the need for incremental design and construction of interoperation interfaces. When dealing with large scale systems, modularizing the design into small implementable and manageable objects makes practical sense. It would also be quite desirable to reuse much of the objects, so whenever it is possible, system functionality should incorporate the object-oriented design abstractions such as generalization, specialization, and aggregation.

Autonomy
A reasonable and preferably minimal amount of repository modification is required when repositories participate in interoperation with other repositories. Imposing a global schema is not practical in an environment where autonomy of the individual information sources is important. To respect the autonomy of individual systems, it is important that we:

- treat the external applications and any interoperable system similar to its ordinary clients.

- use data model independent language constructs and facilities to allow users to define their views of data in their preferred representations, so users can formulate queries and receive query results based on their own understanding of data

User autonomy is another issue. It would facilitate user querying of sources, if users were not made to compromise the way in which they prefer to understand their domain semantics and were not forced to surrender to a system-supplied, canonical integration schema.

In addition to the USECA properties, there are other system development requirements which are important. For example, efficient distributed query processing, security, automated semantic representation and matching are other important areas of concern systems should consider.

Our approach to a large-scale cooperative system is based on the Distributed Interoperable Object Model (DIOM) [31]. The DIOM mediation architecture features the support of the USECA properties in the development of distributed object query services, and the flexible cooperation of a network of mediators for transparent query processing. This research attempts to prototype parts of the DIOM model by developing the Rainbow mediation toolkit and by building the basic framework for the eventual completion of the overall system in plementation.

Th[illegible] include:

- T[illegible] the Rainbow implementation architecture and the design of functionality for the [illegible] prototype.

- The selection of software environment tools for the implementation development.

- The coding and testing of the implementation using a real-world application.

The next section will detail the motivation and the importance of the DIOM query mediation framework by presenting an example application scenario and by studying the heterogeneity problems which hinder the effective retrieval of information.

1.2 Heterogeneity Issues

A real world multidatabase system can include tens or hundreds of information sources separately managed by different types of DBMSs, simple file systems, or knowledge base systems. The heterogeneity of information is natural and unavoidable. The individual information sources are often heterogeneous in logical data structure, in type and attribute naming, and in representational semantics. This heterogeneity in information source models results in heterogeneity in query formulation, query processing, and query result assembly.

1.2.1 Application Scenario

A real-world application scenario will be used as a running example throughout this thesis to illustrate important features and describe the Rainbow implementation. The application scenario is as follows: A multimedia expert is searching for jobs located in the USA or Canada which are related to her domain of expertise.

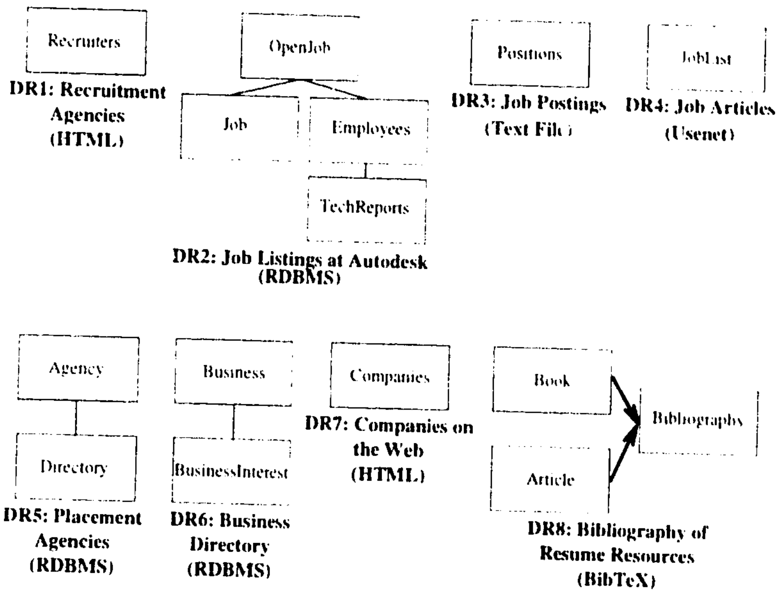

Recruiters
DR1: Recruitment Agencies (HTML)
OpenJob
Job
Employees
TechReports
DR2: Job Listings at Autodesk (RDBMS)
Positions
DR3: Job Postings (Text File)
JobList
DR4: Job Articles (Usenet)
Agency
Directory
DR5: Placement Agencies (RDBMS)
Business
BusinessInterest
DR6: Business Directory (RDBMS)
Companies
DR7: Companies on the Web (HTML)
Book
Article
Bibliographys
DR8: Bibliography of Resume Resources (BibTeX)

Figure 1.1: Examples of information sources

Assume that the relevant information sources (as shown in Figure 1.1) that are currently available include a Recruitment Agencies HTML document, a Job Listing database at Autodesk [2], a Job Postings text file, Job Articles from Usenet News, a Placement Agency Database, a Business Directory Database, a Companies on the Web HTML document, and a Resume Resources BibTeX file. We refer to these information sources as component information sources of the given enterprise system. The relations or classes are described as boxes and the links indicate the primary key and foreign key relationships or object reference relationships. The superclass and subclass relationships are depicted using arrow links. Figure 1.2 contains fragments of the export schemas for these information sources.

In order to find most of the relevant job opportunities, the job seeker (information consumer) would have to search and possibly revisit many different data sources. This task may involve browsing and navigating through the above mentioned repositories. What is required is a more efficient and manageable system to connect this consumer to these job repositories and make querying easier over these heterogeneous repositories.

Automating the connection between information consumers with relevant information producers requires: (1) gathering knowledge about the type of data the consumer desires, (2) gathering knowledge about the type of information a producer supplies, and then (3) efficiently matching consumers with producers. Hence, new information technologies must use information about the information (metadata) to automate the abstraction of data and mediate heterogeneity problems.

[2]Autodesk is a trademark of Autodesk Inc.

Recruiters=[name, city, state, mail_code, phone, e-mail, description] **DR1: Recruitment Agencies (HTML)**	Job=[jobtitle, description, start_wage, max_wage] OpenJob=[jobtitle, opendate, deadline_date, department, internal] Employees=[last_name, first_name, sin, dob, sex, jobtitle] TechReports=[report_id, author, date, keywords] **DR2: Job Listings at Autodesk (RDBMS)**
Positions=[title, description, company, pay] **DR3: Job Postings (Text File)**	JobList=[position, description, organization] **DR4: Job Articles (Usenet)**
Agency=[name, description] Directory=[name, address, state, city, country, zip_code] **DR5: Placement Agencies (RDBMS)**	Business=[name, street, city, state, country, zip] BusinessInterest=[name, description, investment_level] **DR6: Business Directory (RDBMS)**
Companies=[name, address, city, province, country, postal_code, homepage] **DR7: Companies (HTML)**	Book=[author, title, publisher, address, year] Article=[author, title, journal, year, pages] **DR8: Bibliography of Resume Resources (BibTeX)**

Figure 1.2: Fragments of the export schemas of example information sources

This application scenario brings to light some of the expected functionality and tools required in the interoperable system:

- Automated tools to capture knowledge of consumer needs, demands, and expected objects.
- Automated tools to register producer sources and to capture knowledge of producers over their source data including the information objects they supply.
- Querying over multiple data sources in terms of a single query language rather than having the consumer learn all of the different data access languages for each source.
- Representing the result of a consumer query in a format understandable by the consumer.

1.2.2 Types of Heterogeneity

Some of the example information sources have similar data contents, but their representations are different. This is generally true for most of the information sources that are created and managed by separate organizations. Moreover, different heterogeneous factors at information source models may have different effects on multidatabase query processing strategies. Here is a list of the types of heterogeneity problems:

- **Heterogeneity in data model types.**
  Examples are relational models (Oracle, Sybase, Informix), network/IMS models, object-oriented models (ObjectStore, GemStone) and simple record-based data models (e.g., a file system). One main cause of this type of heterogeneity is the fact that one real world entity often has different representations in different data models. The heterogeneity in data models may further incur heterogeneity in query languages, query formulation, query optimization methods, and query result representations.
- **Heterogeneity in domain requirements.**
  The export schema in DR2 includes certain constraint information (e.g., only employees of Autodesk can apply to certain posted jobs in DR2), while this constraint may not be relevant in other information source such as DR3 and DR4.
- **Heterogeneity in schema designs.**
  This type of heterogeneity includes
  - *structural heterogeneity*: The number of object types or relations used by different database schemas to model the same domain of applications may vary. For instance, the `Recruiters` relation in DR1 is only one relation while to get the same information in DR5 requires using two relations. Furthermore, the number of attributes used to model the same entity in the real-world may differ from schema to schema (e.g., the `start_wage` attribute in DR2's `Job` relation is not available in the `JobList` relation for DR4.
  - *naming heterogeneity*: The naming variations may occur at both attribute and relationship level (e.g., `title` in DR3 and `position` in DR4) and object type level (e.g., job listings are modeled using `OpenJob` in DR2, `Positions` in DR3, and `JobList` in DR4.)

- *semantic heterogeneity*: The same named attributes may have different underlying semantics. For instance, the `start_wage` and `max_wage` for DR2 may be in Canadian dollars while pay in DR3 may be in US dollars.
- *Missing, erroneous or conflicting data*: the actual data values may be missing, erroneous, or conflicting:

  Missing data
  For example, job postings in DR3 and DR4 contain less fields and therefore are missing data in comparison to job postings in DR2.

  Erroneous data
  Errors in data also leads to errors in reported results. In an autonomous distributed environment the accuracy of data on sites is not guaranteed, at best the system could rate sites for a certain degree of accuracy and this could be reported back as part of the result. For example, official government sites releasing statistical data could be rated with high data accuracy whereas data from an individual user could be given a low accuracy rating.

  Conflicting data
  Consider a search for job titles and job earnings. When considering actual data on job wages, one company may offer higher wages than another for the same job title. If the company name is not taken as a projection field, then the system must deal with the conflict and consider which wage or if both wages should be reported back in the result.

These heterogeneity factors make it difficult to effectively retrieve data from various component information sources. First, given a set of information sources, different queries may be relevant to different subsets of the available information sources. Second, the same query *"find all jobs relating to multimedia and located in North America"* must be written differently for each relevant information source because of the varying logical data structures, the heterogeneity in attribute naming, and the different semantic meanings in each information source schema design. The situation becomes even worse when the number of information sources is large (tens and hundreds versus three or four), many new sources are added, and each source may evolve dynamically. It is not only time consuming for the system users to customize queries for each component information source, but also nearly impossible to keep track of the schema changes.

1.3 Scope and Organization of Thesis

This thesis presents the design and implementation of a prototype query tool, namely Rainbow, based on the DIOM query mediation methodology [31]. The theoretical model and architecture are based on previous work from the DIOM [31] Project. This prototype illustrates the ability to automate the linking of information consumers (users searching for information) with information producers (hosts providing information). The prototype will also demonstrate querying these multiple heterogeneous sources. The Rainbow system consists of several components that gather information from unstructured data sources or from other information brokers/mediators, and represent gathered information in terms of a consumer's interface description defined using DIOM IDL/IQL [32]. The main component services that the RAINBOW prototype addresses include the producers' information source registration, metadata library management, query routing, query decomposition, parallel query plan generation, subquery transformation and execution, and results assembly. This thesis will present our initial prototyping effort by a work-through example, with the objective of demonstrating the practical feasibility of the DIOM query mediation method [34].

The design of the Rainbow prototype system satisfies the USECA properties for distributed interoperation and addresses the following issues in particular:

- The prototype design and implementation of the consumer domain model.
- The prototype design and implementation of the DIOM metadata catalog including the interface repository and implementation repository.
- The prototype design and implementation of a simple keyword based DIOM multi-source query facility.
- The prototype design and implementation of a DIOM-IQL expert query facility including the simulation of query routing and decomposition.
- The implementation of subquery translation to the designated wrappers, subquery execution, and simple forms of query result assembly.
- The interface design for both metadata functions and installed queries.

Figure 1.3 shows the major areas of research for the DIOM project. The highlighted boxes represent the areas the **Rainbow** prototype attempts to address.

The remainder of this thesis is organized as follows: Chapter 2 presents an overview of the DIOM system architecture, Chapter 3 describes the DIOM metadata model which is the basis of the Rainbow system design and implementation, Chapter 4 describes the DIOM query mediation approach which utilizes consumer and producer metadata to process distributed queries. Chapter 5 presents the Rainbow prototype design, and the individual implementation-level components, Chapter 6 will describe and illustrate the functionality of the current Netscape-based DIOM user interface implemented in the Rainbow prototype, Chapter 7 will describe the wrapper design and implementation details, Chapter 8 gives a brief overview of the state of the art research in heterogeneous multidatabase systems, Chapter 9 summarizes the contributions of the Rainbow prototype to the overall development of the DIOM system, and the issues addressed by the Rainbow prototype implementation.

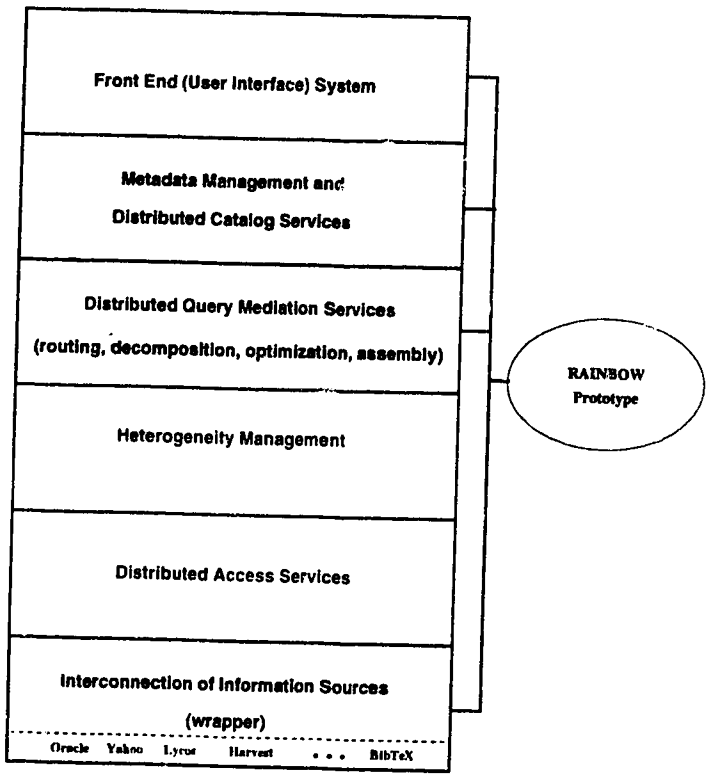

Front End (User Interface) System
Metadata Management and
Distributed Catalog Services
Distributed Query Mediation Services
(routing, decomposition, optimization, assembly)
Heterogeneity Management
Distributed Access Services
Interconnection of Information Sources
(wrapper)
Oracle
Yahoo
Lycos
Harvest
...
BibTeX
RAINBOW
Prototype

Figure 1.3: DIOM research subdivisions. Highlights show focus of this research.

Chapter 2

DIOM System Architecture

The DIOM system architecture is given in Figure 2.1 which conforms to the mediator-based framework [27]. The architecture is a two-tier architecture offering services both at the mediator level and the wrapper level. The information sources are located at the bottom of the diagram and are accessed through the wrappers. The mediator sub-system is responsible for: (1) the creation and maintenance of both information consumer and information producer metadata, (2) the initial steps in distributed query processing of consumer queries, and (3) the coordination and maintenance of metadata at the various wrappers. The main task of the wrapper sub-system is to control and facilitate external access to the information repository by using the local metadata kept in the implementation repository and by using the wrapper query processing modules.

The two tiers consist of the following services:

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

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

Runtime Supervisor: executes subqueries by communicating with wrappers

Information Source Catalog Manager: responsible for the management of repository metadata and communicates with the local implementation repository managers to cooperate in the maintenance of wrapper metadata

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

Implementation Repository Manager: manages the implementation repository metadata by coordinating with the Information Source Catalog Manager

Figure 2.2 shows an example network of information mediators collaborating through the DIOM mediator network architecture. This network includes simple wrapper-based

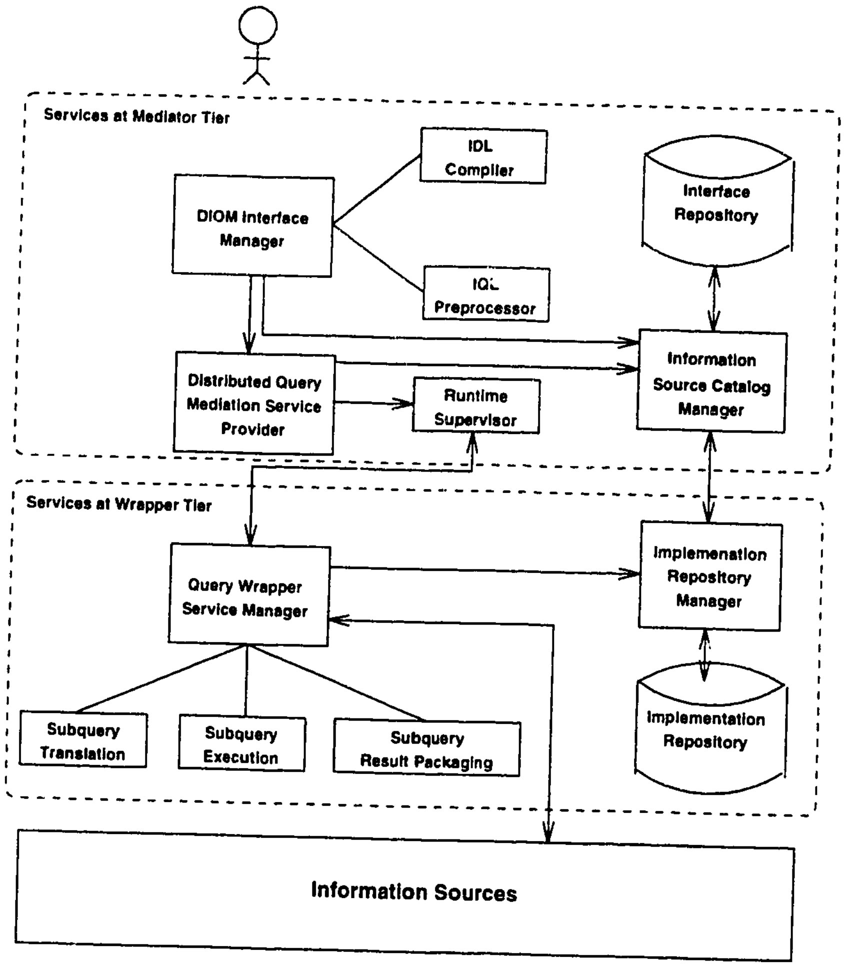

Services at Mediator Tier
IDL Compiler
DIOM Interface Manager
Interface Repository
IQL Preprocessor
Information Source Catalog Manager
Distributed Query Mediation Service Provider
Runtime Supervisor
Services at Wrapper Tier
Implemenation Repository Manager
Query Wrapper Service Manager
Implementation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

Figure 2.1: DIOM System Architecture

mediators such as the wrapper which provides information about and access to the BookStoreDB repository. This wrapper-based mediator is in turn used to construct a BookSale mediator. The BookSale mediator is again used to build both a document inquiry mediator and a travel plan mediator. This recursive construction and organization of information access results in some important features. First, the individual mediators can be independently built and maintained. Each specialized mediator represents a customized personal view for a particular group of information consumers over the large amount of information from the growing number of information sources. Second, these mediators can be generated automatically or semi-automatically by using the DIOM IDL/IQL interface specification language and the associated incremental compilation techniques. These features also make it possible to scale the DIOM architecture not only to the large and growing number of information sources but also to the varying information customization needs from diverse information consumers.

In the DIOM multidatabase framework, an information consumer (i.e., user or domain expert) may build his/her special-purpose information mediator using the DIOM-IDL (Section 3.1). A typical application-specific information mediator contains two levels of interface descriptions: (1) an information consumer's *domain usage model* (personal view), describing the usage and expertise of the domain, and (2) a number of information producers' *source data models*, each providing relevant information contents that are of interest to the information consumer. Every mediator defined in terms of DIOM-IDL is specialized to a specific application domain and provides access to the available information sources that are relevant to that domain. A complex information mediator can be built based on a network of simpler mediators which interact with each other in order to accomplish tasks. For example, a `Document Inquiry` mediator (as shown in Figure 2.2) can be built by interacting among several smaller mediators such as `Book` mediator, `TechReport` mediator, `Medical ClaimFolder` mediator [32], and so on.

To build a network of specialized and interacting mediators, we need an architecture for a single mediator (perhaps called a meta mediator) that can be instantiated to provide multiple mediators. The ultimate goal of developing such a meta mediator architecture is to provide a uniform interface description language and the associated query services that can be utilized and customized by a number of application-specific mediators to facilitate access to multiple and heterogeneous information sources. Such a development not only enables rapid prototyping and maintenance of individual mediators but also provides a robust approach to managing cooperation among the network of DIOM mediators and preservation of the USECA properties.

The mediators, meta-mediators, wrappers, and meta-wrappers can be seen as the middle layer between the clients (information consumers) and information repositories (information producers). Clients make requests to mediators for information, mediators communicate with other mediators and wrappers to gather and process data, and the wrappers allow repositories to participate in the system. The meta-mediator and meta-wrapper modules allow for the instantiation of new mediators and wrappers respectively. The following is a description of the architectural components:

Mediators

Mediators simplify, abstract, reduce, merge, and explain data from the producer repositories [53]. The DIOM application mediator can be viewed as a logical grouping of consumer defined interfaces that represents an information consumer's view of data with respect to

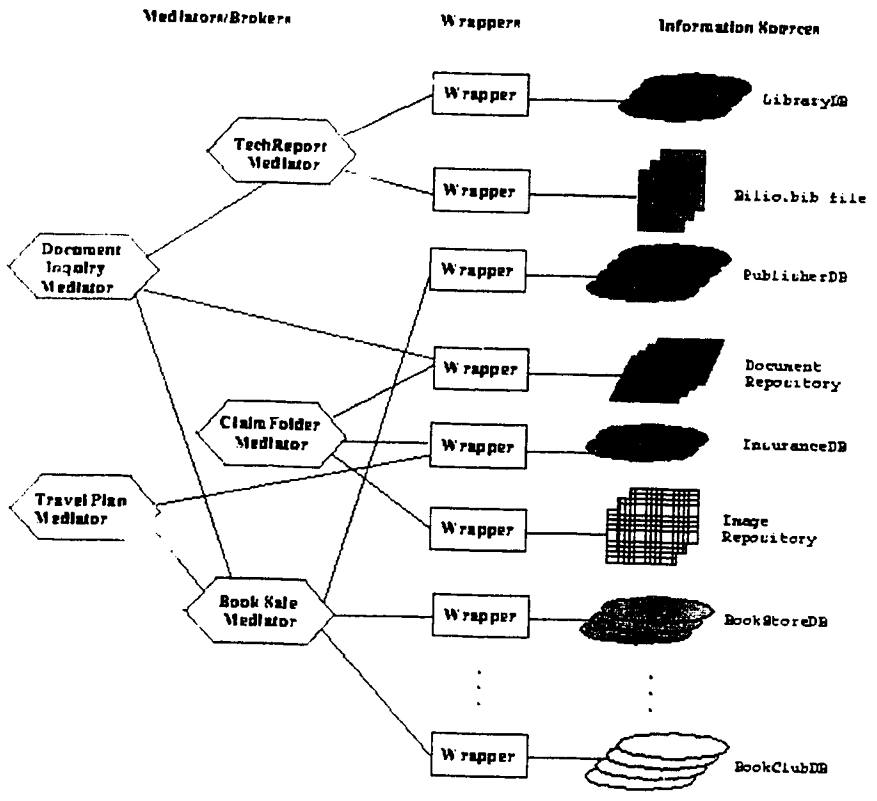


Figure 2.2: Network of mediators

a particular domain. The set of consumer defined interfaces are used to exploit encoded knowledge about producer repositories to create information for a higher layer of client application or other mediators. Producer's self-representation is gathered through source registration and is described in Section 6.2.1. Consumer's domain knowledge is specified or recorded in the form of interface definitions described in Section 3.1.

The high degree of modularization in this architecture allows for a corresponding degree of reuse and scalability. Each mediator can be composed from base repositories (via wrapper) and from existing mediators. An example of this is shown by the *Document Inquiry Mediator* in Figure 2.2. It is composed of the *TechReport Mediator*, *Book Sale Mediator*, and *Document Repository*.

Wrappers

In order to make an existing information source available to the network of mediators, building a wrapper around the existing local system is necessary to turn a repository into a DIOM local agent. The wrapper is responsible for accessing that information source and obtaining requisite data for answering the query.

The main task of an information source wrapper is to provide services that may facilitate the communication and the information exchange between a mediator and its component systems. This includes translating a source-level subquery from DIOM Interface Query Language (DIOM-IQL) expression into an information producer's query language expres[illegible] submitting the translated query to the target information source, and collecting the subquery result. Note that only one such wrapper would need to be built for any given type of information source. For example, a wrapper implementation for Oracle could be used for all Oracle DBMSs.

Using wrappers to incorporate different information repositories allows for uniform ac[illegible]ess to [illegible]ach repository and modularizes the heterogeneity problems among information sources into manageable implementation e[illegible]ts. Any source can become a participant as [illegible]ong [illegible]s a wrapper to that source type is p[illegible]ded. The autonomy of the local repository [illegible]ntained as the wrapper is responsible [illegible] translating DIOM consumer requests into [illegible] requests which are equivalent to requests made by external users of the local system. Using wrappers to modularize the interconnection bridges the gap between the information consumers and the producer sources. This leads to a higher degree of extensibility and scalability.

Repositories

Repositories can be well structured (e.g., RDBMS, OODBMS), semi-structured (e.g., HTML files, text based records), or unstructured (e.g., technical reports). Every information source is treated as an autonomous unit. Component information sources may make changes without requiring consent from mediators. Wrappers access repositories as if they were external users (i.e., no direct access to the repository). Building wrappers for each type of repository requires mapping generic wrapper functions into equivalent local functions.

Automatic generation of mediators and wrappers requires meta-mediators and meta wrappers. A meta-mediator would require knowledge of the producer repositories available and the consumer interface descriptions to be able to automatically compile mediators which map the producer repositories to the consumer created interfaces. The creation of a new wrapper can be partially automated by using a meta-wrapper to instantiate a skeleton wrapper. Specific parts of this wrapper may have to be coded by the repository-type expert. For example, utilizing web search tools as data repositories requires forming the proper URL [1] [6] to that web tool. The wrapper implementors to these web sites would have to write code to form the proper URLs from the query. New wrappers can be built by reusing components from similar wrappers (subtyping).

[1] URL stands for Uniform Resource Locator, otherwise known as a web address.

Chapter 3

The DIOM Metadata Facility

Each mediator consists of a consumer's domain model and many information producer's source models. The consumer's domain model specifies the mediator's domain of expertise (e.g., the Book-Sale consumer's domain model area of expertise is in the sale of books). The producer's domain models are managed by other mediators or data repository managers and are chosen for their relevancy to the mediator's expertise. The consumer's domain model and information producer's source models constitute the general knowledge of a mediator and are used to determine how a consumer's information request is processed.

3.1 Consumer's Domain Usage Model

The consumer's domain usage model is targeted at the specific application domain that a group of information consumers are interested in. It provides descriptions of the classes of objects (interfaces) and how they are related to each other. It also describes the relationships between the consumer's usage model and the relevant information producers' source models. The domain model of a mediator not only defines its area of expertise but also the terminology that other mediators can use to interact with this mediator. Figure 3.1 shows a fragment of a domain usage model for a Job Search Assistant application domain.

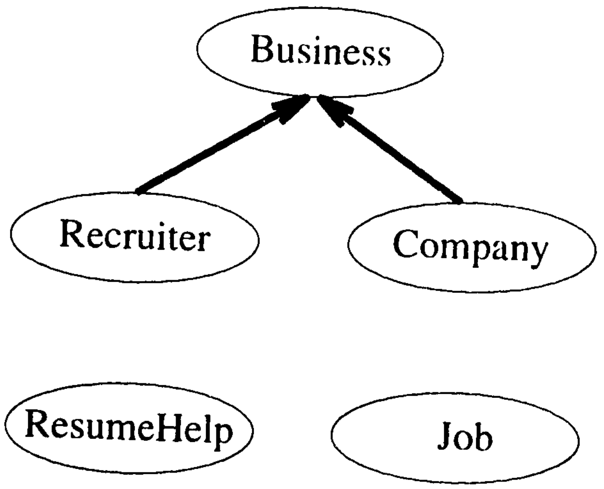


Figure 3.1: Fragment of example consumer's domain model

In this figure, the nodes represent classes of objects defined in terms of IDL interfaces, the thick arrows represent specialization/generalization relationships. Note that (1) the domain model of a mediator is intended to be a description of the application domain and domain expertise from the information consumer's point of view and (2) the classes defined in the consumer usage model do not necessarily correspond directly to the objects described in any particular information producer's source model.

3.2 Producer's Source Models

The information producers' source models describe the resources that are available to the DIOM system when answering information consumer's requests. A producer source model contains only the portions of the producer source that the source chooses to export to DIOM, rather than a complete description of the entire information source. The typical information that an producer source model consists of includes: (i) a description of the relevant classes exported from an information producer's source and (ii) the relationships between these classes. The correspondence between the domain usage model of a mediator and its corresponding information source models is needed for transforming a consumer's query into a set of producer's queries, each over a single information source.

3.3 DIOM-IDL

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

Interface Composition Mechanism: Aggregation

The *aggregation* abstraction is a mechanism that allows for the composition of new interfaces from existing interfaces such that objects of the container interface may access the objects of component interfaces directly. As a result, the operations defined in the component interfaces can be invoked via the container's interface. The aggregation mechanism is a useful facility for implementing behavioral composition [29, 30] and ad-hoc polymorphism [11] based on coercion of operations. Another benefit of using interface aggregation abstraction is to minimize the impact of component schema changes over the application programs working with the existing interoperation interfaces.

Interface Composition Mechanism: Generalization

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

Interface Composition Mechanism: Specialization

The *specialization* abstraction is a useful mechanism for building a new interface in terms of some existing interfaces through type refinement. This mechanism promotes information localization such that changes in an object type or its implementations can automatically be propagated to the subtypes that are specialized versions of it. In the first phase of DIOM implementation, the specialization abstraction is only supported for construction of a new interface based on one or more base interfaces whose scope is the same data repository (see [31] for more details).

Interface Composition Mechanism: Import

The *import mechanism* is designed for importing selected portions of the data from a given export schema [1], instead of importing everything that is available. For a data repository that manages complex objects, the import mechanism performs the automatic checking of type closure property and referential integrity of the imported types/classes. The type closure property refers to the type consistency constraint over subtype/supertype hierarchy such that whenever a type/class is imported, all the properties and operations it inherits from its supertypes have to be imported together. The referential integrity property refers to the type "completeness" rule on object reference relationships, and is used to guarantee that there is no dangling reference within the imported schema.

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

Using the repositories from the running example in Section 1.2.1, an interface creator who is an expert in the Job Search domain may construct a Job Search Assistant consumer domain model which consists of the four interfaces:

```
Job=         [title, pay, descrip, company, post_date]

Company=     [name, address, city, prov/state, country,
              mail_code, descrip, phone, fax, URL]

Recruiter=   [name, address, city, prov/state, country,
              mail_code, descrip, phone, fax, URL]

ResumeHelp=  [author, format, title, pages, publisher,
              address, year]
```

[1] An export schema is part of a conceptual schema (logical schema) of a database, which is accessible to external users

Here are the full IDL definitions for our four interfaces:

```
CREATE INTERFACE Jobs
{   GENERALIZATION OF
        select interface-name from Interface-Repository
        where description CONTAINS ['Job' || 'Position'];
    ATTRIBUTES
        String      title,
        Double      pay,
        VarString   descrip,
        String      location,
        Company     company,
        String      post_date  }

CREATE INTERFACE Business
{  GENERALIZATION OF
        select interface-name from Interface-Repository
        where  description CONTAINS
               [('Business') && ('Address' || 'Description']);
   ATTRIBUTES
        String      name,
        String      address,
        String      city,
        String      prov/state,
        String      country,
        String      mail_code,
        VarString   descrip,
        string      phone,
        string      fax,
        string      email,
        string      URL          }

CREATE INTERFACE Company
{  GENERALIZATION OF
        select interface-name from Interface-Repository
        where  description CONTAINS
               [('Business' || 'Company') &&
                ('Address' || 'Description']);
   ATTRIBUTES
        Business   BusinessPtr }

CREATE INTERFACE Recruiter
{  GENERALIZATION OF
        select interface-name from Interface-Repository
        where  description CONTAINS
               ['Placement' || 'Recruiter'];
   ATTRIBUTES
        Business    BusinessPtr }

CREATE INTERFACE ResumeHelp
{  GENERALIZATION OF
        select interface-name from Interface-Repository
        where  description CONTAINS
               ['ResumeHelp' || 'CVBooks' ||
                'ResumeArticles'];
   ATTRIBUTES
        String author,
        String format,
        String title,
        String pages,
        String publisher,
        String address
        String year       }
```

3.4 Generation of Consumer Objects

Figure 3.2 presents how to generate the Job and Company object models described in IDI from the source information models in our example application. Figure 3.3 illustrates how

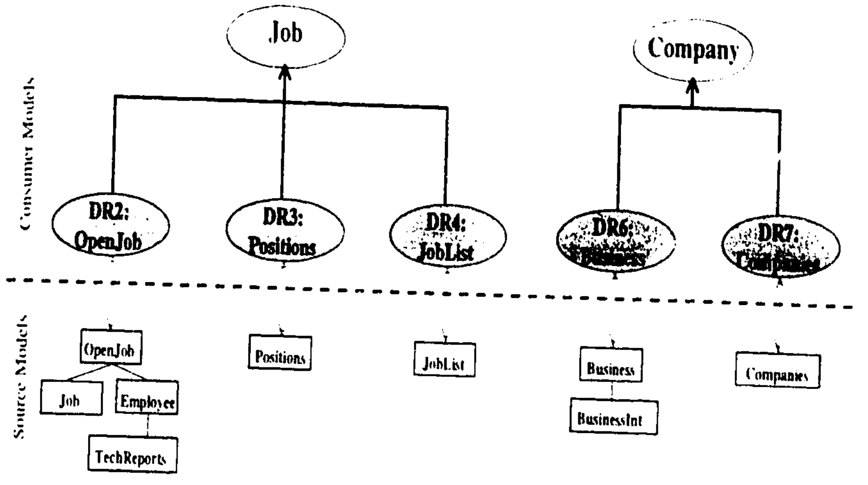


Figure 3.2: Automatic generation of the object access interfaces for the information sources

the relevant portions of an information source model are related to a customer's domain model. A mapping link is maintained between a source model object and a domain model object. It is important to note that:

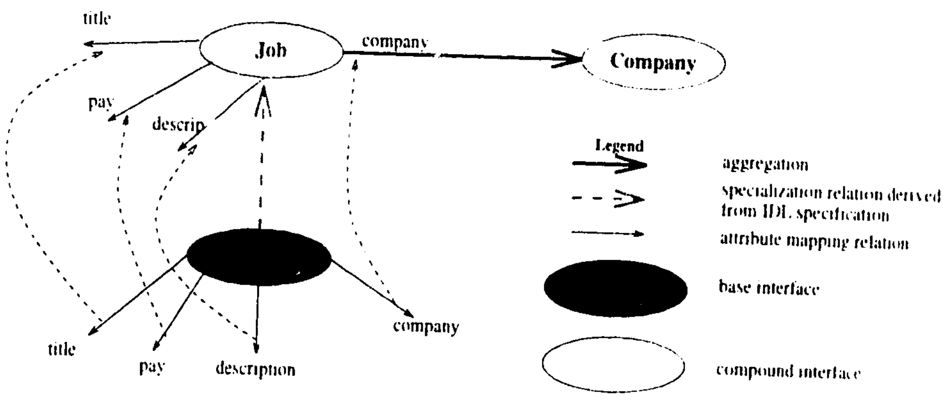


Figure 3.3: Fragment of an information source model

- both the consumer's domain usage model and the information producer's source models are expressed in the DIOM interface description language [31].
- The idea of building specialized mediators for specific application areas provides not only a modular organization of the growing number of information sources on the Internet but also a clear characterization of the types of queries each mediator can handle.

- The interface construction mechanisms provided in DIOM-IDL also enable the complex domain to be broken down into meaningful sub-domains and an information mediator to be built for each sub-domain.

- Once a consumer's domain model is defined, an application-specific mediator is created. It invokes the DIOM metadata catalog management services to build its mediator-specific metadata catalog and to establish the correspondence between its domain model terminology and the corresponding source model terminology. It makes use of the DIOM distributed object access services to process the consumer's information request and obtain the requisite data for answering the query.

3.5 Metadata Catalog Management

In the DIOM system, metadata extracted from information consumer's interface schemas and information producers' export schemas are managed and maintained in the DIOM metadata catalog. In the following sections we will discuss the three most important components of the metadata catalog: the information source repository, the interface repository, and the implementation repository.

3.5.1 Information Source Repository

The *information source repository* is created and maintained by the DIOM information source registration manager. The source registration manager is responsible for recording all the available source producers' information, such as (1) the information producer's name, (2) the location pointer (e.g., the URL in WWW), (3) the information source description (keywords), (4) the export schema name, (5) the source type, etc. A fragment of the DIOM information source repository is shown in Figure 3.4. Every application-specific mediator, once created, is also registered as an information source that is available to the customer set of the DIOM system.

SourceName	PointerToProducerSite	SourceType	Description	ExportSchema	Owner
Harvest	http://harvest.com/	Broker	Security,PCSofware,...	HarvestHTML	Harvest Inc.
Yahoo	http://yahoo.com/	Broker	...	YahooHTML	Yahoo! Corp.
DR1	http://dr1.com/	HTML	Recruiters	RecruitersHTML	JobsEx Corp.
DR2	dr2.com/	RDB	Jobs,OpenJobs,...	DR2.JobsDB	Autodesk
DR3	dr3.com/	text file	Positions	DR3.JobList	Bill Smith
UsenetJobs	dr4.com/	AppMediator	JobList	Diom_UsenetJobs	N/A
DR5	dr5.org/	RDB	PlacementAgency,...	DR5.RecruitDB	HireAStdent Inc.
DR6	dr6.com/	RDB	Business,BusinessInt...	DR6.BusinessDB	ACME Finance
DR7	http://dr7.org/	HTML	Company	CompanyHTML	VirtWeb Inc.
DR8	dr8.org/	BibTeX	ResumeHelp	DR8.ResumeHelpBib	Susan Jones
JobSearch	DIOM	Mediator	Job,Company,...	(Default)	Y. Lee
...	...	...	...	...	...

Figure 3.4: Fragment of DIOM information source repository

3.5.2 Interface Repository

The *Interface repository* is another basic component of the DIOM metadata catalog. Whenever a new information consumer's domain model is defined in DIOM-IDL, the interface repository will be extended. The first scan of the IDL preprocessor over a given IDL schema handles all the interfaces defined by means of the **import** and **hide** mechanisms. The base interface generator is invoked for each IDL interface defined using **import**.

Figure 3.5 shows a fragment of the interface repository. For each consumer-defined (compound) interface or system-generated base interface, a pointer to the list of attributes, relationships, or methods associated with this interface definition is also maintained in the interface repository. The description field contains synonyms to be used in object linking. The synonyms for each keyword are matched with keywords from the interface descriptions.

interface-name	interface-type	description	source-producer
Harvest	base	Security,PCSofware,CSTechReport,...	Harvest
Yahoo	base		Yahoo
DR1.Recruiters	base	Placement Agency	DR1
DR2.Job	base	CompanyJobs	DR2
DR2.OpenJobs	base	OpenJobs,Jobs,Positions	DR2
DR2.Employees	base	Employess, Workers..	DR2
...	...	...	...
DR3.Positions	base	Positions, Jobs	DR3
...	...	...	...
JobSearch.Job	generalization	Job,Jobs,Positions,OpenJobs	Diom
JobSearch.Company	generalization	Business, Company	Diom
JobSearch.Recruiter	generalization	Recruiter, Recruiters, PlacementAgency	Diom
JobSearch.ResumeHelp	generalization	ResumeHelp, CVBooks,	Diom
...	...	...	...

Figure 3.5: Fragment of DIOM interface repository

3.5.3 Implementation Repository

The DIOM servers (mediator level and wrapper level) provide a number of general services to facilitate global query planning, query reformulation, and parallel multi-information source query execution. One of the most important services is the creation and maintenance of implementation repository metadata, which are used heavily by the DIOM query services provided by the mediators and their associated wrappers. The main issues to be addressed include: (1) how to establish a semantic correspondence between the types and relationships defined in the consumer's domain model with the classes and relations used in the producers' source models. (2) how to maintain the implementation metadata catalog in the presence of changes. The second issue will be addressed separately in Section 3.5.5.

The internal representation of relating an information source to a consumer's domain model for the **Job Search Assistant** application is illustrated in Figure 3.6, Figure 3.7, and Figure 3.8. Each of these figures represents the attribute correspondence between mediated attributes defined in the consumer's domain model and source attributes defined in a particular information source model. The first column in each table contains the names of the mediated attributes. The second column contains the corresponding information source level attributes, prefixed by the class or relation they belong to. For example, the attributes from database DR2, DR3. and DR4 are described in the second column of the table in Figure 3.6, Figure 3.7. and Figure 3.8 respectively. A null value in a source attribute column denotes the non-existence of the attribute in the corresponding information source.

From the implementation point of view, the mediated attributes and their types are used as instruments to establish connections among similar attributes and their classes or relations defined in the relevant component information sources. It is also possible

Mediated→attributes	DR2
Job→title	OpenJob.jobtitle
Job→pay	Job.start_wage
Job→descrip	Job.description
Job→company	<DIOM.Catalog.owner>
Job→post_date	OpenJob.opendate

Figure 3.6: Fragment of implementation repository maintained at the wrapper to DR2

Mediated→attributes	DR3
Job→title	Positions.title
Job→pay	Positions.pay
Job→descrip	Positions.description
Job→company	Positions.company
Job→post_date	

Figure 3.7: Fragment of implementation repository maintained at the wrapper to DR3

in special cases for some mediated attributes to be connected to data from the catalog (e.g., Job→company in DR2 can be mapped to the Owner attribute in the DIOM interface repository). These semantic links are central to the automation of global query planing, query refinement, query decomposition, and query result assembly.

Note that this type of metadata is mainly obtained by applying machine learning techniques to the metadata catalog. For example, the knowledge about connecting term JobList, OpenJobs, Positions to the term Job can be learned directly from the interface definition of Job. In order to build comprehensive domain-specific terminology matching rules and ontology, a sophisticated learning and knowledge discovery tool is needed [39, 43, 28]. Appropriate interaction with the information consumer or domain expert may also be required.

3.5.4 Source-specific Metadata

In order to automate query translation and query result assembly, additional metadata are required. For example, for each attribute defined at an individual information source, the following information is needed: (1) the scale of the attributes, (2) the key constraint of the attributes (key, non-key, foreign key), and (3) the logical connection paths.

The connection path assignment scheme varies for different types of information sources. For example,

- the connection path assignment scheme for the information sources managed by relational systems is the following. A non-key attribute points to its key attribute(s). A

Mediated→attributes	DR4
Job→title	JobList.position
Job→pay	
Job→descrip	JobList.description
Job→company	JobList.organization
Job→post_date	

Figure 3.8: Fragment of implementation repository maintained at the wrapper to DR4

key attribute points to its foreign keys, and each foreign key points to its home key. If a relation has no foreign key, the pointer field is null. This simple path assignment schema ensures that all the necessary joins can be constructed when translating a consumer-level query into a group of subqueries at the information producer's source level, each against a single information source.

- The connection path assignment scheme for object-based information sources is even simpler. Each *oid* field corresponds to the key field of a given object class. The rest are non-key fields. The navigational path for the same attribute type may vary from query to query, depending on the specific structural pointers (object references) specified in each query.

Source-specific metadata is mostly related to a single data source. It can be obtained directly from the export schema of the given information source, and incrementally maintained by the corresponding DIOM wrapper. More importantly, the translation of subqueries into the component query expressions should be carried out at the wrapper layer to prevent a query processing bottleneck at the DIOM distributed object server layer. Figure 3.9 shows the internal representation of the metadata catalog maintained by the corresponding DIOM wrapper, the local DIOM agent to the DR2 information source.

DR2.attribute	DR2.domain	DR2.scale	DR2.key	DR2.keyFD
Job.jobtitle	none	none	key	
Job.description	none	none	nonkey	Job.jobtitle
Job.start_wage	none	US$	nonkey	Job.jobtitle
Job.max_wage	none	Canadian$	nonkey	Job.jobtitle
Job.internal	(Yes, No)	none	nonkey	Job.jobtitle
OpenJob.id	none	none	key	
OpenJob.jobtitle	none	none	fkey	Job.jobtitle
OpenJob.opendate	none	none	nonkey	OpenJob.id
OpenJob.deadline_date	none	none	nonkey	OpenJob.id
OpenJob.department	(Multimedia, Networks)	none	nonkey	OpenJob.id
...	...	...	...	...

Figure 3.9: Fragment of source-specific metadata extracted from the export schema of DR3

3.5.5 Maintaining Metadata Catalog in the Presence of Changes

When a new information source is registered, to guarantee this new information source will be used to respond to a query, updates to the existing metadata catalog are needed. Two strategies can be used for maintaining the metadata catalog up to date: (1) Immediate update or (2) Deferred update.

With immediate update approach, the maintenance proceeds in three steps: First, the DIOM server triggers the metadata catalog manager to perform an incremental recompilation on all the existing domain interface schemas to which this new information source may be related. Second, for each relevant domain interface schema, the implementation repository manager will be invoked to relate object types used in this new information source to the object types defined in the given domain model. Third, the source-specific metadata catalog manager associated with the wrapper to this new information source is invoked. The source-specific implementation metadata is extracted from the export schema of this new information source.

With the deferred update approach, the incremental recompilation of existing domain interface schemas is delayed to the time when a query is issued. By using the deferred approach, instead of propagating changes to all the domain interface schemas only those domain interface schemas that are frequently used by queries are updated.

When a change is made to the information consumer's domain model or the information producer's source model, a modification to the corresponding metadata is performed accordingly and transparently.

For example, in information source DR2 modifying the domain of attribute `department` by adding `Administration` in the relation `OpenJob` requires a change of the source-specific metadata (i.e., the cell in the `DR2.domain` column and the `OpenJobs.department` row of Figure 3.9 will be updated from `((Multimedia, Networks)` to `(Multimedia, Networks, Administration))`.

Chapter 4

The DIOM Query Mediation Facility

Queries to multiple information sources are expressed in the interface query language (IQL). IQL queries use the naming conventions and terminology defined in the information consumer's domain model. There is no need for the query writers to be aware of the many different naming conventions and terminology used in the underlying information sources. Given a query expressed in terms of the consumer's domain model, an information mediator identifies the appropriate information sources, decomposes the query into a collection of subqueries (each expressed in terms of an information producer's source model) and then forwards these reformulated queries to the corresponding sources for subquery translation and execution. Note that (1) distributing subquery translations from the object server layer to the wrapper layer helps to prevent query processing bottlenecks in the object server and (2) the approach of building wrappers to coordinate between a mediator and its underlying sources greatly simplifies the implementation of individual mediators, since each mediator only needs to handle one underlying language. It also makes interoperation among networks of mediators easily scalable to the ever growing number of information sources.

4.1 Informal Overview of DIOM-IQL

The DIOM interface query language (IQL) is designed as an object-oriented extension of SQL. The basic construct in DIOM-IQL is an SQL-like `SELECT-FROM-WHERE` expression where certain fields are optional (i.e., fields surrounded by brackets '[' and ']'). The syntax is:

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

The *MEDIATOR* field refers to the associated mediator of the referenced interfaces. The *TARGET* field specifies the target repositories. The *SELECT* field specifies the interface attributes to display. The *FROM* field specifies which interface definitions are to be utilized. The *WHERE* field specifies the condition applied to the interface attributes. The *GROUP*

BY and *ORDER BY* are display options which arrange the results by groups and/or in a certain order.

The syntax of the SELECT-FROM-WHERE expression is as follows:

```
<Fetch-expressions>::= <Fetch-expr> | <Fetch-expr>, <Fetch-expressions>
<Fetch-target-list>::= <Fetch-target> | <Fetch-target>, <Fetch-target-list>
<Fetch-target>      ::= <interface type> | <target-Info-Source name> |
<Fetch-conditions> ::= <Fetch-condition> | <Fetch-condition> <Logic-ops>
                       <Fetch-conditions>
<Fetch-condition>  ::= <Fetch-expr><comparison-op><Fetch-expr>
                       | <Fetch-expr><Comparison-op><constant-object>
                       | <Fetch-expr><Comparison-op><Fetch-cond-list>
<Fetch-expr>       ::= <Path-expr>-><attribute>
<Fetch-cond-list> ::= <consumer-defined attribute><logic-ops>
                      <Fetch-cond-list>
<attribute>        ::= <consumer-defined attribute> | <source-attribute>
<Path-expr>        ::= <interface type> | <interface type>-><Path> | *
<Path>             ::= <consumer-defined attribute>-><Path> | ?
<Comparison-op>    ::= < | > | <= | >= | <> | =
<Logic-ops>        ::= AND | OR | && | ||
```

The IQL design follows a number of assumptions: First, IQL by itself is not computationally complete. However, queries can invoke methods. Second, IQL is truly declarative. It does not require users to specify navigational access paths or join conditions. Third, IQL also deals with the representational heterogeneity problem. For example, different information sources may have different structural and naming conventions for the same real world entity. These information producers' representations may not fit to the information consumer's preference. Using IQL, the information consumer only needs to specify the query in terms of his/her own IDL specification. The system is responsible for mapping a consumer's query expression into a set of information producers' query expressions. Furthermore, IQL allows the use of object type names of information sources in the FROM clause by prefixing the source name with dot notation. It is also possible to use information source names directly in the FROM clause of a query. The result of an IQL query is by itself an object of the special interface type QueryResult. The main objective of these IQL extensions is to increase the accessibility of the MDBS and to improve the quality of global information sharing and exchange.

4.1.1 Example Query

For example, the query Q_1: *"find all jobs relating to multimedia and located in North America"* can be expressed in IQL as shown in Figure 4.1. For the query in Figure 4.1 we use a '*' in the SELECT field to represent all fields from the two interfaces (Job, Company) (Figure 4.2).

```
SELECT *
FROM   Job, Company
WHERE  Job->descrip contains 'multimedia'
       AND (Company->country contains 'Canada' || 'USA')
```

Figure 4.1: An example query

In representing this query, the query writer does not need to be concerned with the different naming conventions and different structural designs from JobList to OpenJobs to Positions. Instead, she can write a query using the terminology that is preferable in her

```
Job->title, Job->pay, Job->descrip,
Company->name, Company->address, Company->city,
Company->prov/state, Company->country, Company->mail_code,
Company->descrip, Company->URL
```

Figure 4.2: Fields represented by '*' in example query

own application domain. Second, the query writer does not specify any "join" conditions in her query expression. The IQL preprocessor will automatically insert the necessary connection paths to relate different object types involved in the query with each other, since such connection semantics should be easily derived from the metadata maintained in the DIOM interface repository and implementation repository.

Another feature of the DIOM query model is the automatic creation of result type for each consumer query. It means that, for every consumer query expressed in DIOM-IQL, a compound interface will be generated as the result type of the query, and maintained in the DIOM interface repository. This functionality is particularly important for preserving the closure of the DIOM object model and for assembly of multidatabase query results in terms of a consumer's preferred representation. For example, the query given in Figure 4.1 generates the following compound interface description:

```
CREATE INTERFACE Result_Q1
{
   AGGREGATION OF Job, Company;
   ATTRIBUTES
     Job->title,
     Job->pay,
     Job->descrip,
     Company->name,
     Company->address,
     Company->city,
     Company->prov/state,
     Company->country,
     Company->mail_code,
     Company->descrip,
     Company->URL
}
```

This query result can also be further incorporated in the example domain model of the Job Search Assistant mediator. Using the aggregation function, a diagram of the new domain model is shown in Figure 4.3.

4.2 Query Processing

4.2.1 The Framework

The main task of the DIOM distributed query manager is to coordinate the communication and the distribution of the processing of information consumers' query requests among the mediator and its associated wrappers.

The general procedure for multidatabase query processing is outlined in Figure 4.4. It consists of the following five steps:

1. **Query Routing**
   This is done by mapping the domain model terminology to the source model terminology, by eliminating null queries, which return empty results, and by transforming

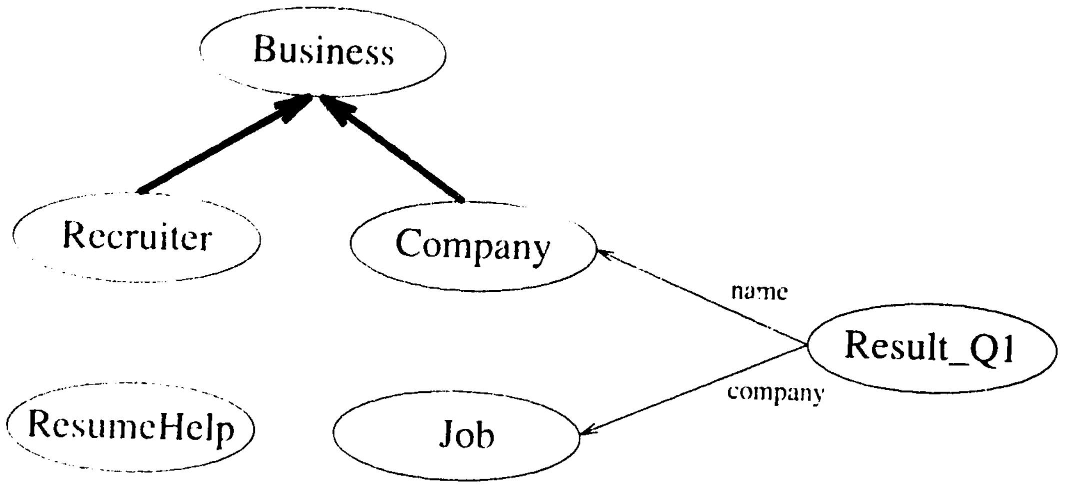


Figure 4.3: Consumer's domain model with query result interface

ambiguous queries into semantic-clean queries. In general, the choice is made such that the number of different information sources used to answer a query is minimized. The number of sources chosen depends on the fullness of the query result desired by the user versus the processing time overhead required when more repositories are selected.

2. **Query decomposition**
   This is done by decomposing a query expressed in terms of the domain model into a collection of queries, each expressed in terms of a single source model. Both straightforward query decomposition and complex query decomposition are considered.

3. **Parallel query plan generation**
   The goal of generating a parallel access plan for the groups of subqueries is to obtain the maximum parallelism and the best quality of cooperation in searching for query answers from multiple information sources. A parallel query plan is constructed for the modified query resulted from query decomposition. The plan is composed of single-source queries (each posed against exactly one local export schema), move operations that ship results of the single-source queries between sites, and the post processing queries that assemble the results of the single-source queries in terms of the information consumer's query request expressions.

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

5. **Query result assembly**
   The result assembly process involves resolving the semantic variations among the subquery results. Annotation/semantic attachment is one of the main techniques that we use for resolving the semantics heterogeneity implied in the query results.

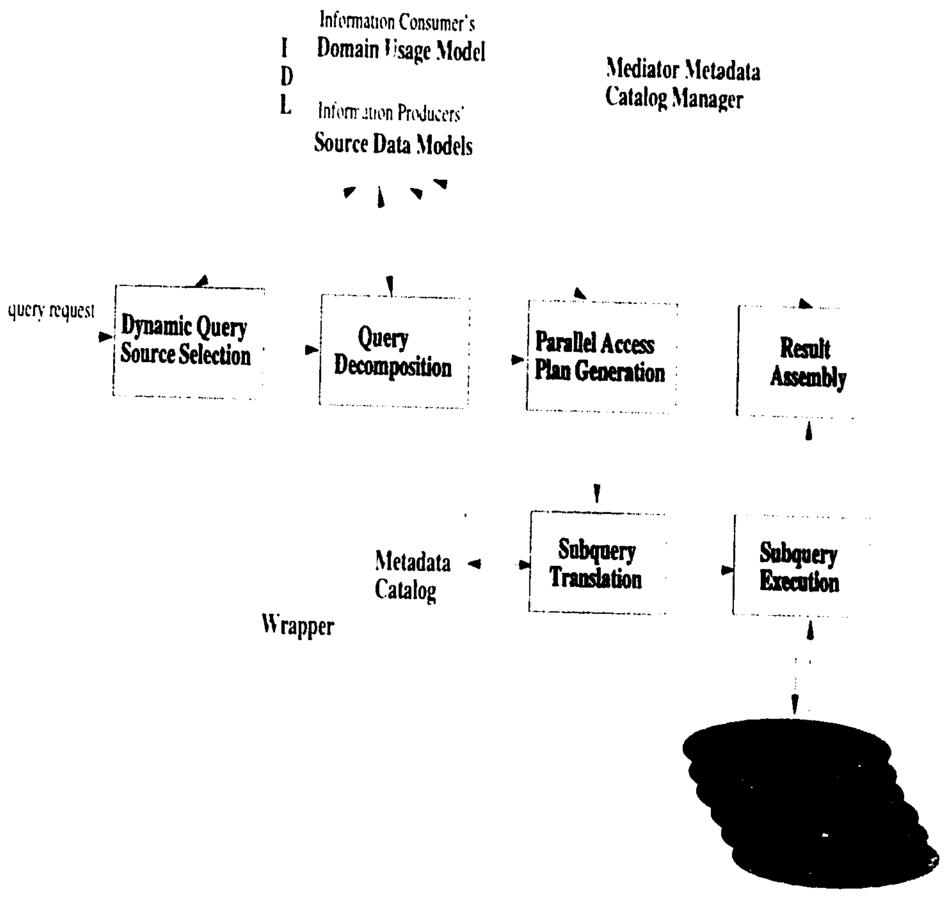


Figure 4.4: Query steps in a distributed environment with wrappers

After each query is submitted, it is sent to the individual wrappers which do the translation, local query execution, addition of attachments, and returns a result back to the query manager for assembly.

4.2.2 Query Routing

The first step in answering an IQL query is to select the appropriate information sources. The ultimate goal for dynamic query source selection is to eliminate the information sources which are irrelevant for answering the query. This section will not attempt to discuss more complex query situations (such as overlapped data) instead we focus on the general source selection algorithm. To identify the set of candidate information sources that may be needed to answer a query, the mediator applies a set of catalog mapping operators to map each compound interface type defined in the consumer's domain model into the base interface types at the information source level. We summarize these catalog mapping operators in Figure 4.5.

The application of operators **A-map**, **G-map**, or **S-map** is recursive. The recursion ends when all the compound interface types defined in terms of interface general-

Operator	Operand	ReturnResult	Purpose
Direct-map (D-map)	base interface	source name	Return the information source that corresponds to the base interface type.
Aggregation-map (A-map)	compound interface	a set of component interface types	Replace a compound interface defined in terms of aggregation with an appropriate set of component interface types
Generalization-map (G-map)	compound interface	a set of specialized interface types	Replace a compound interface defined in terms of generalization with an appropriate set of specialized interface types
Specialization-map (S-map)	compound interface	a generalized interface type	Replace a compound interface defined in terms of specialization with a generalized interface type

Figure 4.5. Operators for candidate information source selection

ization/specialization and interface aggregation are replaced by the corresponding base interface types. Then the operator **D-map** is used to find the corresponding information sources.

Consider the query Q_1 given in Figure 4.1. By applying the operators **G-map** and **D-map** to the compound interface type Book, the set of candidate information sources selected contains DR2, DR3, DR4, whereas by applying **G-map** and **D-map** to the compound interface type Supplier, the information sources selected are DR6, DR7. Thus, the solution to our query is found from data on the repository set: DR2, DR3, DR4, DR6, DR7. The purpose of this step is to avoid retrieving extraneous data and to remove the potential ambiguity implied in a given query. This is done by inserting necessary connection paths into the query selection condition. As a result, additional information sources may need to be added into the set of candidate information sources.

Consider the query Q_1 given in Figure 4.1. There is only one connection path (join condition) between Job and Company: "Job->company=Company->name". Therefore, the query in Figure 4.1 is reformulated by inserting this connection path to the query selection condition in Figure 4.6.

```
TARGET DR2, DR3, DR4, DR6, DR7
SELECT *
FROM   Job, Company
WHERE  Job->descrip contains 'multimedia'
       AND (Company->country contains 'Canada' || 'USA')
       AND Job->company=Company->name
```

Figure 4.6: Reformulated example query with targets and connection paths added

However, when multiple connection paths exist between two or more IDL interface descriptions, automatic insertion of connection paths may become rather difficult, because the system has to make the decision as to which connection path should be selected on the fly. In such cases, a query written in IQL is considered *ambiguous*. In the case of ambiguous queries, the DIOM system will present all the possible connections to the user and make a final choice based on the feedback from the user.

4.2.3 Query Decomposition

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

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

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

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

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

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

   The process of target split results in a set of subqueries and a modified query expression that is semantically equivalent to the original query. The operations used to combined the results from the subqueries are also presented.

2. **Useful subquery dependency recording.** This is done by recording all the meaningful subquery dependencies with respect to the efficiency of the global query processing.

   For each consumers' query, if the number of subqueries resulting from the query decomposition phase is k, then the number of possible synchronization alternatives is $C_k^1 + C_k^2 + \ldots + C_k^k$ ($k \geq 1$). Not surprisingly, many of the possible synchronization alternatives will never be selected as the parallel access plan for the given query. For example, assume that, for any two subqueries, the independent evaluation of a subquery $subQ_1$ is more expensive than the independent evaluation of subquery $subQ_2$. Thus, the synchronization scheme that executes $subQ_1$ first and then ships the result of $subQ_1$ to another source to join with the subquery $subQ_2$ is, relatively speaking, an undesirable access plan because of the poor performance it presents. The following simple heuristics may be used to rule out those undesirable synchronization alternatives.

   (a) The selectivity level of a *Fetch* condition with one of the following operators "<=, <, >, >=" is higher than the selectivity level of a *Fetch* condition with the operator "=", but lower than the selectivity level of a *Fetch* condition with "<>" operator.

(b) Subqueries with a lower level of selectivity are executed first. If there is more than one subquery of lowest level of selectivity, then they should be executed in parallel.

(c) Subqueries with highest level of selectivity are executed last.

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

It is well known that the quality of a query processor relies on the efficiency of its query processing strategies and the performance of its query execution plans. The performance of a distributed query processing plan is not only determined by the response time of the local subqueries but also affected by the synchronization scheme chosen for synchronizing the execution of subqueries. A good synchronization scheme may utilize the results of some subqueries to reduce the processing cost of the other subqueries whenever it is beneficial. Consider the example query reformulated at the end of Section 4.2.2. We shall illustrate the query decomposition of the reformulated query using the following symbols:

- Let X_1 denote "Job->title"
- Let X_2 denote "Job->pay"
- Let X_3 denote "Job->descrip"
- Let X_4 denote "Job->company" or "Company->name"
- Let X_5 denote "Company->address"
- Let X_6 denote "Company->city"
- Let X_7 denote "Company->prov/state"
- Let X_8 denote "Company->country"
- Let X_9 denote "Company->mail_code"
- Let X_{10} denote "Company->descrip"
- Let X_{11} denote " Company->URL"
- The predicate P_1 denotes the condition "Job->Descrip contains 'multimedia"'
- The predicate P_2 denotes the condition "Company->country contains 'Canada' || 'USA' "

The query decomposition process first generates the query tree graph as shown in Figure 4.7(a). Then the **target split** module is invoked. The procedure of target split starts by repeatedly replacing the compound interface based on generalization (or aggregation) by the distributed union (or distributed aggregation join) over the corresponding base interfaces as shown in Figure 4.7(b). This is done by searching the interface repository of the **Job Search Assistant** mediator (recall Figure 3.7 and Figure 3.8).

To replace the generalization-based interface Job, only DR2.OpenJob, DR3.Positions, and DR4.JobList are used since DR2, DR3, DR4 are the only sources that are relevant to this query. Similarly, for the Company interface only DR6 and DR7 are used.

The next step is to move the target downward close to the leaf nodes. The query tree is then transformed to the one in Figure 4.7(c). After grouping the leaf nodes by target source, the query is modified into the query tree shown in Figure 4.7(d). By applying the conventional distributed query processing tactics, such as moving selection down to the leaf level, performing selection and projection earlier, performing local joins before moving the data among sites for inter-source joins, the query is reformulated as shown in Figure 4.7(d).

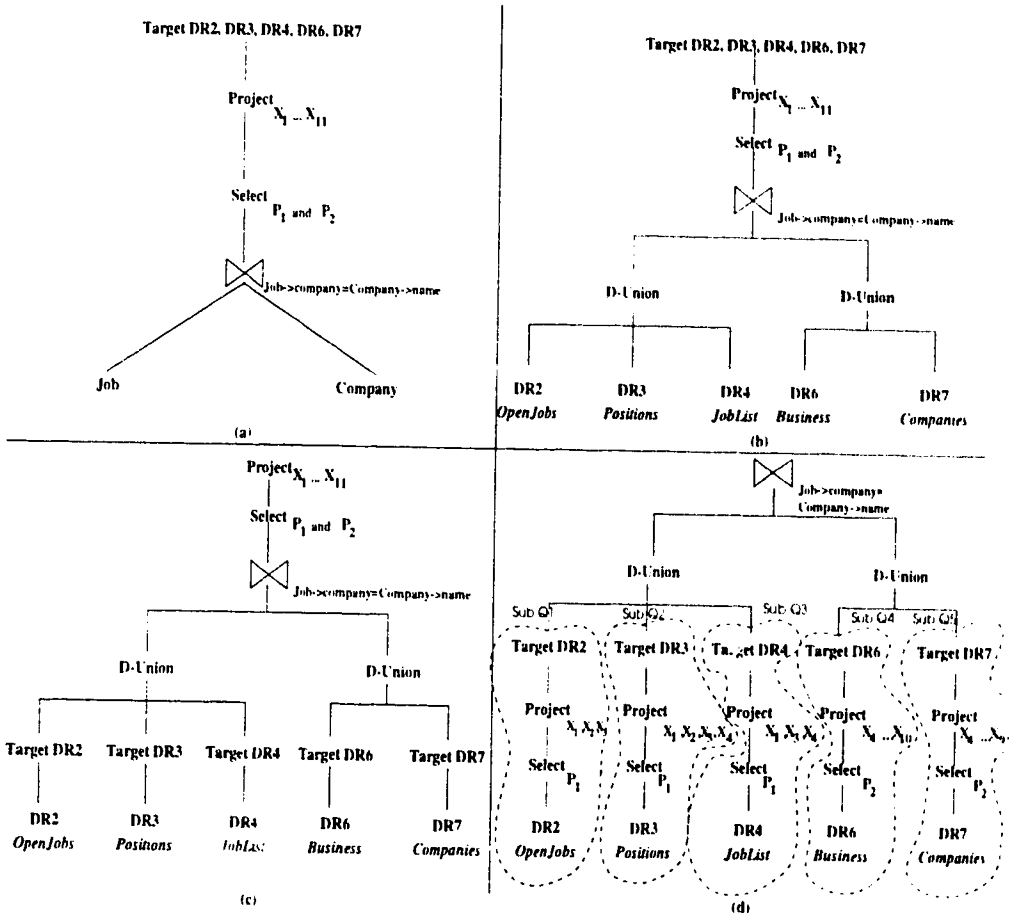


Figure 4.7: An example of query decomposition

The final query tree allows the system to detect whether or not the interfaces, Job and Company, are independent or dependent. If our repositories contained data for both interfaces then it would be possible to send queries containing local join conditions for the two interfaces (i.e., Job->name=Company->name). But in this example the system notices that no repositories contain data for both interfaces, and so the join must be performed after all the data for each interface has been collected (the join symbol in the query tree remains above all interfaces). Thus, our query shown in Figure 4.6 can be split into two separate queries for each type of interface. All repositories for the Job interface receive this query:

```
SELECT Job->title, Job->pay, Job->descrip, Job->company
FROM   Job
WHERE  Job->descrip contains 'multimedia'
```

Since there are three repositories (DR2, DR3, DR4) for the Job interface we send out three subqueries (SubQ1, SubQ2, SubQ3) which are identical as the above query but with the addition of a TARGET field specifying the repository (e.g. TARGET DR2).

This is the query sent to all repositories in the Company interface:

```
SELECT Company->name, Company->address, Company->city,
       Company->prov/state, Company->country, Company->mail_code,
       Company->descrip, Company->URL
FROM   Company
WHERE  Company->country contains ['Canada' || 'USA']
```

Again, since there are two repositories (DR6, DR7) for the Company interface we send out two subqueries (SubQ4, SubQ5) which are identical as the above query but with the addition of a TARGET field specifying the repository (e.g. TARGET DR6).

The original query is now transformed into an semantically equivalent DIOM distributed query algebraic expression: *D-Union*(SubQ1, SubQ2, SubQ3) and *D-Union*(SubQ4, SubQ5). The operation *D-Union* is a distributed union operator which adds the necessary semantic attachments to the results of each set to make them union-compatible before applying the union operation to combine the two operands. A detail discussion is provided in Section 4.2.6.

The second phase of query decomposition is the recording of useful subquery dependencies. According to the result of target split (Figure 4.7(e)), the original query in Figure 4.1 is transformed into an equivalent IQL query expression composed of five subqueries.

Thus the execution of the original query Q_1 can be reduced to the execution of the five subqueries with two distributed unions.

We list three possible execution plans here:

1. SubQ1 || SubQ2 || SubQ3 || SubQ4 || SubQ5;
2. SubQ1 $\mapsto$ SubQ2 || SubQ3 || SubQ4 || SubQ5;
3. SubQ1 $\mapsto$ SubQ2 $\mapsto$ SubQ3 $\mapsto$ SubQ4 || SubQ5.

The synchronization operator || is called *parallel dispatch* operator. The dependency expression SubQ1 $\mapsto$ SubQ2 indicates the sequence that SubQ1 is executed before SubQ2.

Because the subqueries are independent of each other we do not need to consider selectivities of each subquery when choosing the best execution sequence.

Therefore the parallel execution of all subqueries (i.e., sequence 1) is the only useful dependency with respect to the performance of subquery processing. This dependency annotation specifies that the submission and execution of all the subqueries should be carried out in parallel, the results returned for each union set can be merged directly in terms of the DIOM annotation-based union operator.

Obviously, in order to make the above subqueries executable over the corresponding information sources, each of the subqueries must be translated into the correct query statements expressed in the specific source query language. This is one of the tasks that will be carried out at the wrappers level since the subquery translation and execution can be seen as a source-specific task. A detailed discussion is found in Section 4.2.5.

We have given an introduction to the query routing and query decomposition strategies by using an example. Since the global optimization of mediated queries is beyond the focus of this paper, we will not further discuss selection of useful subquery dependencies, or the design of the DIOM distributed object algebra operators.

4.2.4 Parallel Access Plan Generation

The parallel access plan generation module is responsible for determining the final synchronization access plan for the set of subqueries resulting from query decomposition. The selection is based primarily on the number of useful subquery dependencies recorded at the query decomposition phase. The factors that may influence the final decision include (1)the size of the extents of the relevant information source classes or relations, and (2)the cache information available on the client side. This part of our query processing implementation is still under design.

4.2.5 Subquery Processing at Wrapper Level

After source selection, query decomposition and parallel access plan generation, the subqueries in IQL expressions will be passed from the mediator to the corresponding wrappers for subquery translation and execution.

Each wrapper will first convert the IQL query into a query expression that is understandable to the data source which the wrapper serves. When the repository is a RDBMS, say Oracle, the wrapper will map an IQL expression into the local language which in this case is an Oracle/SQL query statement.

It is also the wrapper's responsibility to collect and package the result in terms of DIOM objects before sending the result of a subquery back to the mediator. If the data source is an OODBMS (say ObjectStore), the wrapper will map each IQL expression into an ObjectStore query that is bound to an ObjectStore class dictionary. When the data source is stored and managed solely by a file server, say in the form of HTML, then the wrapper will map the IQL expression into, for example, a C module or a Perl module that scans the source data and returns the matching records.

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

1. **Attribute conversion and type replacement.** This is done by looking at the implementation repository metadata and finding the matching expression for each attribute and entity type in the IQL query expression, and by replacing all the attributes and interface types defined at the domain model level by their matching attributes and entity types at the information source model level. The target information source names are used to lead the search and conversion.

2. **Completion of the subquery expression.** Each information source may have specific requirements for representing a query over the source. For example, if the source is an Oracle database, a subquery over more than one relation must provide explicit join conditions among these relations to make this subquery executable over the Oracle database source. The role of the subquery expression completion step is to guarantee that the subquery to be executed at the source conforms to the source-specific query language requirement.

3. **Subquery refinement** The purpose of subquery refinement is to utilize the available constraints to further reduce the search scope of the subquery and to detect null queries.

Consider the *Job Search Assistant* scenario. Recall Section 4.2.3, where the query Q_1 was decomposed into five subqueries SubQ1, SubQ2, SubQ3, SubQ4, and SubQ5. The procedure of subquery translation proceeds as follows:

- **Step 1: Attribute conversion and type replacement** The procedure of subquery translation starts with attribute conversion and type replacement . For each subquery, the query decomposition module searches the source-specific implementation repository in the metadata catalog using the TARGET information source name as index, and replaces the mediated attributes and interface types by the corresponding source attributes and entity types defined at the information source model level. For instance, using DR3 as an index to search the metadata maintained in the implementation repository in Figure 3.7, we may easily find that the mediated attribute Job->pay has the matching source attribute Positions.pay in the source DR3. Following a similar approach, all the mediated attributes are replaced by the corresponding source attributes. Those attributes not available will be dealt with later during query result assembly (Section 4.2.6). The five subqueries generated at the query decomposition module (Section 4.2.3) are further modified as follows:

```
SubQ1  TARGET DR2
       SELECT OpenJob.jobtitle, Job.start_wage, Job.description,
       FROM   Job
       WHERE  Job.description contains 'multimedia'

SubQ2  TARGET DR3
       SELECT Positions.title, Positions.pay, Positions.description,
              Positions.company
       FROM   Positions
       WHERE  Positions.description contains 'multimedia'

SubQ3  TARGET DR4
       SELECT JobList.position, JobList.description, JobList.organization
       FROM   JobList
       WHERE  JobList.description contains 'multimedia'

SubQ4  TARGET DR6
       SELECT A.name, A.street, A.city,
              A.state, A.country, A.zip,
              BusinessInterest.description
       FROM   Business A
       WHERE  A.country contains ['Canada' || 'USA']

SubQ5  TARGET DR7
       SELECT A.name, A.address, A.city, A.province, A.country,
              A.postal_code, A.homepage
       FROM   Companies A
       WHERE  A.country contain ['Canada' || 'USA']
```

Obviously, these subqueries are still incomplete with respect to relational systems because relational SQL requires that (i) all relations involved in a query be declared in the FROM clause; and (ii) the join condition be explicitly expressed when multiple relations are involved in a query. Thus, the next step of the subquery translation is to complete the subquery expression and to refine the subqueries according to the constraint information available in the export schema of the source.

- **Step 2: Completion of the subquery expression** Since both the source DR2 and the source DR6 are relational data repositories, according to relational SQL, all the relations involved in the query should be declared in the FROM clause, and the join conditions among the relations should be explicitly expressed in the WHERE clause. Based on the metadata maintained by the wrapper to DR2 and the wrapper to DR6 (recall Figure 3.7, Figure 3.8 and Figure 3.9), the join condition "Job.jobtitle = OpenJob.jobtitle" should be added into the WHERE clause of the subquery SubQ1. The join condition "Business.name = BusinessInterest.name" should be added into the WHERE clause of the subquery SubQ4. The rest of the subqueries do not require any further completions. Thus, the subquery SubQ1 is further modified at the corresponding wrapper to DR2 as follows:

```
SubQ1: TARGET DR2
       SELECT OpenJob.jobtitle, Job.start_wage, Job.description,
              OpenJob.opendate
       FROM   Job, OpenJob
       WHERE  Job.description contains 'multimedia'
              AND Job.jobtitle=OpenJob.jobtitle
```

The subquery SubQ4 is modified at the corresponding wrapper to DR6 by adding the additional join condition to the WHERE clause and the additional relation to the FROM clause:

```
SubQ2: TARGET DR6
     SELECT A.name, A.street, A.city, A.state, A.country,
            A.zip, B.description
     FROM   Business A, BusinessInterest B
     WHERE  A.country contains ['Canada' || 'USA']
            AND A.name=B.name
```

4.2.6 Query Result Assembly

Once all the subqueries are processed and the results are returned to the wrappers (the DIOM local agents), the results must be assembled and attached with additional semantic information before being presented to the user in terms of the user's preferred terminology. Instead of enforcing the integration of structural heterogeneity and semantic heterogeneity in various information sources, we propose a semantic attachment approach to assemble the query results. The techniques we apply include source attachment, scale attachment, access path attachment, and absent attribute specification. These techniques are described below using our example from the *Job Search Assistant* application scenario given in Section 4.1.1 (recall the query from Figure 4.1).

After the subqueries are processed based on the subquery synchronization plan generated from the phase of parallel access plan generation, the results of these five subqueries are shown in Figure 4.8.

By applying the following *semantic attachment* techniques to the query results, the outcome of the query results assembly is shown in Figure 4.9 and Figure 4.10.

- *Nonexisting attribute specification*
  All mediated attributes which do not convert to any attribute in the local information source must be noted when assembling the subquery results. For example, the mediated attribute Job->pay does not convert to any attribute in the subquery to information source DR6 because there is no corresponding attribute available.

- *Scale attachment*
  The scale for the mediated attribute Job->pay may be different for different information sources. For example, the unit of the pay attribute in the information source DR is Canadian Dollars (CAN$) while the unit of the pay attribute in the rest of the information sources are in US dollars (US$). Thus, when the user does not explicitly indicate what the preferred currency unit should be for the mediated attribute Job->pay, we add the scale attachment to the query result, rather than enforcing the price values returned from different information sources to be either US$ or CAN$.

- *Source attachment*
  We also provide the source attachment option as a default. It means that unless the user explicitly excludes the source attachment, we will attach each returned result object with the name of its source. For instance, the first tuple in Figure 4.9 is from the information sources DR6 and DR4.

- *Outer Join Option*
  It may be useful to be able to perform outer joins on our mediator interfaces. System IQL queries can be designated as outer join queries where all records which were touched in the processing of the query are returned back to the user. Figure 4.10 shows the final results of the example query when using the outer join option. The first record shows the company *Fujitsu Business* as having company information but no jobs offered. There is also a job listing for *Senior Multimedia Consultants* but no company information available. These queries are not shown in Figure 4.9.

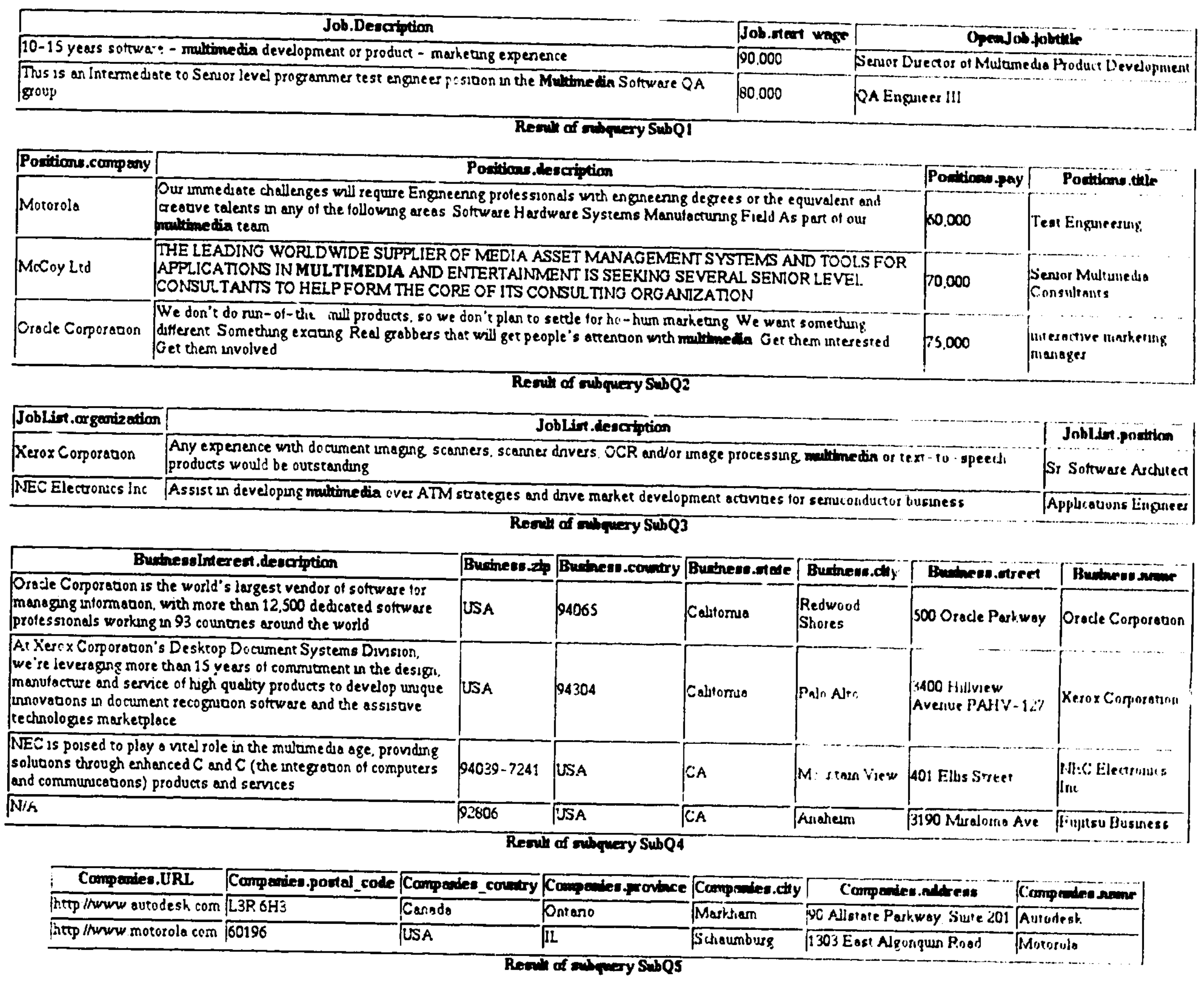
4.3 Benefits of the Metadata Model

The DIOM metadata model provides a simple scheme for managing distributed metadata, and a high-level query language for the multidatabase users to request and gather information from multiple and disparate information sources. The advantages of using the DIOM object model and query language are the following:

- First, users may express queries even when they have no precise knowledge of which information sources are relevant. The information source location will be dynamically selected by the system using the DIOM catalog functions. Our dynamic information source selection mechanism allows the new information producers' data sources to be selected upon their arrival.

- Second, users are no longer required to specify the inter-data source connections or the intra-database join conditions (typically the joins between primary key and foreign key attributes) in the query, as they are inferred automatically by the system using the metadata. When compared with writing traditional SQL queries such as in a RDBMS, this simplifies the query formulation and query planing process.

- In addition, the DIOM global query model encourages the delay of resolution of the heterogeneity problems in a multidatabase system to the query evaluation time. The DIOM system resolves the heterogeneity problems by combining an adaptive approach to query planning with a semantic enrichment scheme for result assembly, rather than through the enforcement of an integrated and monolithic global schema.



Job.Description	Job.start wage	OpenJob.jobtitle
10-15 years software - **multimedia** development or product - marketing experience	90,000	Senior Director of Multimedia Product Development
This is an Intermediate to Senior level programmer test engineer position in the **Multimedia** Software QA group	80,000	QA Engineer III

Result of subquery SubQ1

Positions.company	Positions.description	Positions.pay	Positions.title
Motorola	Our immediate challenges will require Engineering professionals with engineering degrees or the equivalent and creative talents in any of the following areas Software Hardware Systems Manufacturing Field As part of our **multimedia** team	60,000	Test Engineering
McCoy Ltd	THE LEADING WORLDWIDE SUPPLIER OF MEDIA ASSET MANAGEMENT SYSTEMS AND TOOLS FOR APPLICATIONS IN **MULTIMEDIA** AND ENTERTAINMENT IS SEEKING SEVERAL SENIOR LEVEL CONSULTANTS TO HELP FORM THE CORE OF ITS CONSULTING ORGANIZATION	70,000	Senior Multimedia Consultants
Oracle Corporation	We don't do run-of-the mill products, so we don't plan to settle for ho-hum marketing We want something different Something exciting Real grabbers that will get people's attention with **multimedia** Get them interested Get them involved	75,000	interactive marketing manager

Result of subquery SubQ2

JobList.organization	JobList.description	JobList.position
Xerox Corporation	Any experience with document imaging, scanners, scanner drivers, OCR and/or image processing, **multimedia** or text-to-speech products would be outstanding	Sr Software Architect
NEC Electronics Inc	Assist in developing **multimedia** over ATM strategies and drive market development activities for semiconductor business	Applications Engineer

Result of subquery SubQ3

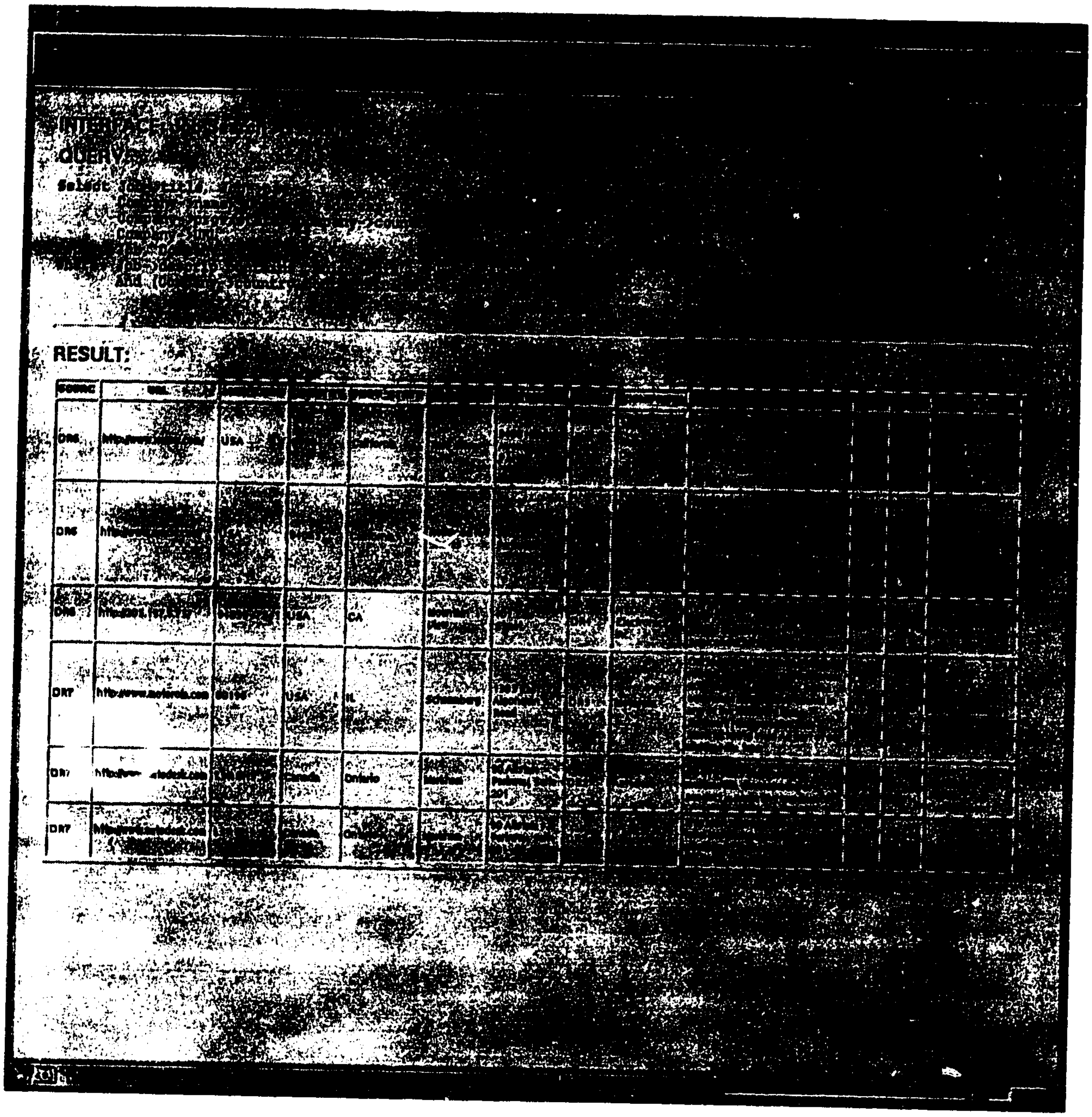
BusinessInterest.description	Business.zip	Business.country	Business.state	Business.city	Business.street	Business.name
Oracle Corporation is the world's largest vendor of software for managing information, with more than 12,500 dedicated software professionals working in 93 countries around the world	USA	94065	California	Redwood Shores	500 Oracle Parkway	Oracle Corporation
At Xerox Corporation's Desktop Document Systems Division, we're leveraging more than 15 years of commitment in the design, manufacture and service of high quality products to develop unique innovations in document recognition software and the assistive technologies marketplace	USA	94304	California	Palo Alto	3400 Hillview Avenue PAHV-127	Xerox Corporation
NEC is poised to play a vital role in the multimedia age, providing solutions through enhanced C and C (the integration of computers and communications) products and services	94039-7241	USA	CA	Mountain View	401 Ellis Street	NEC Electronics Inc
N/A	92806	USA	CA	Anaheim	3190 Miraloma Ave	Fujitsu Business

Result of subquery SubQ4

Companies.URL	Companies.postal_code	Companies_country	Companies.province	Companies.city	Companies.address	Companies.name
http //www autodesk com	L3R 6H3	Canada	Ontario	Markham	90 Allstate Parkway Suite 201	Autodesk
http //www motorola com	60196	USA	IL	Schaumburg	1303 East Algonquin Road	Motorola

Result of subquery SubQ5

Figure 4.8: Results of the subqueries



INTERFACE [illegible]

QUERY [illegible]

Select [illegible]

[illegible] Company [illegible]

[illegible] And [illegible]

RESULT:

SOURC	URL	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
DR6	[illegible]	USA	[illegible]	[illegible]	[illegible]	[illegible]
DR6	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
DR6	http://201.197.[illegible]	[illegible]	USA	CA	[illegible]	[illegible]
DR7	http://www.motorola.com	60196	USA	IL	Schaumburg	[illegible]
DR7	http://www.[illegible]desk.com	[illegible]	Canada	Ontario	[illegible]	[illegible] 201
DR7	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Figure 4.9: Final query result

Chapter 5

The Rainbow Prototype Design

In DIOM environments, the information sources may differ by their organization (e.g., traditional databases, knowledge bases, and flat [illegible] y their content (e.g., HTML hypertext, relational tables, and objects of complex [illegible] ictures), and by the many browsers and graphical user interfaces (GUIs) as well as q[illegible] languages used for information access. It would be convenient and beneficial if there existed software systems that provided facilities allowing for easy interoperation among these diverse information sources.

Rainbow implements an interoperable software architecture based on DIOM to support USECA properties. The foundation of the Rainbow architecture design is with the DIOM system architecture [31] described in Chapter 2. Therefore, for background material, the reader may refer to the earlier presentation of the DIOM system architecture during the presentation of the following Rainbow prototype design.

5.1 Rainbow Detailed System Components

This section describes the Rainbow design and prototype of the DIOM query mediation methodology, which offers querying capabilities over multiple heterogeneous data sources. This tool was designed to illustrate the functionality of the DIOM interoperable object model, architecture and query mediation process. The target environment was the Internet and the target was a Job Search Assistant application.

Rainbow is implemented as a WWW application. Its interface functions are created using HTML [14] and Perl [52] CGI-scripts. Linking to the underlying data sources and metadata library database is implemented with Oraperl [5].

Figure 5.1 illustrates the detailed system components for Rainbow, which includes the clients, the Rainbow Services Managers (RSMs), the Official Metadata Catalogs (OMCs), the wrappers, and the data repositories. All components are connected by network. Figure 5.2 shows the connections from the DIOM clients to a RSM which is designed using a client-server model. Each client connects to a RSM server which processes client requests and can simultaneously handle more than one client.

All components of the system interact through the HTTP [17] protocol. This is a popular protocol which has good flexibility in types of data that can be sent and received (both text and binary). Using HTTP also allows us to focus on the components and non-network issues because of the large HTTP support base of tools and libraries. Furthermore, using a popular protocol such as HTTP allows Rainbow components to be more readily incorporated into new developments of the DIOM system as well as other external systems.

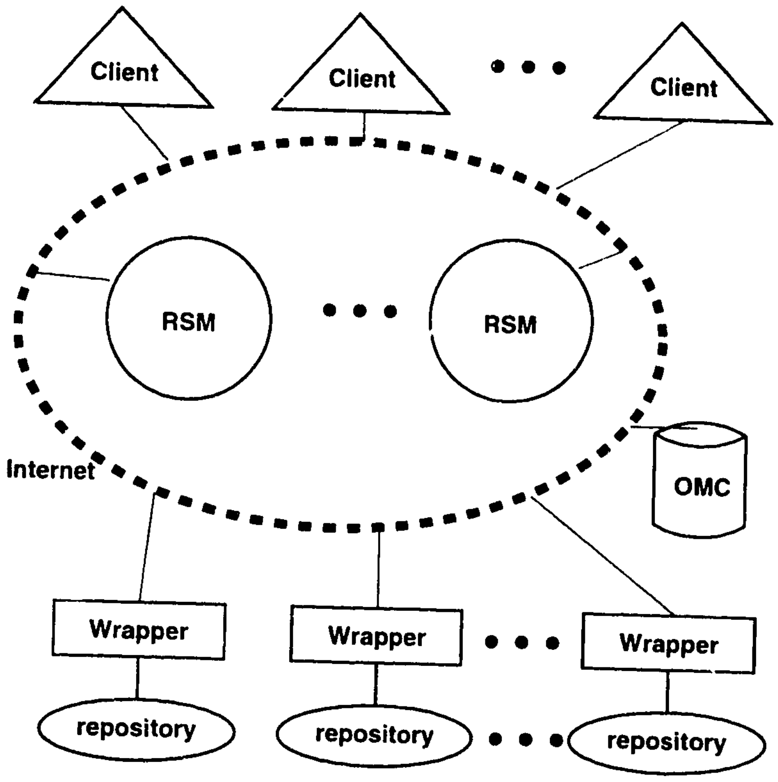


Figure 5.1: Rainbow detailed component diagram

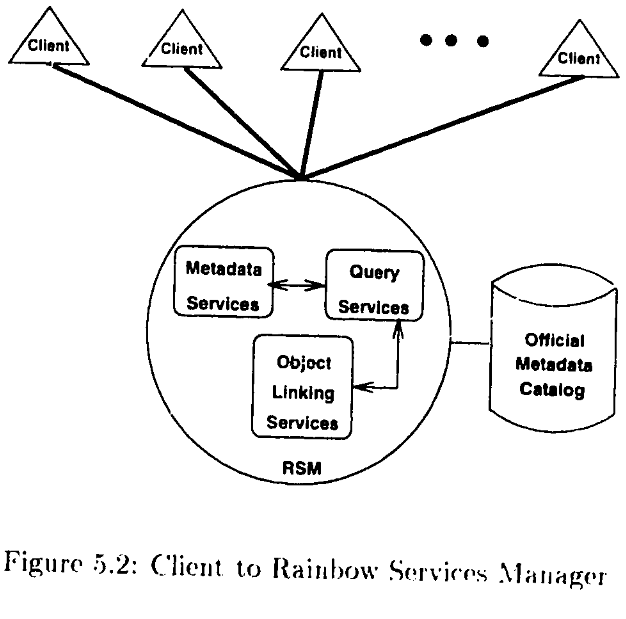


Figure 5.2: Client to Rainbow Services Manager

5.1.1 Clients

Clients consist of any popular GUI interface and underlying client modules to the Rainbow system using the DIOM client API. Clients communicate with the RSM using the HTTP protocol as the underlying network protocol and using a Perl API of available RSM commands. Clients may be implemented using any GUI-tool such as Web Applications (HTML-based), TCL/TK applications, Smalltalk applications etc. Applications can be official Rainbow clients which allows for a full range of functionality or specialized clients. Specialized clients may offer limited functionality where only one working view of data is offered (e.g., a Real Estate information application).

5.1.2 Rainbow Services Manager

The Rainbow Services Manager (RSM) is the primary service module which receives and processes client requests. The RSM services include metadata services (such as repository registration, consumer interface creation, and metadata browsing), query services (such as query installation, query retrieval, and distributed query processing). and object linking services (see Figure 3.2). Each RSM is identical, offering identical services, and is designed to be replicated throughout the network.

All consumer and producer metadata creation and maintenance is through the RSM. The RSM accepts metadata for the creation of new consumer interfaces or the registration of new repositories and returns whether or not the metadata was successfully registered. The RSM also allows for the browsing and searching of consumer and producer metadata.

Object linking services link consumer interfaces with the relevant repositories which fulfill that interface. The RSM must use system metadata on the interface and on the producer sources to do this linking. The RSM currently does this matching dynamically where the repositories are chosen when the query is executed. Future implementations may choose to preprocess the query and save the relevant repositories during query composition which would allow for better optimization and quicker query execution.

The RSM is also responsible for providing distributed query processing facilities. DIOM-IQL queries are composed and can be installed within the system. These installed queries are saved and managed by the RSM. Query execution involves loading the necessary metadata (i.e., instantiating the appropriate mediator interface objects and producer source models) and following the query processing steps of source selection, query decomposition, and parallel access plan generation. The RSM then sends the subqueries to the individual wrappers which return the results back to the RSM for result assembly. Figure 4.4 illustrates the query processing steps of the RSM and the query processing steps of the wrapper.

5.1.3 Official Metadata Catalog

To process requests each RSM communicates with an Official Metadata Catalog (OMC) server. For example, when a RSM needs to load mediator information to process a consumer query, it instantiates the mediator by retrieving the mediator definition from the OMC. OMC's are identical and store the metadata received from the RSM. The OMC handles the storage of the information source repository and interface repository (see Chapter 3). OMC's can also be replicated but should remain under the control of a Rainbow system administrator for the following reasons: (1) metadata should be carefully maintained and verified by Rainbow system administrators to maintain system integrity and security, (2) information is a valuable commodity whi[illegible] can be exploited commercially. OMC's can be strategically placed throughout the Internet for optimal performance. A replication algorithm is needed to maintain consistency of metadata between all OMC's.

5.1.4 Wrapper

Each different repository type needs a wrapper to connect with the system and ultimately to the consumers. To facilitate the incorporation of new information sources, the wrappers are specified with a generic function suite (see Chapter 7) which is composed of underlying local fun[illegible]ns. New sources are then able to participate by implementing these underlying functions. Wrapper generation may be partially automated by having meta-wrappers produce skeleton wrapper code depending on the category of data repository (structured, semi-structured, unstructured). Using the DIOM specified wrapper interface, wrapper creation for new types of informatio[illegible] [illegible]ositories can be partially automated.

A generic wrapper archite[illegible] [illegible]useful for the automated construction of wrappers to different types of data reposit[illegible] [illegible]he base set of functionality for the wrapper design is grouped into query, update, a[illegible] [illegible]ocal metadata functions. These function requests are translated by the wrapper into the local command equivalent and the result is returned as a DIOM result object. Thus, only those methods local to the repository need to be re-implemented.

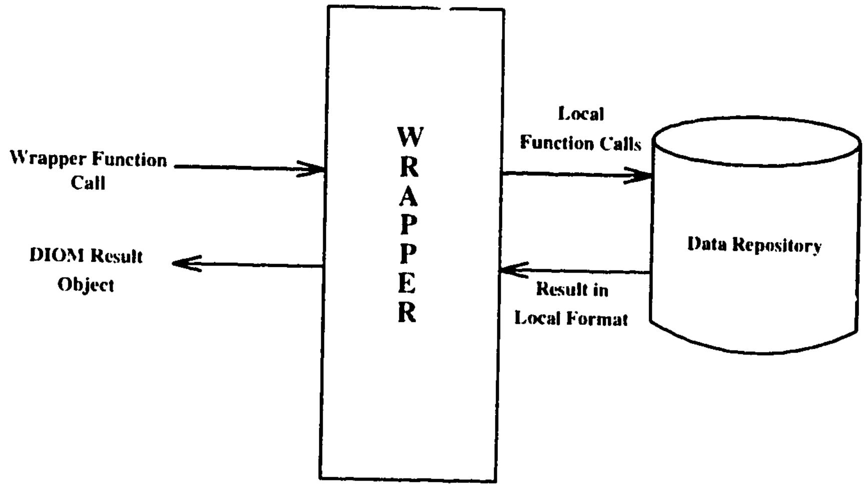


Figure 5.3: Wrapper translation of request and results

A sketch of our wrapper architecture is shown in Figure 5.4. The Wrapper receives a service request from mediators via the Rainbow Services Manager. The DIOM service request is processed by the subquery translation module which translates the service request or consumer query (in DIOM-IQL) into the local language. The source-specific metadata, provided by the source producer and pre-processed by the DIOM metadata catalog manager, is used to facilitate the translation. For example, a wrapper to Oracle DBMS receiving a DIOM-IQL query will translate the IQL into the Oracle SQL equivalent using the local equivalent relations and attribute names. Next, the subquery refinement module utilizes the available constraints to handle the possible refinement of the subquery, such as the elimination of null queries and the reduction of the query search space. The subquery execution module then sends the query to the local manager for local execution. The local query result packaging module takes the returned result which is in local format and normalizes the data into a DIOM-specific data format. Depending on the functionality of the local repository the wrapper may have to perform different amounts of processing (e.g., text based records would require all processing done by the wrapper vs. database repository where much processing can be done by the repository).

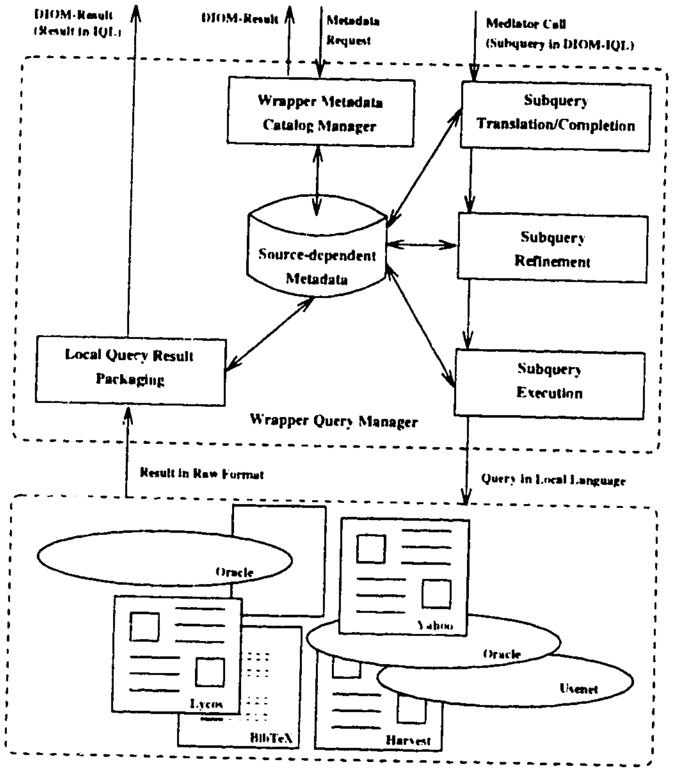


Figure 5.4: Wrapper Architecture

5.2 Implementation Considerations

The current implementation of the RSM is through Perl and the metadata catalogs are saved as Oracle relations. Clients access the RSM using a URL address with the commands embedded as part of the URL search string. The wrapper implementation will be discussed in more detail in Chapter 7.

Because the components are interoperating via HTTP, the individual components can easily be replicated and made to run on different hosts throughout the Internet. The network details are taken care of by HTTP.

The RSM is able to register new repositories, register new consumer interfaces, browse metadata, and process consumer queries. This requires much more work in dynamically decomposing the query and assembling the results. Currently, the OMC sub-system is integrated within the RSM. Work needs to be done in creating a separate OMC sub-system which can replicate data between other OMC sites. Limited querying capabilities are available such as simple keyword based queries and SQL-like queries over the pre-installed Job Search Assistant mediator. Query tracing is simulated and not implemented.

A completely working interoperable system will require much more work and research into many areas. Much of the underlying sub-system components are *simulated* and need to made fully functional.

Chapter 6

The User Interface

This chapter describes the Rainbow design of the HTML-based DIOM interface accessible by World Wide Web clients such as **Netscape** [42]. Specifically the following functionality was taken into consideration in the user interface design:

- Producer Metadata Facilities
  - Registration of new repositories including individual information sources and external information brokers such as Lycos. Infoseek etc.
  - Update of previously registered repositories
  - Browse or filter to find repository metadata
- Consumer Metadata Facilities
  - Generation of a DIOM-IDL specification for a DIOM-IQL query
  - Composition and submission of new consumer interfaces defined using DIOM-IDL.
  - Update of previously entered consumer interfaces
  - Browse or filter to find an IDL interface definition
- Query Facilities
  - Simple keyword query form and query trace interface
  - Expert query form based on DIOM-IQL
  - Query processing trace interface including:
    - interface to query routing
    - interface to query decomposition
    - interface to local query processing
    - result assembly screen
  - Browse or filter to find IQL definitions of installed queries

6.1 System Startup

Starting up the Rainbow prototype requires access to a web client which supports tables and frames (e.g., Netscape 2.0). The web client starts the application by opening a URL [1] to the Rainbow main menu. The initial main menu page is shown in Figure 6.1. The web page in Figure 6.1 is broken up into three main sections. First, the menu options at the top of the screen are Netscape program functions which are beyond the scope of this discussion (See [42] for Netscape details). Second, the larger frame to the right of the screen offers all major Rainbow features: the Producer Metadata Facilities, the Consumer Metadata Facilities, and the Query Facilities. Lastly, the smaller frame on the left side offers a functionally identical menu which remains throughout the rest of the Rainbow session [2] to allow for convenient access to all the major functions. The user can now choose any function by mouse-clicking the boxes in the main frame or any of the underlined words in the side menu.

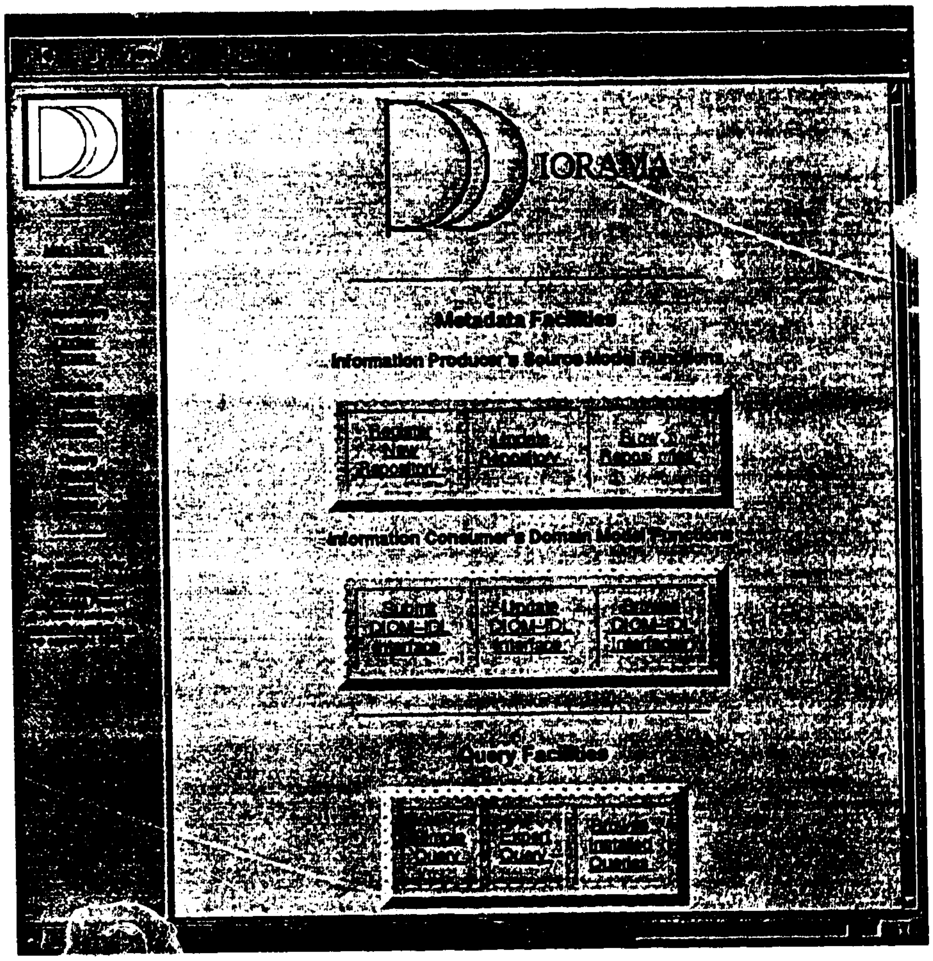

IORAMA
Metadata Facilities
Information Producer's Source Model Functions
Information Consumer's Domain Model Functions
Query Facilities

Figure 6.1: Rainbow main menu

[1]Uniform Resource Locator (e.g., http://www.cs.ualberta.ca/)

[2]for the purposes of the following user interface discussion this left frame will be omitted in the screen figures

6.2 Producer Metadata Facilities

6.2.1 Source Registration and Update

All information sources must be registered to be utilized by the DIOM system. It is assumed that repository registration is performed by local repository owners or local repository administrators who possess knowledge about their repository. There is some work required by the administrator in registering their repository.

Mouse-clicking the *Register New Repository* or the *Update Repository* links from the main menu brings us to the interface shown in Figure 6.2. For new registrations the fields to be filled in are left blank. For update registrations, a preliminary dialogue which is used to query the user for a DIOM repository ID. Using the repository metadata, the repository information fields will be filled. Unique DIOM repository IDs are assigned to all new repositories when they first register with DIOM.

The first field *Repository Name* simply allows the administrator to associate an intuitive general name to the repository. For example, in Figure 6.2 the name assigned is `Jobs Database`. Next, and of significant importance, is the URL to the repository wrapper. This URL is used to access the repository wrapper which must be installed before registration (see Chapter 7).

The next field, *Repository Source Type*, records the underlying system type which the repository is actually running. For example, repository sources types can be HTML pages, Oracle DBMSs, Informix DBMSs, Sybase DBMSs, etc. New types of repositories can be entered using the *other* text input field.

The *Repository Export Schema Name* allows the local repository to have multiple export schemas under different names. This allows different parts of their information to be exported separately perhaps for more logical organization. There is a default export schema name in the case no name is specified (as shown in the example).

The *Repository Owner* is the owner or maintainer of the repository. In this case it is **Autodesk Inc.**.

The next input field are the *Repository Keywords*. The repository maintainer supplies us with relevant keywords for their repository. These keywords can be directly related to the repository, such as relation names, or indirectly related. These keywords are critical as they will be used later on in connecting consumer interface descriptions to the producers' source repositories. The list of descriptive keywords must be chosen carefully and must be complete if the repository is to be utilized to its fullest potential [3]. In the example for the `Jobs Database` the repository keywords, `Jobs`, `OpenJobs` and `Employees` correspond to the relation names.

For each keyword, we may want to gather more semantic information. Figure 6.3 shows one way to gather more semantic information. This example allows us to add more keywords for each of the previous keywords. Eventually, for a given application domain, this interface could be advanced to ask more intelligent questions about the keywords. For example, given the keyword `Jobs`, the system could ask the registrant whether they mean "open job listings" or "tasks to do" or "employment".

The final step in registration involves the system checking the URL for a valid wrapper, assigning a DIOM repository ID, installing the repository information into the metadata catalog, and reporting the results (shown in Figure 6.4) back to the user.

[3] Again ontology studies and research may help to uncover more semantic information for the repository

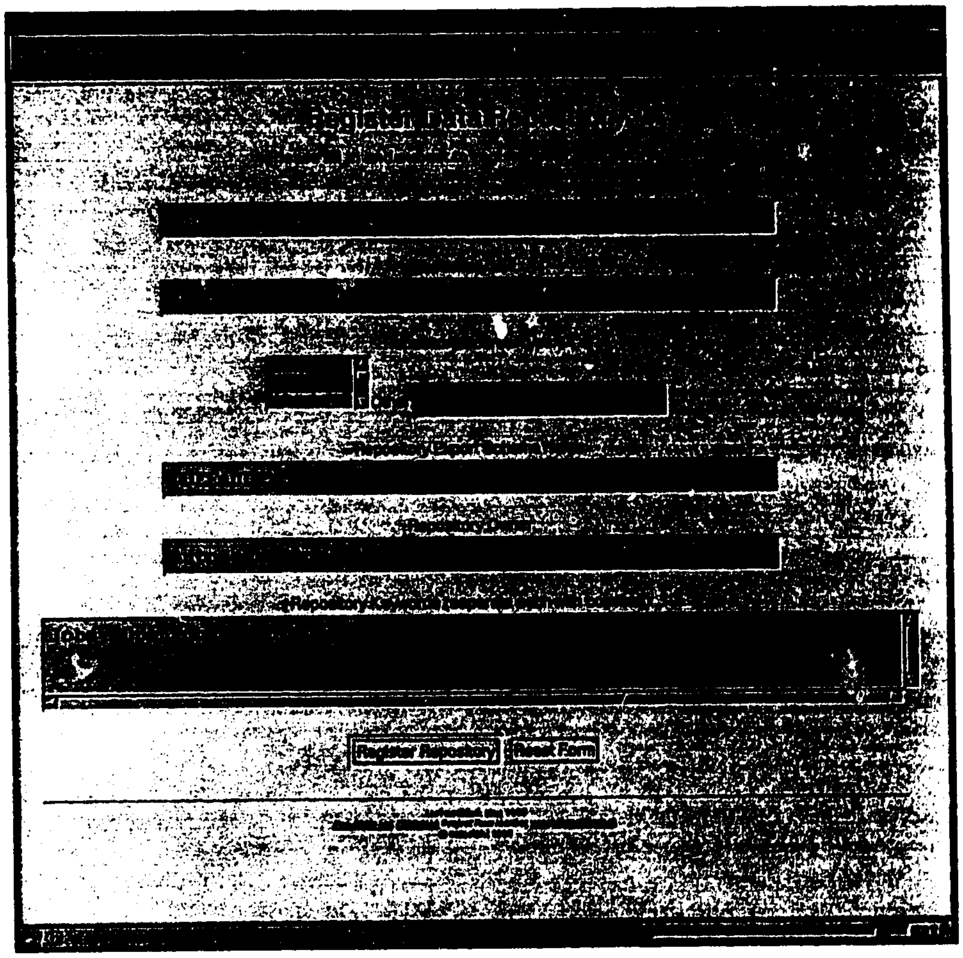


Figure 6.2: Registration Dialog

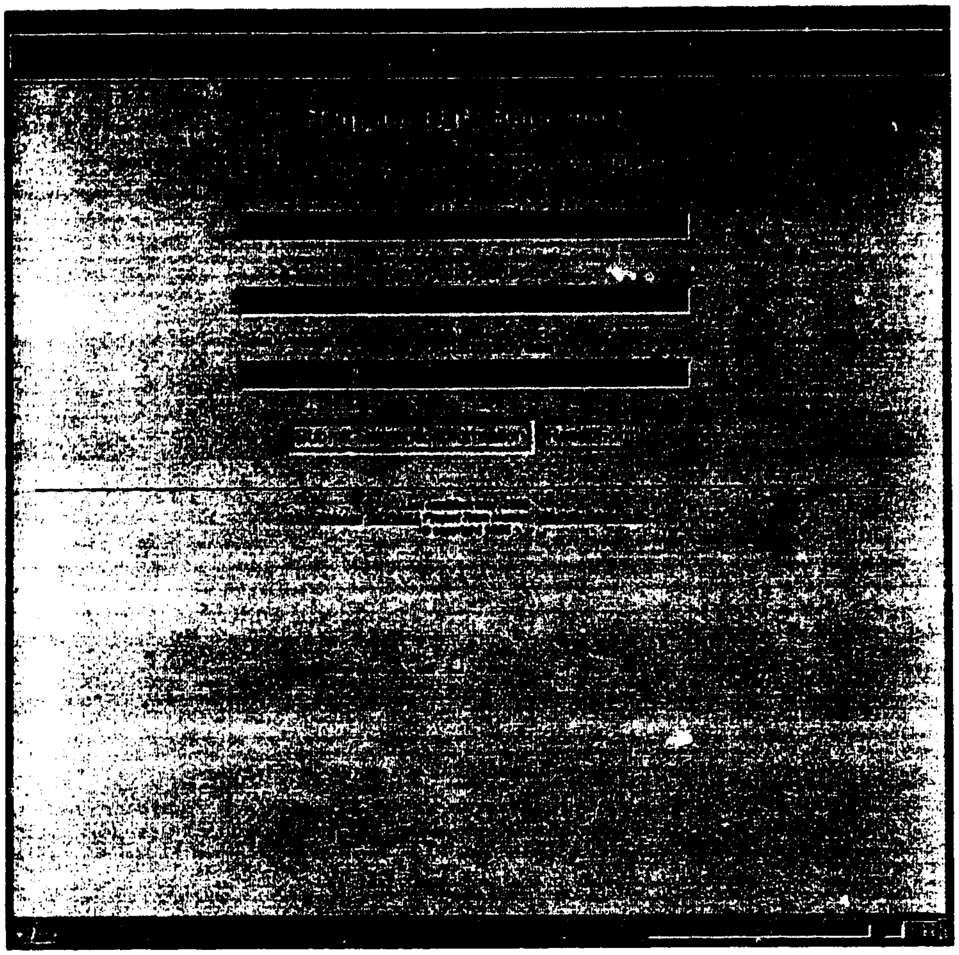


Figure 6.3: Enter more information on keywords

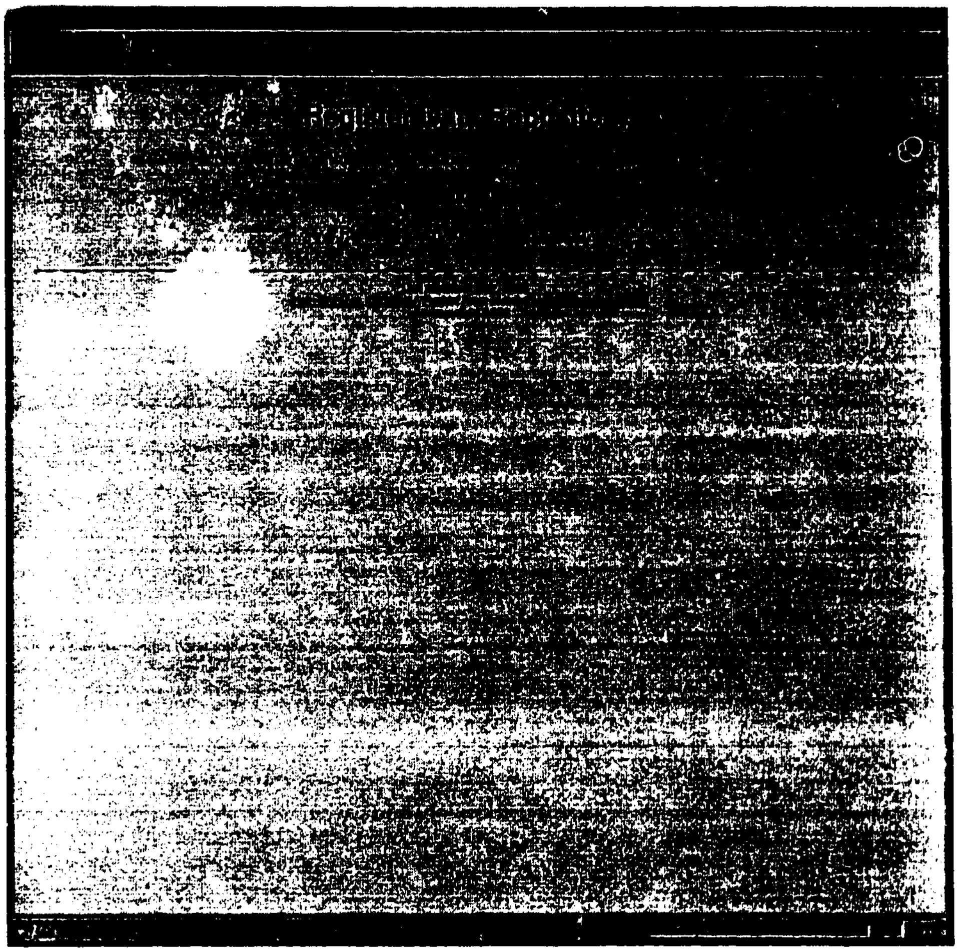


Figure 6.4: Repository is accepted

6.2.2 Source Metadata Browsing

The source metadata browser tool allows users to browse repository metadata from the DIOM metadata catalog, including metadata that was entered during registration. This browsing tool is intended for those users who are interested in the repositories that are online. It offers a convenient service for browsing individual repositories.

From the main menu, choosing the *Browse Repositories* option gives us the screen in Figure 6.5. The first option, *Show all registered repositories*, located at the top of the screen retrieves metadata on all registered repositories. The filter fields located on the bottom half of the screen are used to find specific repositories. The filter fields are a mirror of the fields used during repository registration. Any of the fields can be used to query for certain repository information. For example, by inputting a repository ID a unique repository can be found or by entering a partial URL all repositories with partially matching URLs will be listed in the result. The result screen in Figure 6.6 lists the resultant information repositories in table format with all metadata catalog information.

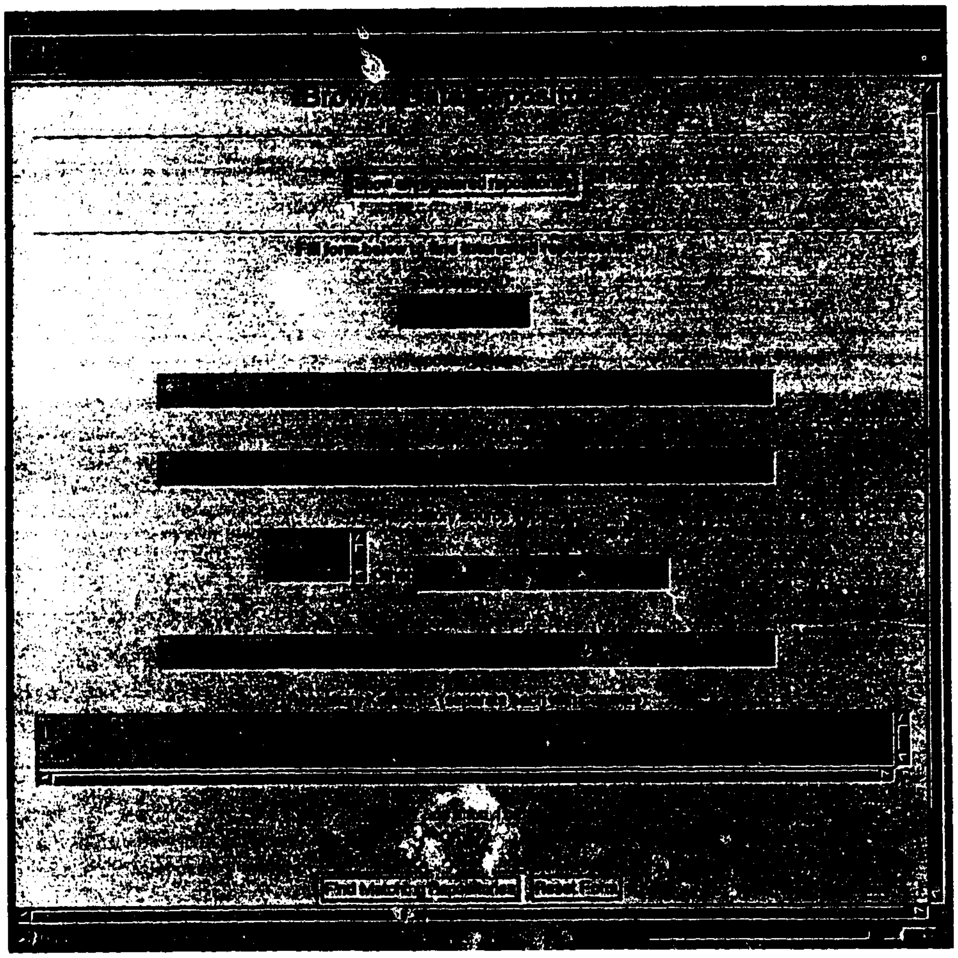
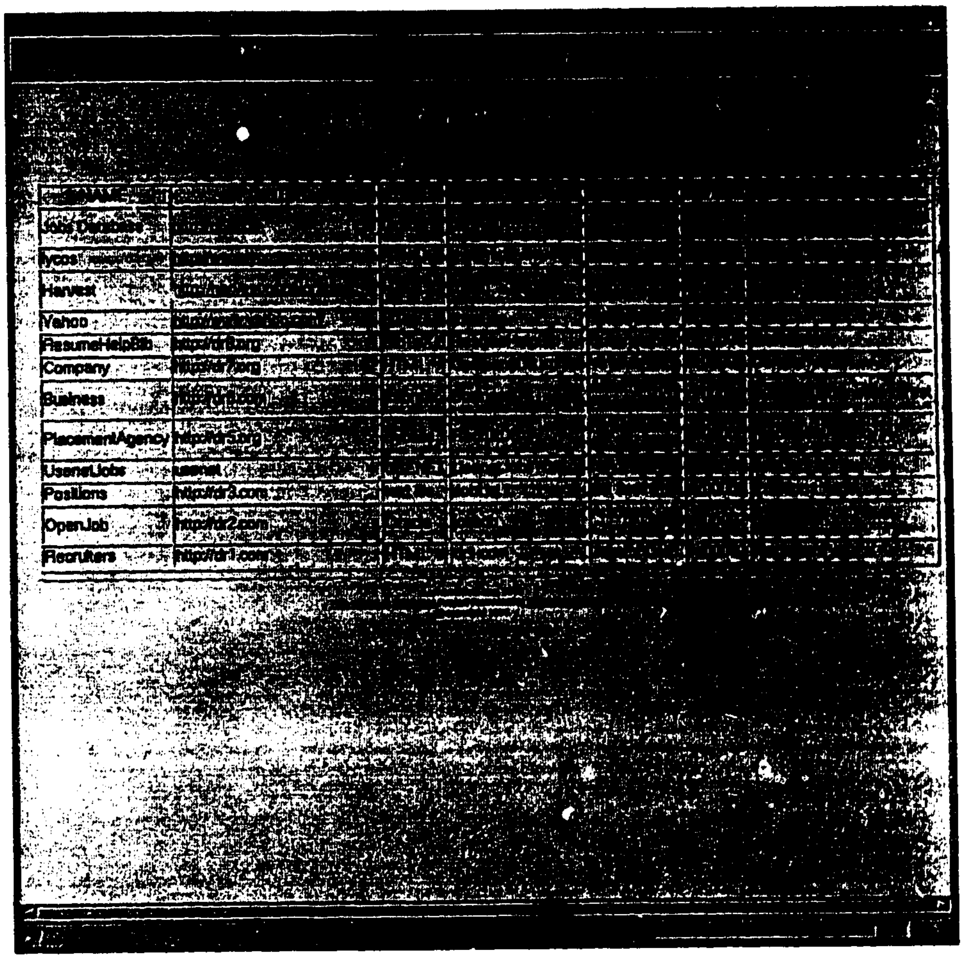


Figure 6.5: Filter to find repository metadata



[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Jobs Database	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
lycos	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Harvest	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Yahoo	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
ResumeHelpBib	http://dr8.org	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Company	http://dr7.org	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Business	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
PlacementAgency	http://dr5.org	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
UsenetJobs	usenet	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Positions	http://dr3.com	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
OpenJob	http://dr2.com	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Recruiters	http://dr1.com	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Figure 6.6: All repository metadata shown

6.3 Consumer Metadata Facilities

6.3.1 Consumer IDL Interface Editor

DIOM-IDL interface definitions as described in section 3.3 can be composed manually, but to facilitate composition a user friendly interface which allows for basic definitions and mixed definitions was created. The interface definitions define the objects the consumer is seeking and the terminology used to access the objects. The system takes the interface definitions and links the object definitions with objects from the available producers. Therefore, having the interface well defined is important for the automatic object linking and embedding services.

The DIOM-IDL interface editor (Figure 6.7) closely follows the IDL syntax given in Section 3.3. The interface is first grouped with a mediator using the select option or a new mediator grouping can be created. The interface is then given a name (i.e., *Interface Name*) and a high-level English description (i.e., *Interface Description*). In the example shown, the user is creating a job interface to list open job listings. The interface name is **Job** and the description is **job listings**. Next we come to choosing the *Interface Abstraction Type*. The abstraction type can be left open in which case the default option will be used which automatically chooses one of the abstractions. Automatic interface creation permits the system to attempt to fulfill the interface by trying out the different abstractions automatically. For example, in using the automatic abstraction type for our Job interface, the system will attempt generalization first, then aggregation to produce a Job object from the available data sources.

The interface creator can also have more control over the creation of interfaces by using one of the abstraction mechanisms. For an example of aggregation abstraction, consider the situation where an interface composer may have previously specified other interfaces say X-ray Image and Doctor Diagnosis. Now, to create a new interface, Patient Record, they may want to aggregate the X-ray and Diagnosis interfaces. This convenience allows for reuse of previously created interfaces.

The next field, *Interface Keywords*, requests the keywords needed to choose the repositories to use for the abstraction. In our example for the generalization of a Job interface, we enter **Job** and **Position** as the keywords in order to find those repositories which will have either **Job** or **Position** information to form our composite job interface.

The attribute editor on the bottom half of the screen is a Java [4] applet which allows for easy editing of attributes. Attributes have a name, type, and description. The name is a semantically relevant word to the attribute, the type can be a basic type such as String or Float, and the description box keeps an English language description of the attribute. When matching attributes from this interface to attributes in a repository, the name of the attribute along with words from the description are used in the matching process. For example, the **title** attribute of the Job interface is matched to object attributes **position**, **jobtitle**, and **name** from our example repositories.

The attribute list box below keeps a list of the attributes added and is a convenient way to summarize the attributes.

Finally, the composed interface definition may be submitted for processing. The interface definition is saved in the metadata catalog and a preliminary linking of repositories with this new interface is done. The result screen (Figure 6.8) reports the id's of the matching repositories and the successful completion of interface creation.

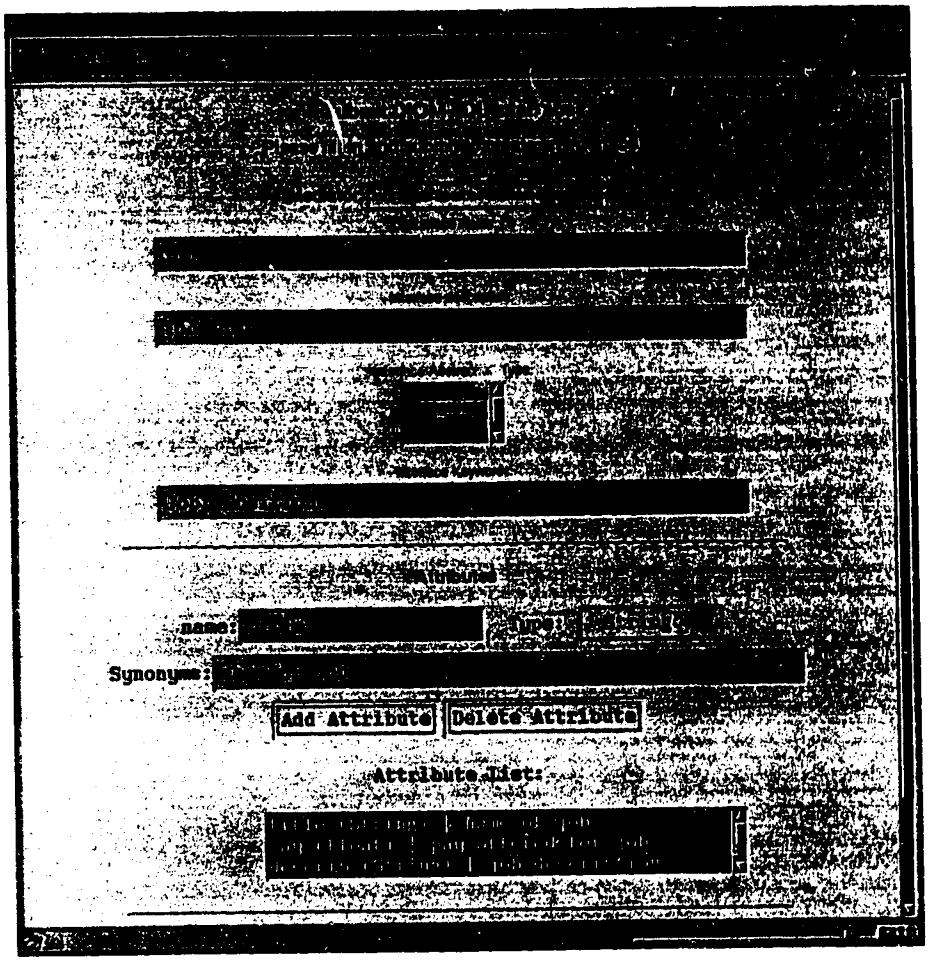
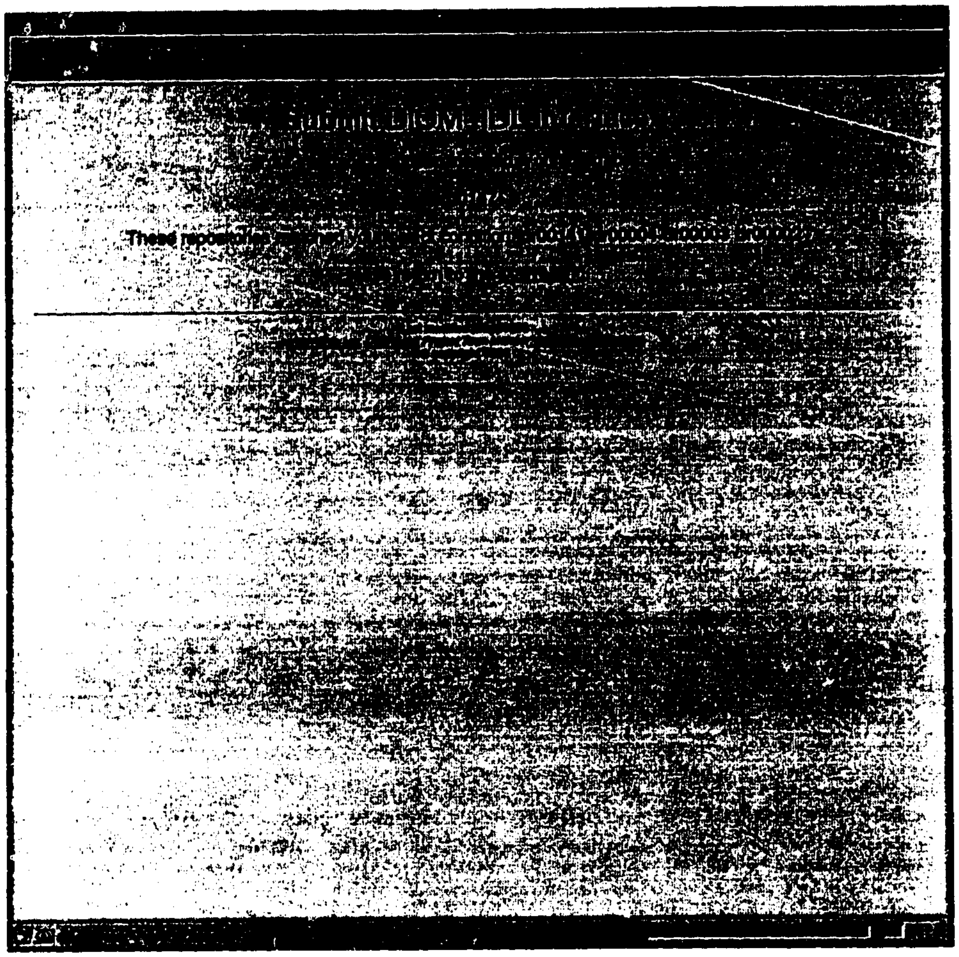


Figure 6.7: DIOM-IDL Interface Editor



Submit DIOM-IDL [illegible]

These repositories [illegible]

Figure 6.8: Interface is accepted

6.3.2 Consumer IDL Interface Browsing

Users may want to browse previously installed interface definitions to find useful interfaces, or to reuse interface descriptions in the composition of new interfaces.

The IDL browse interface (Figure 6.9) allows users to view all installed IDL definitions or use a filter to select a subset. The first option, *Show all DIOM-IDLs*, lists all interfaces descriptions in tabular format. The filter option allows users to browse the interfaces that match to specific fields. The browsing condition can be formed by using the OR/AND boolean operations. For example, to find all interfaces with Job in the keyword, the user would type Job in the keyword field of the filter and submit. The filter results are presented in a tabular format as shown in Figure 6.10. All fields used in the creation of the interface are displayed. The attributes of the interface can be displayed by entering the *interface ID* in the *View Attributes* input field. Interfaces can also be deleted by entering the *interface ID* in the *Delete Interface* input field.

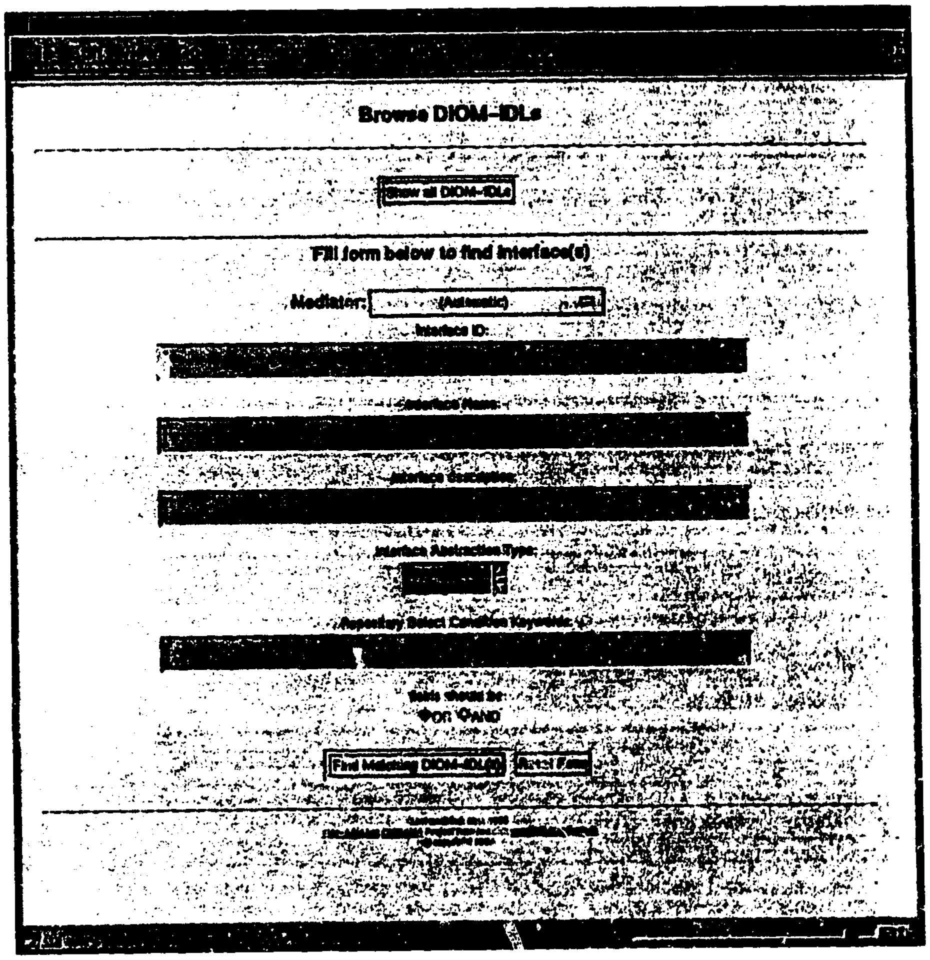
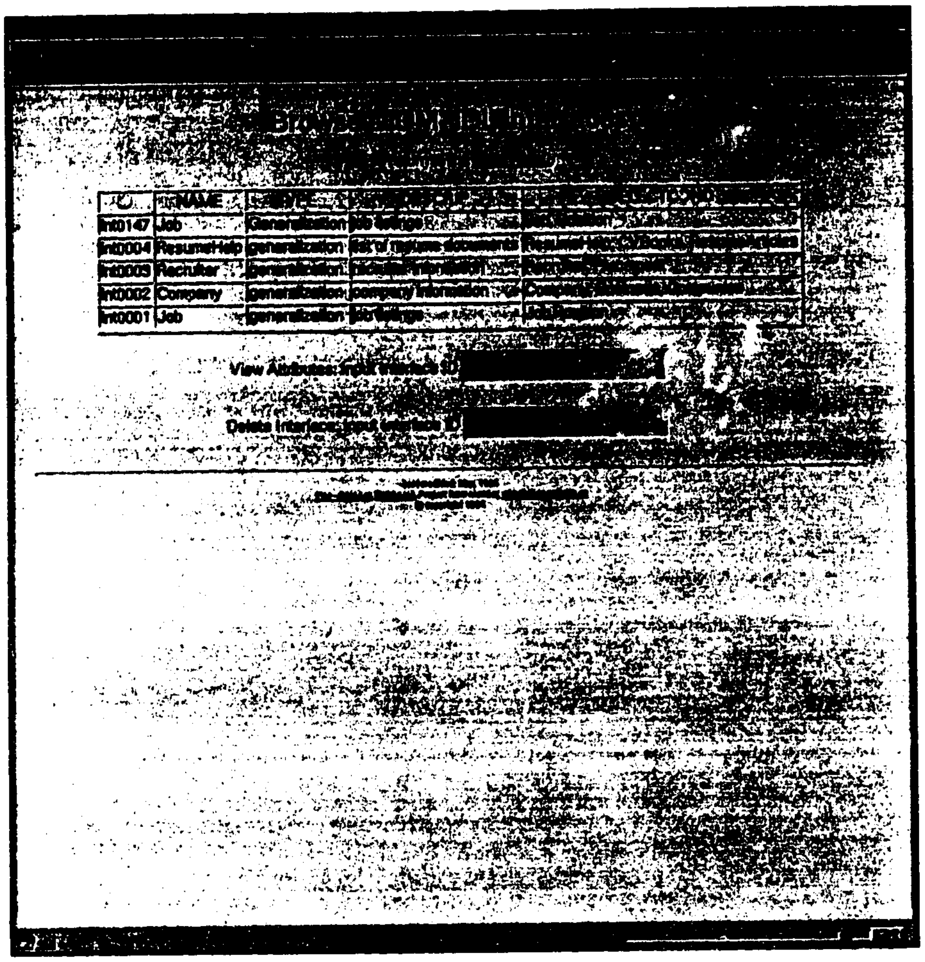


Figure 6.9: IDL Interface Browse Form



[illegible]	NAME	[illegible]	[illegible]	[illegible]
Int0147	Job	Generalization	job listings [illegible]	[illegible]
Int0004	ResumeHelp	generalization	list of resume documents	[illegible]
Int0003	Recruiter	generalization	[illegible] information	[illegible]
Int0002	Company	generalization	company information	Company [illegible]
Int0001	Job	generalization	job listings	[illegible]

View Attributes: input interface ID

Delete Interface: input interface ID

Figure 6.10: IDLs matching browse form are shown

6.4 Distributed Query Services

Querying multiple and heterogeneous information sources through the DIOM system requires a mapping of form-based user input to DIOM-IQL. Clients transform user input into DIOM-IQL syntax. This allows for flexibility in query interface designs based on the consumer requirements. The Rainbow system implementation offers two query interfaces, a simple keyword based query interface, and an expert SQL-like query interface.

6.4.1 Simple Query

This method of querying is intended for naive or irregular system users who are not interested in the design of mediators or are not familiar with the usage of SQL-like language. The simple query user provides keywords related to the information they seek. The choice of keyword is important in obtaining the desired information. The system will use these keywords to search from the available information sources and generate a return set of matching objects.

Figure 6.11 shows the query interface for a simple query. The *KEYWORDS* field is where the user enters their keywords.

The user also has the following options available: (1) choice of "AND" or "OR" boolean stringing of keywords, (2) specification of the number of resulting objects to be shown on each page, (3) the partial matching or exact matching of keywords, and (4) choice of how much information is to be presented in the result.

An *optimal* solution is generated by utilizing information brokers which support keyword based searching. For example, in processing a simple query on the keywords, `job` and `multimedia`, our system could query all information brokers registered which provide general keyword based search services such as **Yahoo** and `Harvest`.

6.4.2 Simple Query Tracing

Simple query processing of information sources is currently composed of a five step process:

1. source selection
2. for each source, translation of the simple query form data into a valid URL address to that source
3. parallel execution of each subquery against an individual source
4. translation of each subquery result into the consumer's preferred result representation
5. the merging of the subquery results from different sources into the final result format

The first step of source selection is done manually from the *Choose Repositories* selection box in Figure 6.11. To monitor the other four steps of simple query processing, the query is decomposed and presented in an interactive interface. Figure 6.12 shows the decomposed query components for the `multimedia` and `jobs` query. The top of the screen shows the query options chosen to answer the query: 1) the sources are **Yahoo**, `Harvest`, and `Lycos`, 2) the keywords are `multimedia` and `jobs`, 3) the maximum requested number of results are 30, 4) all records should contain both keywords (**and**), 5) and the query is based on partial matching keywords only.

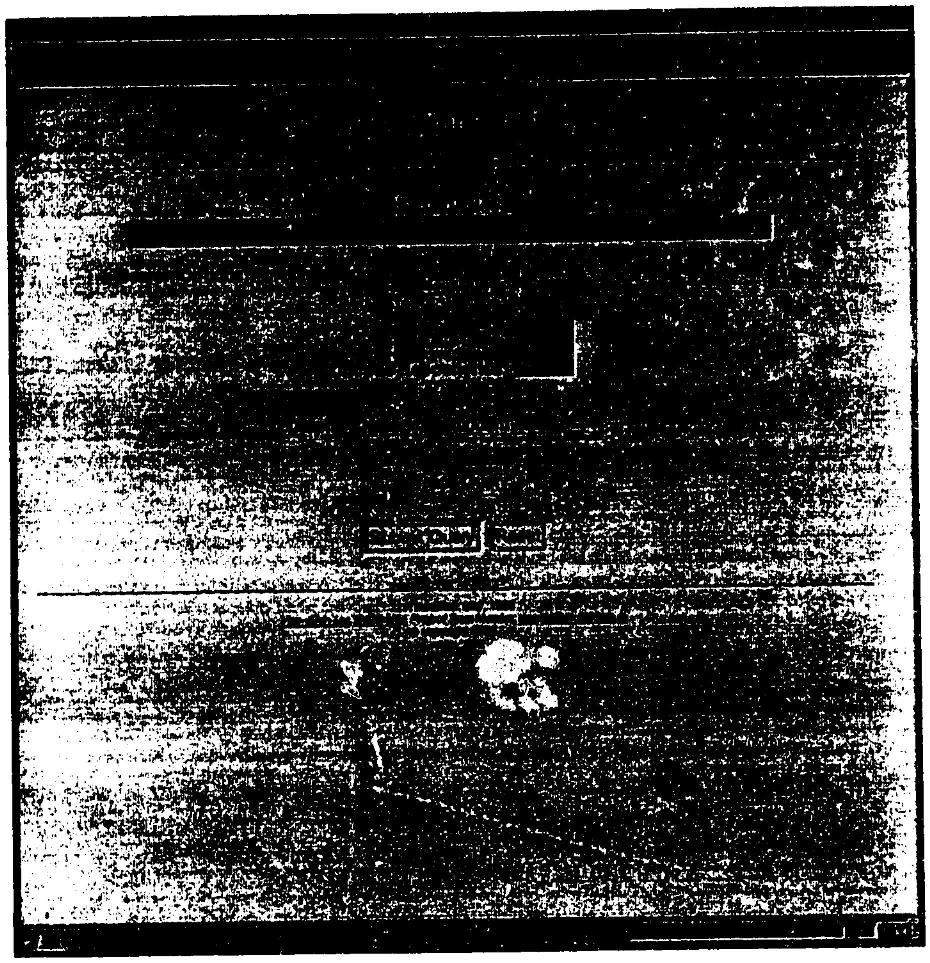
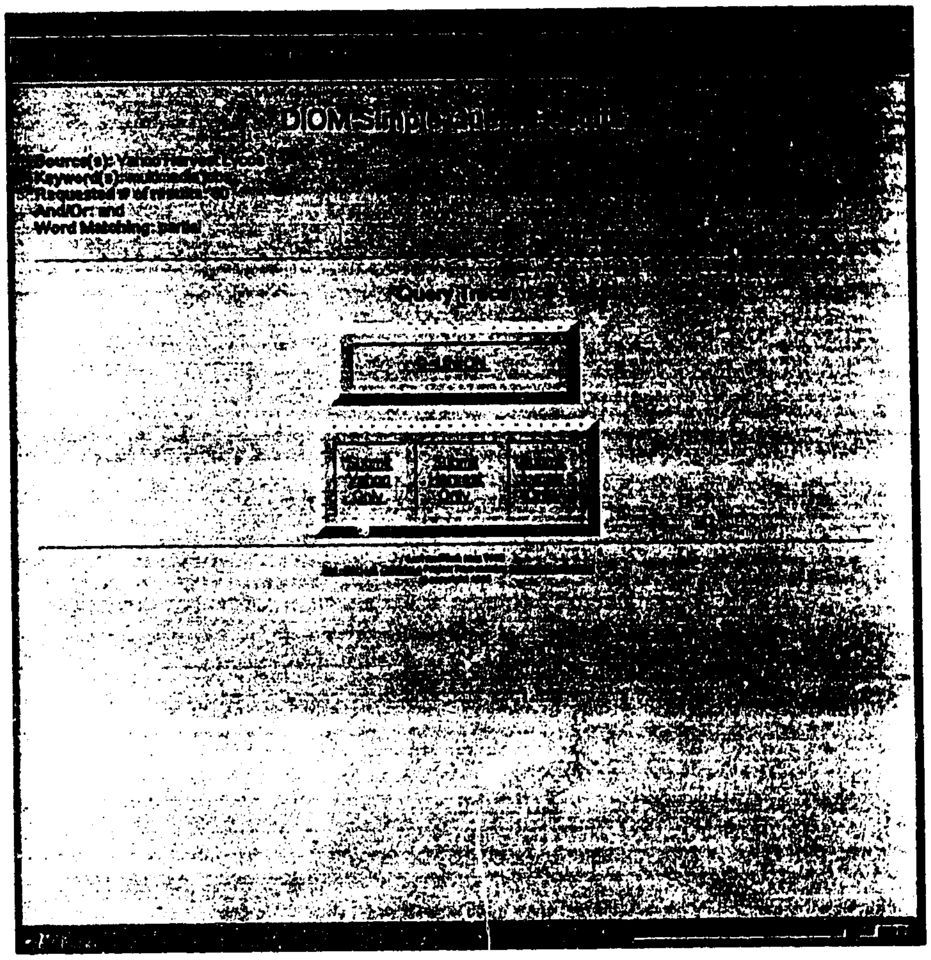


Figure 6.11: Simple query composition form



Source(s): Yahoo Harvest Lycos
Keyword(s): [illegible]
Requested # of results: 50
And/Or: and
Word Matching: partial

Figure 6.12: Dialogue for simple query tracing

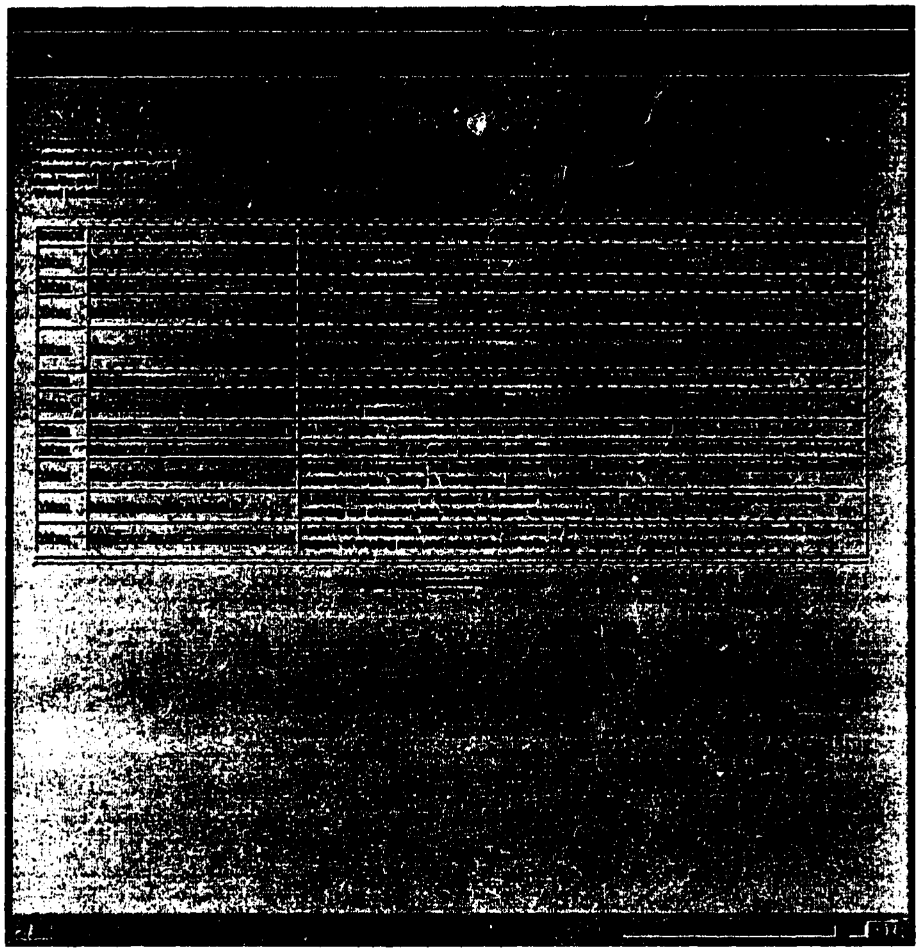


Figure 6.13: Results from Yahoo only

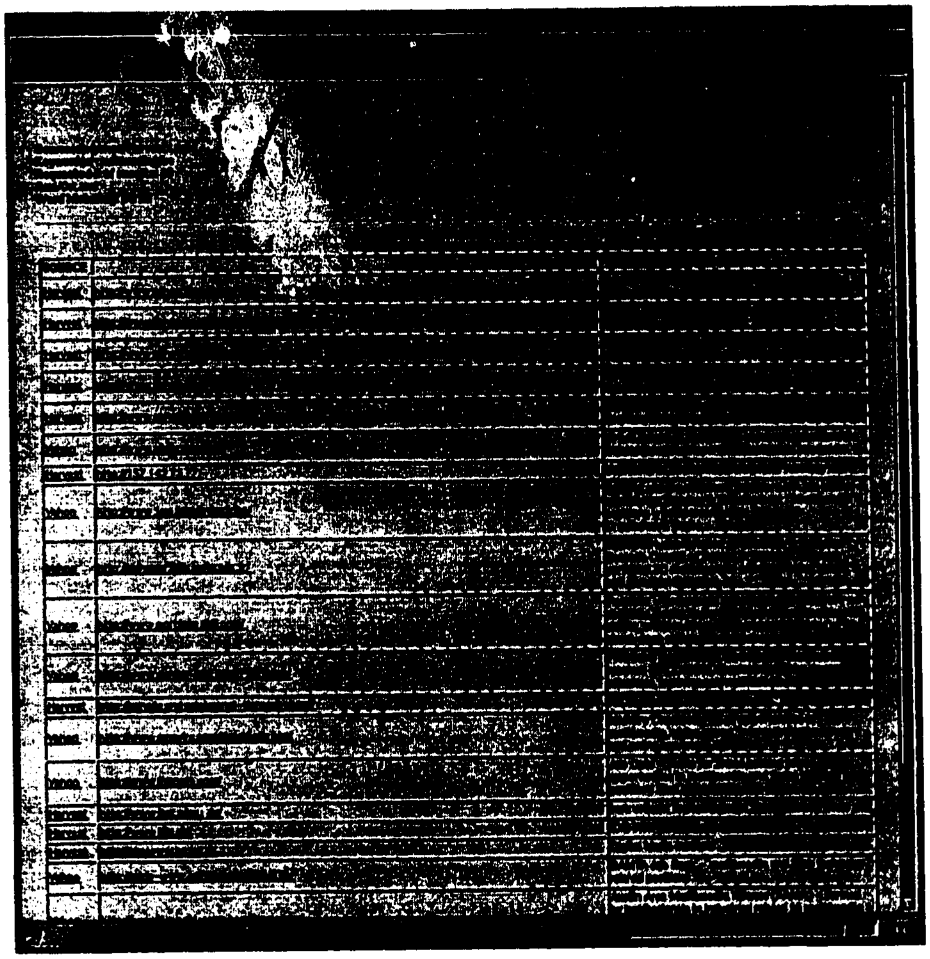


Figure 6.14: Results from all available sources

The middle of the screen shows the decomposed queries. Each leaf box represents a information source and the D-UNION box represents the final union of the results. Clicking on a box will execute the local query and return the result in normalized format. Figure 6.13 shows the results returned from Yahoo. The Yahoo subquery returns a list of pointers to its component information sources which contain the keywords. The results are displayed in a tabular format with three fields: (1) source of the record (which is only Yahoo in this case), (2) URL to actual information (the pointer received from Yahoo), and (3) description taken from actual information with the initial search keywords in bold.

Clicking on the *D-UNION* box from Figure 6.12 allows us to see the final result after assembling the results returned from each sub-query through the distributed union operator *D-UNION*. The *D-UNION* first conforms each subquery result into a common IDL format and then unions them. The final result of the initial example query is shown in Figure 6.14. The table contains the same field names as the Yahoo result page but in this case the *Source* values are alternating between `Yahoo` and `Harvest`.

6.4.3 Expert Query

For those users who are more experienced with the SQL style of querying, Rainbow provides an expert query tool based on DIOM-IQL. The expert query form is intended for more complex query formation, and expert queries can be saved (installed) and reused later. Queries are run over a set of interfaces defined through a mediator. They reference the interfaces and the attributes defined for those interfaces. The resultant object is represented in terms of an IDL specification derived from the query specification itself.

The DIOM-IQL form shown in Figure 6.15 closely follows the DIOM-IQL syntax description given in Section 4.1. The mediator field specifies which grouping of interfaces to use and can be left open to allow the system to generate the corresponding IDL interface. The *Target* field specifies the information repositories to be used or avoided in the query where repositories to be avoided are preceded by an '!'. The *Select* field allows the user to select only those fields wh h are of interest to her. The *From* field specifies the interfaces to use. The *Where* field c tains the query conditions to be matched. Join conditions between interfaces do not ha to be specified here as they will be automatically filled in from the IDL definitions (see Se ion 4.1.1). A full outer join will return all records which were relevant to the query. The r ults can be grouped and ordered by filling in the appropriate conditions in the *group* and *o r* fields respectively. Expert queries can be traced by using the *trace* checkbox. The query can then be installed and be reused later on. Eventually, an extension to expert query installation is the development of a toolkit for supporting continuous (active) queries [33].

Figure 6.15 shows the query form filled out for an example query. The *Mediator* field is the `Job Search Assistant`, so the system will use the interfaces found from that mediator. The *Target* field is left as '*', so all repositories relevant to the interfaces will be used to fulfill the query. The *Select* condition is also left as '*', so all fields will be displayed. The *From* field displays `Job` and `Company`, so these interface descriptions (defined in Chapter 3) will be used in answering the query.

6.4.4 Expert Query Tracing

To fine tune and monitor the steps of querying in the DIOM environment we provide the capability for query tracing through a graphical interface. Each step of the query process

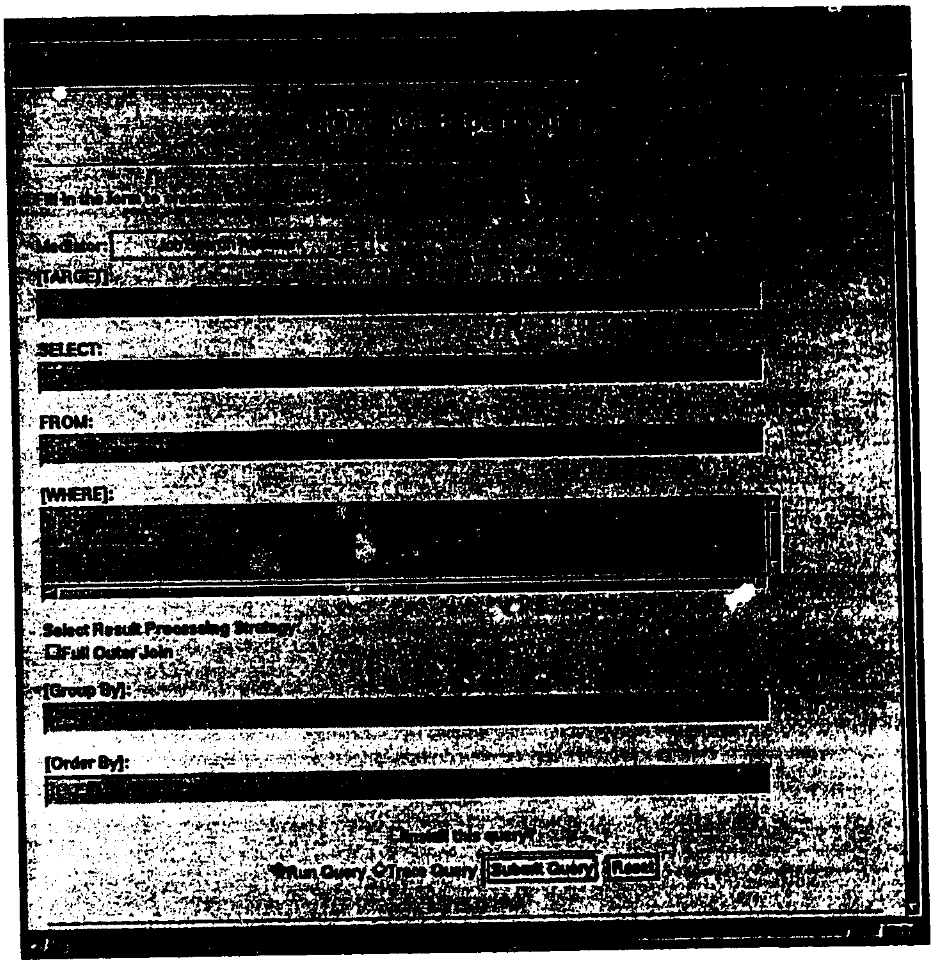


Figure 6.15: Expert query composition form

can be monitored by the user. The DIOM query mediation process consists of the following steps:

1. **Query Routing**
   This is done by mapping the domain model terminology to the source model terminology, by eliminating null queries, which return empty results, and by transforming ambiguous queries into semantic-clean queries.

2. **Query decomposition**
   This is done by decomposing a query expressed in terms of the domain model into a collection of queries, each expressed in terms of a single source model. Both straightforward query decomposition and complex query decomposition are considered.

3. **Parallel query plan generation**
   The goal of generating a parallel access plan for the groups of subqueries is to obtain the maximum parallelism and the best quality of cooperation in searching for query answers from multiple information sources. A parallel query plan is constructed for the modified query resulting from query decomposition. The plan is composed of single-source queries (each posed against exactly one local export schema), move operations that ship results of the single-source queries between sites, and the post processing queries that assemble the results of the single-source queries in terms of the information consumer's query request expressions.

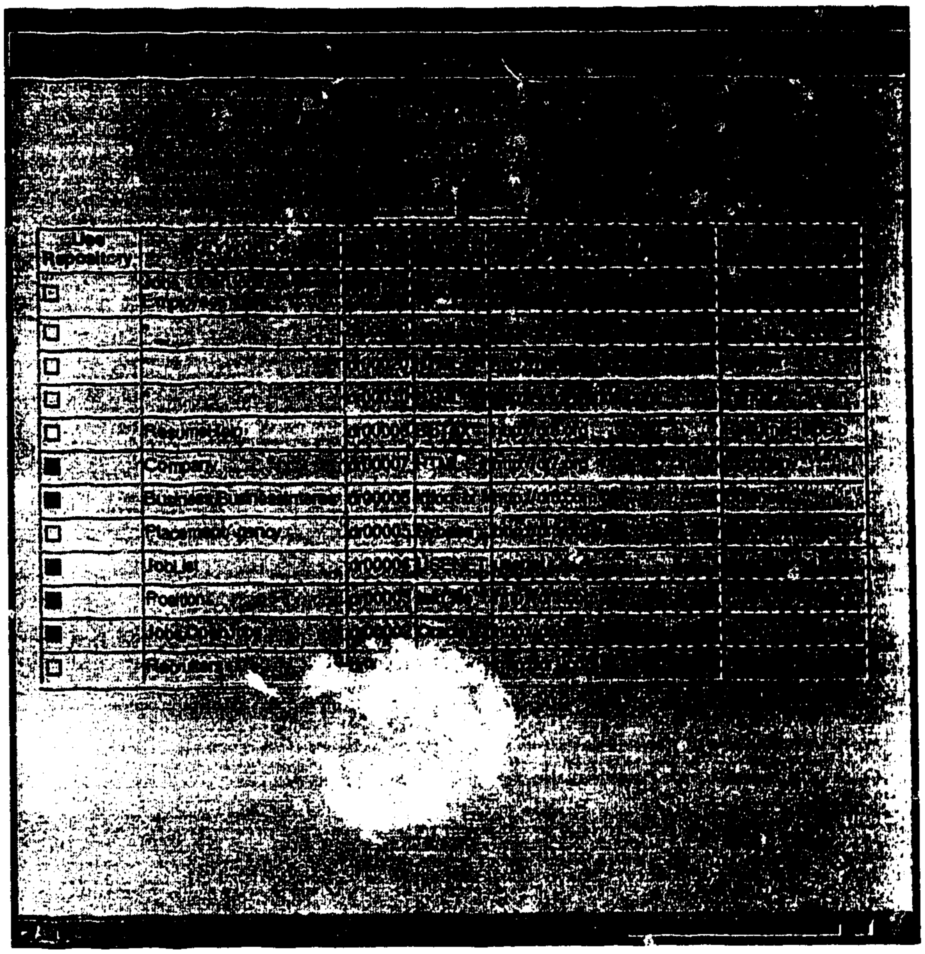
4. **Subquery translation and execution**
   The translation process basically converts each subquery expressed in terms of an IDL source model into the corresponding information producer's query language expression, and adds the necessary join conditions required by the information source system.

5. **Query result assembly**
   The result assembly process involves resolving the semantic variations among the subquery results. The annotation/semantic attachment is one of the main techniques that we use for resolving semantics heterogeneity implied in the query results.

Tracing the query is designed to show the user which repositories are being selected in the process and how the query is actually processed at each site. This can be a useful option perhaps when debugging or determining the role of a repository in answering a query. The query trace option is provided at the bottom of the expert query composition form in Figure 6.15.

The query given in Figure 6.15 is executed with the query trace option. The first step in query tracing is query routing and is displayed in Figure 6.16 which gives a list of all repositories and the repositories selected for answering this query. The repositories with the selected checkbox buttons are the repositories selected as for the Job and Company interfaces and are the repositories that will actually be queried. The user can tailor this list by adding or removing repositories to the query.

Following the query routing step is the query decomposition and parallel query plan generation steps. Figure 6.17 shows the query tree generated from the initial expert query. Each leaf node represents the result from a single repository and the intermediate nodes represent the result of using a relational operator on the results from the underlying connected



Use Repository	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
☐	Job Employer[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
☐	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
☐	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
☐	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
☐	Resume[illegible]	r00004	[illegible]	[illegible]	[illegible]
■	Company	r00007	[illegible]	[illegible]	[illegible]
■	Business[illegible]	r00008	[illegible]	[illegible]	[illegible]
☐	Placement agency	r00005	[illegible]	[illegible]	[illegible]
■	JobList	r00006	USENET	[illegible]	[illegible]
■	Position[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
■	Job[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
☐	Rep[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Figure 6.16: Query Processing: Repository selection

nodes. Clicking on each leaf node will display the subquery that is to be executed at that repository and the result obtained. Clicking on the relational operators will give the results of the operation over the data assembled from the lower connected nodes. Figure 6.18 shows an example subquery result returned from repository DR3.

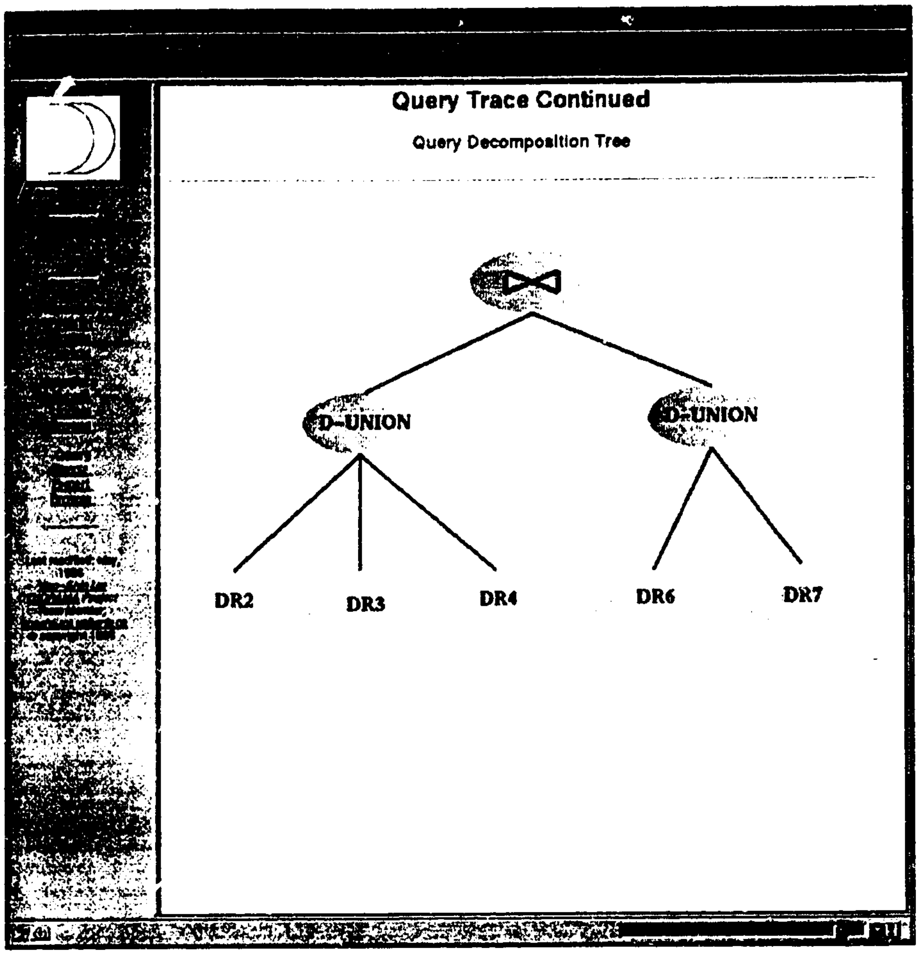


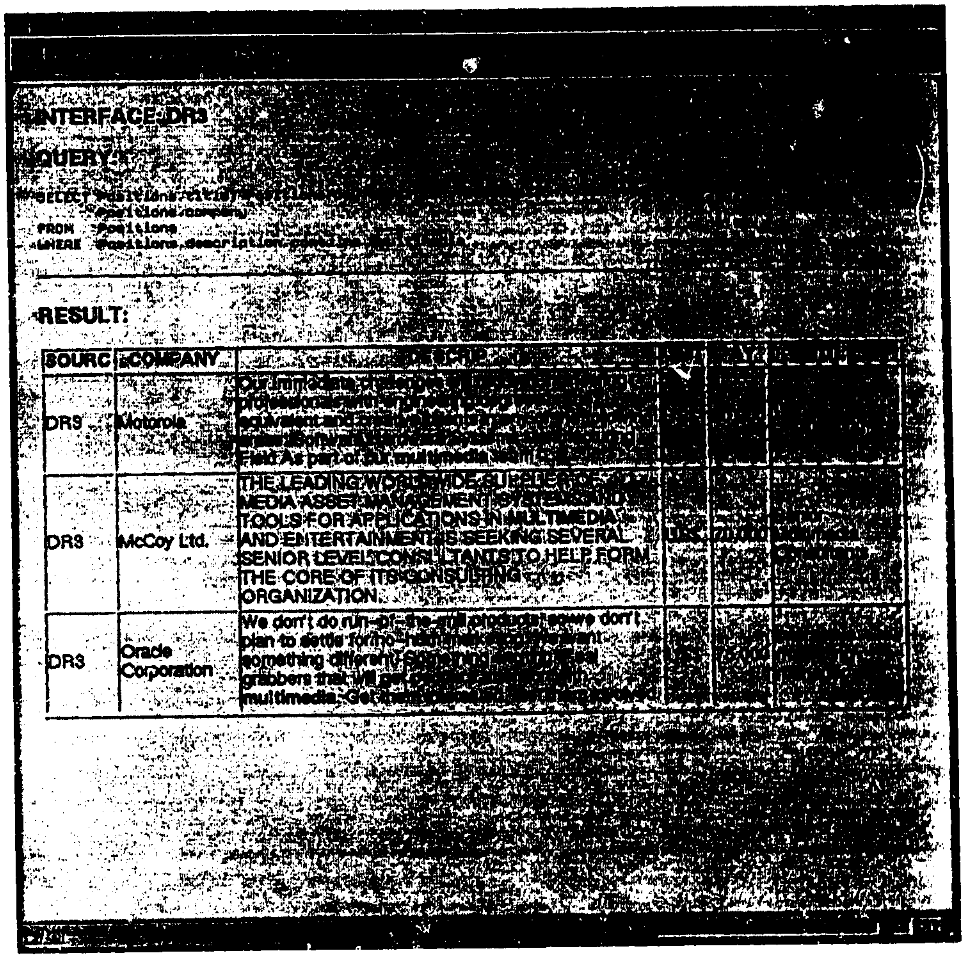
Figure 6.17: Query Processing: Query tree with clickable nodes

6.4.5 Query Result Assembly

Once all the subqueries are processed the results are returned from the wrappers and assembled according to the original IQL query expression. Figure 6.19 shows the result of the initial query from Figure 6.15 in tabular format.

6.4.6 Installed Queries

Similar to the browsing of metadata of information sources, our system allows for browsing over installed queries. This browsing capability results in a display of all installed queries or the installed queries satisfying a particular filtering condition as shown in Figure 6.20. The filter is designed to find queries by specifying filtering conditions on the keywords



INTERFACE:DR3

QUERY:

```
SELECT @positions.title [illegible]
       @positions.company
FROM   @positions
WHERE  @positions.description contains [illegible]
```

RESULT:

SOURC	COMPANY	DESCRIPTION	[illegible]	[illegible]	[illegible]
DR3	Motorola	Our immediate challenge [illegible] professional [illegible] equivalent and [illegible] field. As part of our multimedia team [illegible]	[illegible]	[illegible]	[illegible]
DR3	McCoy Ltd.	THE LEADING WORLDWIDE SUPPLIER OF [illegible] MEDIA ASSET MANAGEMENT SYSTEMS AND TOOLS FOR APPLICATIONS IN MULTIMEDIA AND ENTERTAINMENT IS SEEKING SEVERAL SENIOR LEVEL CONSULTANTS TO HELP FORM THE CORE OF ITS CONSULTING [illegible] ORGANIZATION.	US$	70,000	[illegible]
DR3	Oracle Corporation	We don't do run-of-the-mill products so we don't plan to settle for [illegible] something different. Some [illegible] grabbers that will [illegible] multimedia. Get [illegible]	[illegible]	[illegible]	[illegible]

Figure 6.18: Decomposed query for DR3

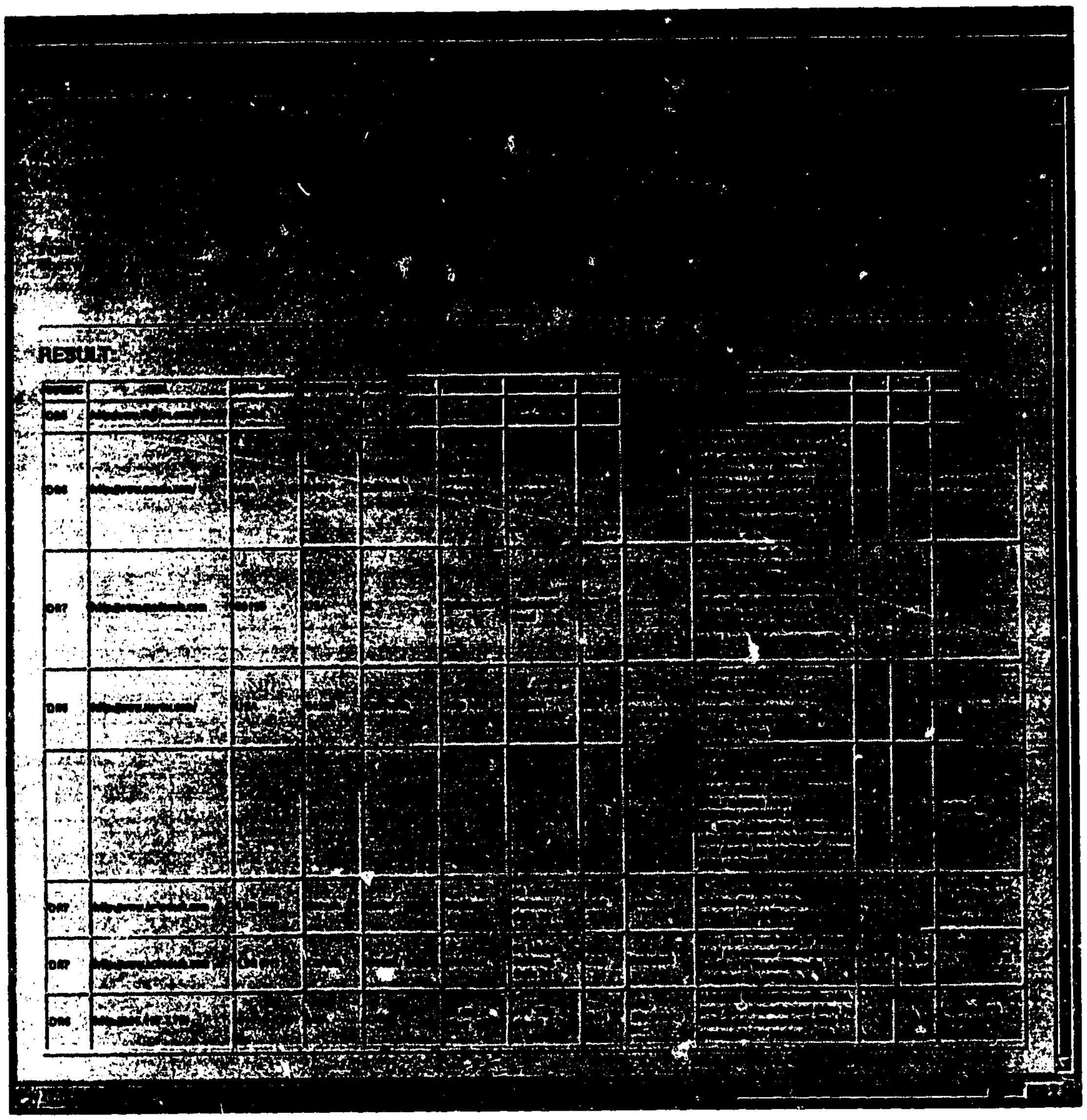


Figure 6.19: Final query result page

edited or executed.

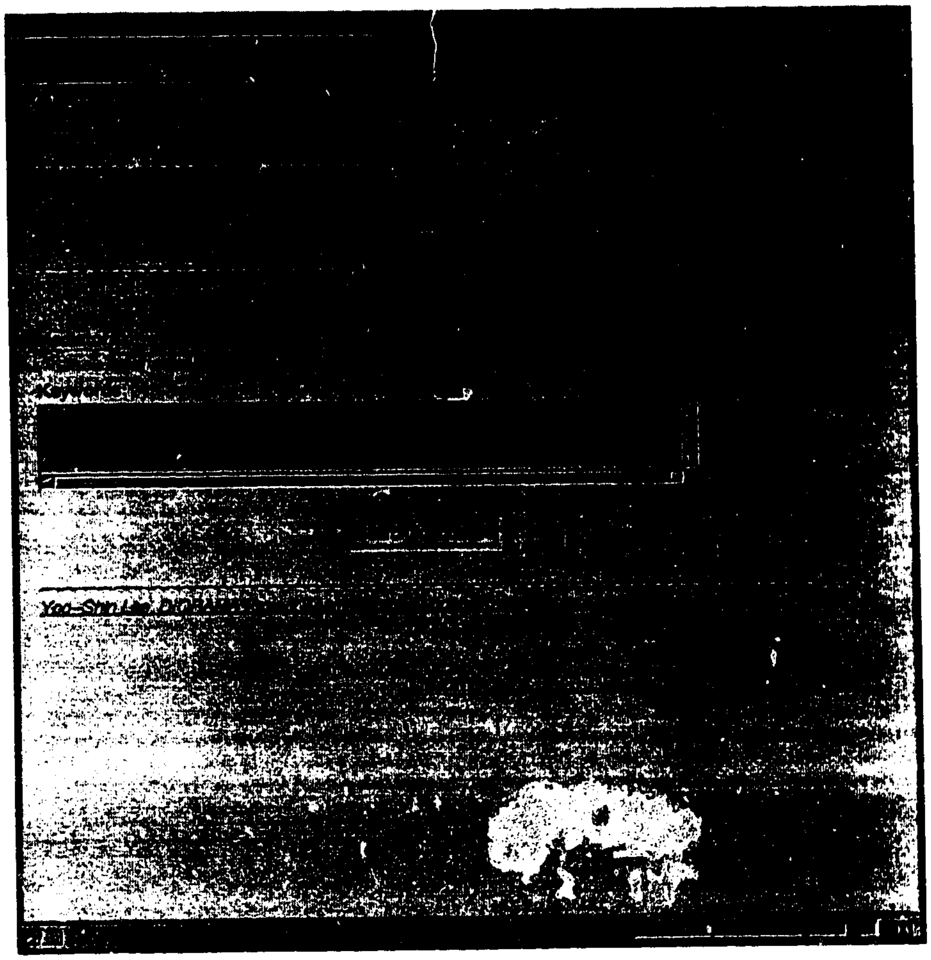


Figure 6.20: Form to query metadata on installed queries

6.5 Implementation Issues

The Rainbow user interface is made up of two components: (1) the HTML markup language (including frames) which the Netscape browser actually interprets and displays, and (2) the underlying Perl CGI modules which process the data from the scripts and return new web pages to display. The client machine running the Netscape browser does virtually no processing other than to interpret and display the HTML markup language. The server machine which processes the form user data does most of the processing by activating the scripts which make up the RSM. The Perl scripts do such things as interpret the form data, access other Rainbow modules (e.g., wrappers), access the Oracle engine (through Oraperl), and transform and normalize data. After all server-side processing is completed, the CGI

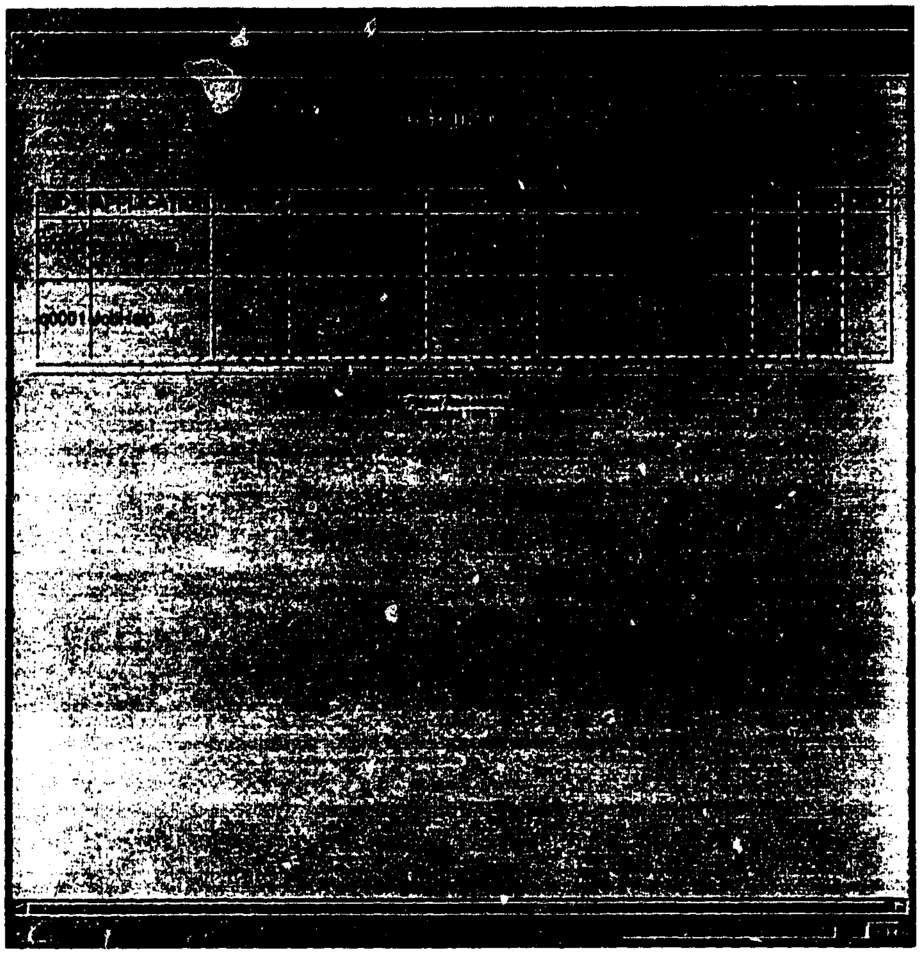


Figure 6.21: Result of metadata search on installed queries

scripts return valid HTML code to the client side browser for display.

An HTML interface has been designed and implemented. Future implementations include the creation of a Java [4] applet for Rainbow. This will make the user interfaces richer in functionality and more dynamic to user input. Due to the popularity and interest of WWW development and WWW tools, the following offers a brief comparison between dynamic user interfaces based solely on standard CGI and interfaces based on Java applets:

Common Gateway Interface

The Common Gateway Interface (CGI) is a standard for interfacing external applications with information servers, such as HTTP or Web servers. A plain HTML document that the Web daemon retrieves is static, which means it exists in a constant state: a text file that doesn't change. A CGI program, on the other hand is executed in real-time, so that it can output dynamic information.

For example, let us say that you wanted to "hook up" your Unix database to the World Wide Web, to allow people from all over the world to query it. Basically, you need to create a CGI program that the Web daemon will execute to transmit information to the database engine, and receive the results back again and display them to the client. This is an example of a gateway, and this is where CGI got its origins.

The database example is a simple idea, but most of the time rather difficult to implement. [illegible]ally is no limit as to what you can hook up to the Web. The only thing you need to remember is that whatever your CGI program does, it should not take too long to process.

Java

Java is a programming language for distributed applications. Java lets you add both the content and the code necessary to interact with your content. You no longer need to wait for the next release of a browser that supports your preferred image format or special game protocol. With Java you send browsers both the content and the program necessary to view this content at the same time.

Previously you had to wait for all the companies that make the web browsers your readers use to update their browsers before you could use a new content type. Then you had to hope that all your readers actually did update their browsers. Java compatibility is a feature that any browser can implement and by so doing implement every feature.

For instance let us say you want to edit tables using your web browser. Previously you would have to wait for a new browser which supports table-editing functions. Now you can write your own code to view and edit the tables and send it to any client which requests the table.

CGI scripts operate on the server side of the interface. They are programs which return dynamic HTML code, but this code is still simple HTML. Web browsers simply display HTML and allow very little else, so the functionality is limited. The use of *frames* allows the user interface to be divided into multiple windows, but this mechanism is still very primitive and is not very flexible. Java applets on the other hand, operate on the client side of the interface (Netscape runs a Java virtual machine). Java gives the interface programmer the ability to have full-fledged user interfaces (dialogues, separate windows, sound, graphics, widgets, etc.).

The DIOM project will slowly migrate to a Java-based user interface, but because of time constraints and the late availability of the Java development kit, a Java version of the interface was not available for this thesis.

6.5.1 Interface Enhancements

Here are a list of user interface enh .ncements which will increase system functionality. Some of these enhancements have only became feasible with the recent emergence of the Java programming language, the Java programmer development tools, and Netscape support.

Consumer Domain Model Editor
The implementation of a consumer domain model editor is currently in progress. The future development of the IDL interface will include a CUI to create diagrams such as Figure 3.1 and to be able to edit these diagrams with semantic information for the nodes and the links.

Source Registration
It can be argued that some of fields in the producer registration form (Figure 6.2) should be automatically retrieved through the wrapper. To make it easier on the registrant the only thing they would be required to fill in would be the wrapper URL. The system could access the wrapper initially given this URL. The repository name, source type, owner, and first level of keywords could then be supplied by the wrapper. The export schema name could be given as a set of choices. This option will be taken into consideration in the next implementation release.

Repository Browsing
For the repository browsing interface shown in Figure 6.5 and Figure 6.6, further improvement includes the ability to browse the actual repository data by mouse-clicking on the repositories found. The interface would then be able to directly access and edit tables from the producer repositories.

Simple Query
Future implementations may include more intelligent methods for determining the semantics of a simple query. For example, the system may ask the user to provide more semantic information on the keywords (via popup dialogue windows) such as whether `multimedia` is part of a title or description. The system may then automatically map the simple query to an IQL expert query.

Expert Query
Future development of this interface will require more automated query composition. For example, choosing the Job Search Assistant mediator would trigger the interface to give the user specific choices on which IDL interfaces and which attributes are available.

Query Tracing
The creation of the query processing tree is simulated. Using Java, it would be possible to create a tree dynamically. The tree would also have clickable nodes and offer the same functionality as in the Rainbow prototype.

Chapter 7

Wrapper Design and Prototype Implementation

Wrappers are software modules that are responsible for connecting individual information sources to external users. In the DIOM system, a wrapper receives and translates instructions written in the DIOM-IQL language into executable local repository commands as shown in Figure 7.1. The result of the repository is then packaged by the local query result packaging module and returned to the client. A sketch of our wrapper architecture was shown previously in Figure 5.4. A generic wrapper architecture is useful for the automated construction of wrappers to different types of data repositories. A generic wrapper architecture allows for uniform access to the wrapper and a uniform return result format. Wrappers which are built with the specified generic architecture may also reuse (by subtyping) much of the higher level code with only certain local functions having to be reimplemented.

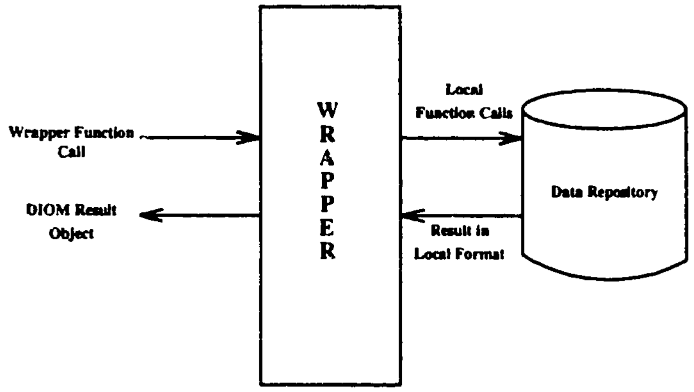


Figure 7.1: Wrapper translation of request and results

The generic function suite for the wrapper design is grouped into query, update, and wrapper metadata functions. Each of these functions is in turn composed of lower level local functions. Wrapper function requests are translated by the wrapper into the local command equivalent and the result is returned as a DIOM result object.

Using the DIOM specified wrapper interface, wrapper creation for new types of information repositories may be partially automated. Thus, only those methods local to the repository need to be re-implemented. In order to support semi-automatic generation of wrappers, the wrapper software module consists of a two-tier design: the generic wrapper function suite and the source-dependent wrapper function suite. All wrappers support the generic function suite which is then mapped to the source-dependent functions. The primary difference between the various DIOM wrappers (e.g., wrapper to Oracle vs. wrapper to Lycos) lies in the connection of the generic wrapper functions to the source-dependent functions and the amount of processing that a wrapper does compared to the local source. These wrapper functions work together to accomplish the primary tasks:

- Transform a wrapper service request into a local command which is executable at the corresponding source repository.
- To connect the execution results into DIOM objects and return the result to the caller.

7.1 Wrapper Network Access

The wrappers are navigated and accessed across the network via the HTTP Internet protocol. This protocol relieves the system of network navigation and access conflicts, and thus offers a solution to the heterogeneity of network access protocols. The DIOM wrapper services are invoked by opening the correctly formed URL [6] to the wrapper. The wrapper is currently implemented as a CGI [1] [41] script, which a is common standard for external gateway programs to interface with information servers. Wrappers are implemented using Perl, the widely ported programming language. The external wrapper interface (URL address) is identical in format for all wrappers.

7.1.1 Syntax of the Wrapper Interface Language

The wrapper interface language allows external clients to request wrapper services and queries in a format that is understandable by the wrapper manager. Therefore, the RSM is not the only possible client of the wrapper, as external developers can create applications directly based on our wrappers. Currently, wrapper invocation expressed in the wrapper interface language format is in the form of a URL to the wrapper which can then be used to invoke the wrapper. A BNF-like description for a HTTP URL looks like this:

```
httpurl  = "http://" hostport [ "/" hpath [ "?" search ]]
```

The *hostport* is the Internet hostname of the machine where the wrapper is located, the *hpath* is the unix-like filesystem path to the wrapper script, and the *search* field is where the wrapper service commands and data are embedded.

In compliance with HTTP protocol, all spaces must be replaced by a '+', and all variable names and values in the *search* field must have their special characters encoded with their hexadecimal equivalent prefixed with a '%'. The special characters which must be encoded are:

```
$ < > ‘‘ ’’ # % { } | ^ ~ [ ] ; \ ? : @ = & $
```

For example, the query:

```
query=select * where symbols=@?#
```

[1]Common Gateway Interface Specification version 1.1

must be encoded as:

```
query=select+*+where+binarycode=%40%3F%23
```

The wrapper services include: *insert*, *delete*, *select*, *createclass*, *destroyclass*, *getclassinfo*, *getexportschema*, and reqvariablesinfo. Each service call maps directly to one of the generic functions in the generic function suite (See Section 7.2 for a detailed description). A fragment of our wrapper interface language syntax is provided as follows:

```
wrapperSyntax ::= http://<wrapper path>?<DIOM Request>

<DIOM Request> ::= service=<service type>&<variable list>

<service type> ::= insert | delete | select |
                   createclass | destroyclass | getclassinfo |
                   getexportschema | reqvariablesinfo

<variable list> ::= <variablename> | <separator> | <variable list>

<variablename>  ::= attlist=<attname1> <attname2>.. |
                    classlist=<classname1> <classname2>.  |
                    condition='<conditionstring>' |
                    value=<name1> '<valuestring1>'
                         <name2> '<valuestring2>'...
<separator>     ::= '|'
```

Here is an example of using the wrapper interface syntax to find *"all jobs offered by IBM"* using one repository and one wrapper (assume the Rainbow Services Manager has targeted this repository as one of the sources to answer the query). The decomposed subquery before begin encoded into the wrapper interface language is:

```
select * from jobs where company_name="IBM"
```

The DIOM-IQL query is then encoded into the following wrapper interface syntax which is generated by the Rainbow Services Manager (note: url is an actual continous string):

```
http://www.test.ca/~diom/wrapper/service.cgi?
service=select&attlist=*&classlist=Job&condition='company_name="IBM"'
```

The wrapper service call is a URL which points to the network address of the repository, the wrapper CGI-Script, and contains the encoded DIOM-IQL query plus any extra variables. Each of the components of the select query have been broken up into the constituent parts: the requested service is a **select**, the attributes to return are all ('*'), the class to query is **Job**, and the query condition is **company_name=''IBM''**.

7.1.2 Wrapper Result Format

The wrapper returns the result from wrapper service calls as a MIME [2] multi-part object [8, 40] which is already a part of the HTTP 1.0 specifications. The current implementation of the wrapper returns a simple text MIME type of the query result in a standard table format.

[2]MIME allows return objects to contain both text and non-textual data such as digitalized audio, images, and video

7.2 Generic Wrapper Function Suite

The generic wrapper functions are classified into three categories: IQL object queries, local update requests, and DIOM directory service requests. The IQL object query allows for the selection and return of local objects in an IQL query fashion. The local update requests allow for updates to the actual local object. The directory service requests allow system programmers to access metadata information on the local wrapper.

Defining generic wrapper functions allows for the creation of wrappers to various types of data repositories. The classification of our wrapper functions closely fits with functions in more structured data repositories such as RDBMs, and OODBMs. For semi-structured data sources such as BibTeX files or HTML documents, more wrapper processing is required to retrieve useful data. Some highly unstructured data sources such as technical reports or journal articles need text-retrieval type or content-based processing. Furthermore, unstructured data need more complicated heuristics to retrieve useful information. It is important to consider the granularity of information wrapping. For example, when designing wrappers to duplicated data such as Usenet data, there are two methods: (1) if each server is considered a information source than wrappers must be implemented for each news server, (2) a finer granularity might be achieved by considering each news group as an information source. In this case, a wrapper may use more than one news server to fulfill a request. The Rainbow implementation opts for the first design where each server is seen as an information source. Each newsgroup is then viewed as an object containing smaller article objects and articles would contain attributes: subject, post_date, organization. Alternatively, the second method may retrieve more useful information from the articles as some newsgroups have highly structured articles. For example, the `alt.jobs` news group has a defined structure of the subject heading which specifies the additional fields: job_location and job_title.

Each wrapper function is implemented as a Perl function. The generic wrapper function suite designed in Rainbow is provided here:

1. IQL Object Query

   &diom_wrapper_Select (attributenamelist, condition)
   This function returns a DIOM object with the instances matching the selection. SQL equivalent:

   Select *attributenames* from *class* ,[*class*]... where *condition*

2. Local Update Requests

   &diom_wrapper_Delete (classname, condition)
   Deletes selected instances from class. Deletes actual local data. SQL equivalent:

   Delete from *classname* where *condition*

   &diom_wrapper_Insert (classname, name-value-pair-list)
   Inserts new instances into classname. Inserts data into local repository. SQL equivalent:

   Insert into *classname* (*name1*, *name2*, ...) *values* [(*value1*[,*value2*]...)]

&diom_wrapper_CreateClass (string classname, attributespeclist)
This creates new classes given the attribute specifications. Creates new data in information source which is equivalent to the class definition.

Create Class *classname attribute spec [,attribute spec...]*

&diom_wrapper_DestroyClass (classname)]
This destroys classes. Destroys the class equivalent in the local information source.

Destroy Class *classname*

3. DIOM Directory Service Requests

&diom_wrapper_GetClassInfo
retrieves class information on all classes.

&diom_wrapper_GetExportSchema
retrieves export schema [3]

&diom_wrapper_ReqVariablesInfo
retrieves variable names needed to access repository

7.3 Source-Specific Function Suite

Each of the generic functions is in turn composed of local function calls which abstract the details of the local source. The functions have been designed in this two-tier manner to allow for easier composibility of wrappers to new source types. The existing generic interface functions can be transferred to new wrappers and the source-specific functions are the only pieces of code which need be altered. Here is the source-specific function suite:

&local_ExternalFetch Returns data from the local repository in its native data format.

&local_ExternalNormalize Takes data in native format and normalizes into an internal format (Such as the Oracle DBMS).

&local_Select Performs a select-from-where on the normalized data. This code may be reused for common normalization internal data formats.

&local_Delete Performs a deletion of data using the local language.

&local_Insert Performs an insertion of data using the local language.

&local_ResultsToDiom Translates the result from a selection into a DIOM-result.

&local_CreateClass Creates a new class (where a local class-equivalent is pre-defined) using the local language.

[3]A view of the local source that is accessible to the external users or external systems

&local_DestroyClass Destroy the equivalent of a class using local language.

&local_GetClassInfo Returns raw data from data repository which contains the class information.

&local_InfoNormalize Translates raw data on class information into an internal form such as a relation in Oracle.

&local_GetExportSchema Returns raw data from data repository needed to create an export schema.

&local_ExportNormalize Translates raw data from GetExportSchema to an internal format.

&local_ReqVariablesInfo Returns a text list of all variable names, its type, and whether it is required or not.

&local_AddToResult Adds objects to the DIOM-Result.

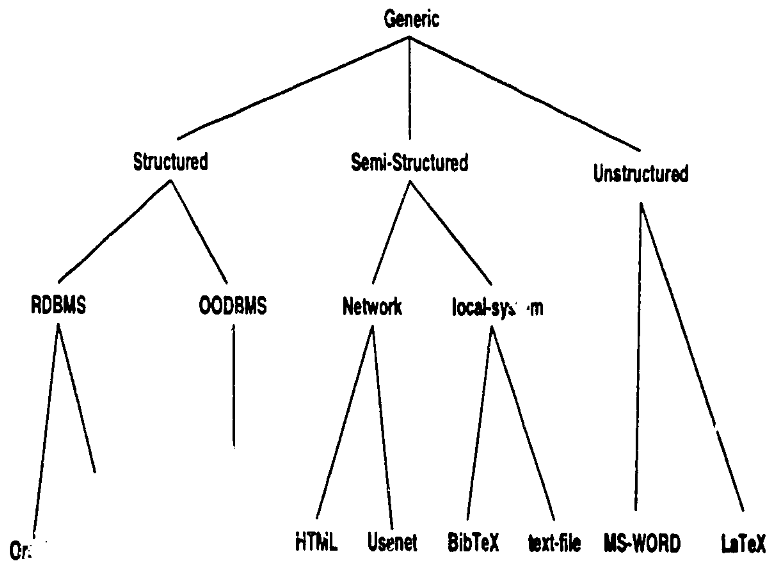
7.4 Hierarchy of Wrapper Types and Examples

The numerous types of repositories and information systems have resulted in the need for a subtyping scheme to group and abstract the repository types. Once repository types are classified and subtyped according to our repository subtype tree, the wrapper generation module is able to produce the correct wrapper skeleton for automatic generation of new wrappers. This subtyping based classification allows much of the wrapper code from previous implementations to be reused.

Figure 7.2 displays the subtyping tree for the wrapper types. The generic wrapper, at the top of the tree, contains the main interface for the generic wrapper services. The first level of subtyping of the generic wrapper is into structured, semi-structured, and unstructured types. This initial subtype of the generic wrapper reflects the processing power available at the repositories. For example, the structured repositories (e.g., Oracle) have powerful data processing capabilities and most likely have clean gateways to their DBMSs. Implementing the generic and local function suites for these repositories should require relatively little effort. For comparison, Semi-structured information repositories typically do not have any local data processing facilities, so the wrapper must perform the data processing. The semi-structured repositories can also be broken up into network information tools and local files. Network information tools will typically require a network access method (e.g., HTML pages requires HTTP access) while local files (e.g., BibTeX files) reside on the same physical machine as the wrapper. Both of these file types require processing by the wrapper to extract information and do basic data manipulation operations. Finally, the last subtype are the unstructured file types such as LaTeX documents which have little set structure. There may be specialized ways to extract information (information from figures, tables, lists) from these sorts of documents, but producing wrappers to these types of data will require the most effort.

7.4.1 Wrapper to Structured Information Sources

As mentioned earlier, the wrapper interface specification is based on common functions found in most database systems. The generic wrapper suite functions to RDBMSs such as



Wrapper subtype tree

Oracle are directly mapped to local Oracle commands. Many commercial database systems provide some external gateways to their products. Most DBMSs have either C or C++ interface libraries. Extremely popular DBMSs such as Oracle [44] also have extensions to popular languages such as Oraperl [5] and PL/SQL [45] as external gateways to their systems.

Wrappers for RDBMSs must translate the DIOM-request to a local gateway command. The generic interface commands: *Delete*, *Insert*, and *Select* require little change, as all RDBMSs have the support for such basic data manipulation. The *Create Class* and *Destroy Class* definition commands are translated into create table and destroy table equivalents. Each relation is interpreted as a separate class and each record in the relation is interpreted as an instance of that class. The metadata queries *diom_wrapper_GetClassInfo*, *diom_wrapper_GetExportSchema*, and *diom_wrapper_GetExportSchema* in a RDBMS return a list of all available tables, return the database export schema as defined by the local DBMS administrator, and return a list of the variables needed to access the export database, respectively.

Oracle

Oracle is a Relational DBMS which is accessed using the Oraperl [4] [5] gateway. Our first relational DBMS wrapper is implemented for the Oracle RDBMS. The data manipulation commands are directly translated into Oracle commands using the *ora_do* Oraperl function call. The data definition commands are slightly altered to *Create Table* and *Drop Table* the Oracle equivalents to *Create Class* and *Destroy Class*. The metadata query, *diom_wrapper_GetClassInfo*, can be translated into the IQL query:

```
select table_name from user_tables
```

The above Oracle SQL returns a list of the tables available. The metadata query, *diom_wrapper_GetExportSchema*, is generated by running an Oracle script or by retrieving a flat file that the DBMS Administrator has created manually.

7.4.2 Wrapper to Semi-Structured Information Sources

The complexity and large scale of the world wide web (WWW) has fueled the development of search tools such as *Yahoo* [56], *the Harvest Web Broker* [9], and *Lycos* [38]. These tools are sufficient for simple keyword-based searching but the results contain simple information and are represented as HTML documents which impedes further processing. There is a need for tools which can assist consumers in the merging, the combining, and the sorting of retrieved results from multiple sources. Currently, users who wish to gather information from various tools need to do so by visiting each tool separately and then assembling the results manually. By creating wrappers to each information tool, Rainbow allows users to merge, combine, and sort their search results within a single query. No similar tool has yet been encountered during the time of this writing.

Since most of these information sources only allow network (HTTP) access, our wrapper is not physically located on the server machine of the tool. The link from our wrapper to the source is implemented through third party software (`get_url.pl` Perl Library) which retrieves output from remote HTML pages and from remote CGI scripts.

The diom_wrapper_Select generic function in Figure 7.3 will be used as the running example to illustrate reusability in DIOM wrapper construction and generation.

[4] a Perl based scripting language

```
sub diom_wrapper_Select {
        local($result);

        ## Fetch Data
        $result=&local_ExternalFetch;
        $verbose &&
                print "RAW PAGE ======\n<pre>\n $result \n</pre>".
                " \n====== End RAW PAGE\n";
        ## Normalization
        $tablename=&local_ExternalNormalize ($result);

        ## process normalize data
        $return=&local_Select ($tablename);

        ## clean up tmp tables
        &local_Close ($tablename);

        ## return answer
        $return;
}
```

Figure 7.3: The diom_wrapper_Select function for network semi-structured wrappers

```
sub local_ExternalFetch {
    ## Template for Web Pages
        local($url);

        ## compose correct URL
        $url=&HTML_makeURL;
        ($verbose) && (print "URL: $url\n");

        ## grab web page
        eval "\$result=&GrabWebPage(\$url)" ||
                die "Page Grab failed\n";

        ($verbose) && (print "Page Grab SUCCESS\n");
        $result;
}
```

Figure 7.4: The local_ExternalFetch function for HTML Information tool wrappers

The code in Figure 7.3 is written for network semi-structured wrappers. The function first fetches the data from the network information tool, normalizes it inside the wrapper DBMS, performs the necessary selections on the data based on the condition and then returns the result as a packaged DIOM object (MIME type).

Figure 7.4 and Figure 7.5 show the code for the HTML information tools local functions: local_ExternalFetch and local_ExternalNormalize network wrapper. These wrapper functions are a subtype of network wrapper function. The local_ExternalFetch function composes the correct URL to the information tool, grabs the web page, and returns the raw result up to the Select function. The local_ExternalNormalize function takes the raw result and normalizes it inside the wrapper database.

Wrapper to Yahoo

The Yahoo system only allows for two types of functionality: keyword searching and category-based browsing. We can consider the classes and instances to be read only. Thus, some of our wrapper functions are limited. The searching functionality of Yahoo is accessible through the URL to Yahoo's search engine.

```
sub local_ExternalNormalize {
        local($result)=@_;

        ## create temporary table name to use
        $tablename=&local_MakeTableName;

        ## create normalization commands
        @insertCommands =
        &HTML_translate($tablename,$result);

        ## print commands
        $verbose && print "INSERT COMMANDS ".
        "=======\n@insertCommands\n===== END INSERT ".
        "COMMANDS\n";

        ## execute commands to insert into table
        eval "&insertScratch (\$tablename, ".
        "\@insertCommands)" || print "ERROR with ".
        "creating scratch table and data";

        # return $tablename;
        $tablename;
}
```

Figure 7.5: The local_ExternalNormalize function for HTML information tool wrappers

The formation of the Yahoo URL starts off with the web host and path to the cgi script:

```
source: http://search.yahoo.com/bin/search?
```

Next, to form a correct URL we have to embed the proper functionality within the search string:

```
p=<keywords>                (separated by +)
d=(ignore this)             (yahoo, usenet, email)
s= o|a                      (or / and)
w= s (partial) | w (exact)  (word matching partial / exact)
n=<number>                  (number of results.. 100 max for Yahoo)
```

Here are some Yahoo URL examples of searching for the keywords job and multimedia:

```
http://search.yahoo.com/bin/search?p=job+multimedia

http://search.yahoo.com/bin/search?p=job+multimedia&d=y&s=o&w=s&n=25
```

The HTML_makeURL function utilized in the function in Figure 7.4 is the function which creates the proper URL to Yahoo. It was created by taking the above Yahoo URL formation knowledge and creating a Perl script to manipulate the condition field given in the call to wrapper. The HTML_makeURL for Yahoo is shown in Figure 7.6.

The HTML_translate function utilized in the function in Figure 7.5 is the function which extracts the URL and description from the raw HTML result where one record has this format:

```
<li><a href="http://mlds-www.arc.nasa.gov/BAMTA/">BAMTA <b>Job</b>
Bank</a> - <b>multimedia</b> and Web technology related positions.
```

It can be seen that if each record in the HTML raw result has the same format as the one give above, then parsing for each such record will allow for the extraction of data. The

```
sub HTML_makeURL {
        local($url, @keyword);

        ### source
        $url .= 'http://search.yahoo.com/bin/search?';

        ### keywords
        $url .= 'p=';

        $verbose && (print "local_ExternalFetch $condition: $condition\n<p>");
        $condition =~ s/'([^']+)'/$keyword[$x++]=$1/eg;
        $verbose && (print "local_ExternalFetch KEYWORDS: @keyword\n<p>");

        foreach $i (@keyword) {
                $url .= "$i+";
        }
        $url =~ s/\+$/&/;

        ### or/and
        if ($condition =~ /or/) {
                $url .= 's=o&';
        }
        else {
                $url .= 's=a&';
        }

        ### partial/complete
        if ($condition =~ /%(.*)%/) {
                $url .= 'w=s&';
        }
        else {
                $url .= 'w=w&';
        }

        if (!($numfetch) || ($numfetch < 50) ) {$numfetch =50;}
        $url .= "n=$numfetch";
        $url;
}
```

Figure 7.6: HTML_makeURL function for Yahoo

```
sub HTML_translate {
## this routine takes yahoo search result and creates oracle
## insert commands

        local ($tablename,$result)=@_;
        local ($description,$url, $commands);

        foreach ( split(/\n/, $result) ) {
                if ("\L$_" =~ /<li>/) {                 # SINGLE ITEM

                        s/^\s+|\s+$//g;                 # DELETE SPACES
                        s/<li>//g;
                        s/<LI>//g;
                        $line = $_;

                        s/\<\/A>.*//g;                  # DELETE HTML TAGS
                        s/\<\/a>.*//g;                  # DELETE HTML TAGS
                        s/<A HREF="//g;                 # DELETE HTML TAGS
                        s/<a href="//g;                 # DELETE HTML TAGS
                        s/"\>.*//g;                     # DELETE HTML TAGS

                        $url = $_;

                        $_ = $line;

                        s/<a href.*"\>//g;
                        s/<A HREF.*"\>//g;
                        s/\'|\"//g;
                        $description = $_;

                        $insert="insert into $tablename ".
                        "(description, URL, URL_URL, Source, ".
                        " Source_URL) values \n".
                        "('$description'\n".
                        ",'$url','$url','Yahoo',\n".
                        "'http://www.yahoo.com/')";

                        #now insert into commands array
                        unshift (@commands,$insert);

                } # if <li>

        } # foreach
        @commands;
}
```

Figure 7.7: HTML_translate function for Yahoo

HTML_translate is based on the format of Yahoo return records. the HTML_translate for Yahoo is shown in Figure 7.7.

Using the diom_wrapper_Select function as an example, the derivation of the local HTML functions was illustrated. For any HTML tool the only two functions which need to be defined are the HTML_makeURL and HTML_translate functions. To generate wrappers for information tools becomes a two-function creation process. Eventually, these function creation steps can be automated using a well designed interface for entering the URL description and a description of how the HTML record is formatted.

Wrapper to Harvest, Lycos and others

Creating wrappers to Harvest, Lycos and others is implemented by reusing parts of the Yahoo wrapper and overwriting the code for URL creation and HTML page translation. For example, the URL to the Harvest Web Broker search engine is different so this part needs to be re-implemented. The translation of HTML page into a DIOM object also may need to be redone, but the general interface to the wrapper is identical to Yahoo and can be reused. This is efficient and doesn't require recompilation of client programs which access the wrapper.

Any HTML page or HTML search tool can now be more readily incorporated into our system using our HTML wrapper skeleton. The only part which has to be re-implemented is the formation of the URL from the condition keywords and the translation of HTML information into class information methods. The main interface can be reused allowing for rapid incorporation of new WWW data sources.

Here is a list of the functions which must be re-implemented from the YAHOO wrapper to other HTML-tools wrapper:

&local_ExternalFetch This contains the code for URL formation

&local_ExternalNormalize This translates the HTML page into internal wrapper format

&local_GetClassInfo This translates the HTML general information or browser page into class descriptions

&local_InfoNormalize This does the same as &local_GetClassInfo

&local_ReqVariablesInfo This returns the list of variables needed to access the web tool. Some tools may require special passwords and usernames

Wrapper to Usenet

Each Usenet newsgroup is modeled as a separate class and each article in the newsgroup is modeled as an instance of that class. Each one of the generic wrapper functions must translate the DIOM request into a Usenet client request [26] and translate the result back into a DIOM result. To facilitate data functions on Usenet data, we scan the data in its native format and insert the articles into a local database system (normalize). We then perform query processing using the database and the result is translated to a DIOM-Result.

Each DIOM wrapper function is based on local functions which are reimplemented for each wrapper. We now present the list of wrapper functions and the pseudo-code of our Usenet wrapper implementation.

&diom_wrapper_Select
is implemented by first gathering all news headers from each article into a database and running the query on the database objects.

1. execute &local_ExternalFetch to fetch news articles.
2. execute &local_ExternalNormalize to translate fetched data into database insert commands and insert into database.
3. execute &local_Select to select article matching condition
4. execute &local_AddToResult to add select articles to DIOM-result (&local_ResultsToDiom)

&diom_wrapper_Delete
operations can be mapped to searching for articles based on the *condition* and then using Usenet cancel article commands to cancel all articles found. The algorithm is shown here:

1. execute &local_ExternalFetch to fetch news articles through the News server.
2. execute &local_ExternalNormalize to translate fetched data into database insert commands and insert into database.
3. execute &local_Select to select article message-id's matching delete condition
4. execute &local_Delete which sends cancel requests to news server for each message-id.
5. execute &local_AddToResult to add SUCCESS or FAIL to result

&diom_wrapper_Insert
operations can be mapped to a Usenet news posting. The algorithm is shown here:

1. execute &local_Insert to translate DIOM Request into local insert request. In this case a Usenet news post request. Post news.
2. execute &local_AddToResult to add SUCCESS or FAIL to result

&diom_wrapper_CreateClass
can be mapped to a create new newsgroup request (&local_CreateClass).

&diom_wrapper_DestroyClass
can be implemented as either a destroy newsgroup command [5] or by deleting the newsgroup from the News Server's available newsgroup's list. Again the DIOM-Result should contain SUCCESS or FAIL. (&local_DestroyClass).

[5]Because Usenet news feeds are duplicated on a mass scale complete deletion of newsgroups may be unrealistic

&diom_wrapper_GetClassInfo
retrieves the list of news groups and news group descriptions available from the server.

1. execute &local_GetClassInfo. This queries the news server for all *available* newsgroups and descriptions.
2. execute &local_InfoNormalize. This translates newsgroup information from the server into the internal database.
3. execute &local_Select to select inserted records from the internal database.
4. execute &local_AddToResult to add the selected records to the DIOM result.

&diom_wrapper_GetExportSchema
retrieves all possible newsgroups and returns the results.

1. execute &local_GetExportSchema. This queries the news server or queries an internal list of all *possible* newsgroups in the Usenet domain.
2. execute &local_ExportNormalize. This translates newsgroup information from the server into database commands which are then used to insert into the scratch database.
3. execute &local_Select to select all newsgroup records
4. execute &local_AddToResult to add the selected records to the DIOM result.

&diom_wrapper_ReqVariablesInfo
retrieves the variables necessary to access the data repository. To access news articles we simply need the newsgroup name and article numbers. To post news we need user name, subject, organization, article body, newsgroup name. Our function will simply return all possible variables that can be filled back to the client.

1. execute &local_ReqVariablesInfo which returns a DIOM-result listing all the variable names and type that can be entered. Required and optional variables are marked.

Each Article retrieved from this wrapper can undergo further processing to extract extra information. For example, the wrapper may return articles from the jobs newsgroup. A simple scan of the data may only reveal the subject, article creator, and the organization of the poster. But further processing may also uncover the job title, job pay, job location, and the employer. We will elaborate on further processing more fully in our Mediator presentation.

All *local* functions must be re-implemented for each new wrapper. Some local functions are highly specific such as the &local_ExternalFetch and &local_ExternalNormalize functions. If a common local database is used for normalization then &local_Select can be reused. For example, if Oracle is used as the common underlying database system, then &local_Select can be used for any wrapper using Oracle for the internal database.

Wrapper to Local Semi-Structured Files
Flat files require some structure if they are to be incorporated into our system. Flat files without any structure are generally harder to wrap as they may require complex knowledge discovery heuristics to gather any useful information. Hence, we will focus on the use of semi-structured data files (e.g., BibTeX) as the target of our flat file wrapper construction.

Many bibliographies are stored in BibTeX format. The BibTeX format allows any sort of documentation such as books, technical reports, magazine articles etc. to be stored and used as references within the LaTeX documentation system. Each BibTeX entry has specific field name and formats. Here is an example BibTeX entry:

```
@techreport{bib:21,
author = ``A. Fekete and N. Lynch and M. Merritt and W. Weihl'',
title = ``Commutativity-based locking for nested transactions'',
institution = ``MIT, Lab. of Computer Science'',
number = ``MIT/LCS/TM-370.b'',
year = ``1989'',
month = jul}
```

Using the the field name on the left side of the equality sign and associated value on the right side, each document entry can be instantiated in a DBMS for further processing. For example, all book entries can be instantiated in the Book class, all journal entries into the journal class, all article entries into the article class. By first scanning and inserting entries into a Oracle database we can reuse most of the code from the Oracle wrapper to implement our BibTeX wrapper.

The *delete* command can delete entries stored in the flat file, the *insert* command can insert new entries, and the *select* command can utilize the normalized database based on the native data to execute the query. Using the metadata browsing capability, users are able to query the classes used by a specific wrapper. To retrieve class information we can return all classes and attributes which are utilized in the flat file. To retrieve the export schema we can return all the classes available in the BibTeX standard.

7.5 Implementation Issues

Currently, the wrapper functions are implemented as a Common Gateway Interface 1.0 script in Oraperl, and can be accessed through HTTP protocol. The invocation commands to a wrapper are embedded within the HTTP call and the result is returned as a MIME object. Oraperl, an Oracle extension of the Perl language, includes functions for accessing Oracle Databases. Raw data from the repositories is normalized into the Oracle database which then allows for further data processing.

Perl is an interpreted language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. It's also a good language for many system management tasks. The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal).

Oraperl is an extension of Perl which offers an API to Oracle DBMSs. The new version of Perl 5.0 offers the DBperl package which offers a generic API to many other DBMSs. Future implementations of the Perl-based wrappers will use the DBperl [25] package which will allow a single wrapper implementation to run over multiple underlying repository types.

During the last stage of the Rainbow prototype development, Sun Microsystems announced the release of the Java JDBC [6] [51] database access API. JDBC will become a

[6]Java and JDBC are trademarks of Sun Microsystems Inc, http://splash.javasoft.com/jdbc/

widely accepted API to many information sources. Java-based wrappers can now also be implemented. Utilizing JDBC will allow single wrapper implementations to be used for all underlying supported repositories. Some wrapper processing with JDBC implementation may also allow for more distributed computing in the sense that those processing tasks which are less data intensive can be shifted to the client machine.

Further features which could be added to our wrapper functionality include: More metadata features such as export schema names (choices), repository name, source type, etc. A clear export schema specification needs to be created so wrappers can export schemas. The processing of more complex parallel query plans may require data to be shifted from one repository to others or to intermediate servers. This calls for more interoperation between wrappers. Furthermore, more research into wrappers to other types of sources such as object oriented databases and knowledge bases is required.

Chapter 8

Related Work

To resolve semantic and schematic heterogeneity, there are a wide range of proposed solutions for global information sharing of multiple heterogeneous sources in a distributed environment. The main strategies of the past have focussed on varying degrees of integration ranging from tightly-coupled to more loosely-coupled [49, 36, 10] along with many other diverse characteristics which make classification difficult. The advantage to tight-coupling is close synchronization among sites leading to efficient global processing. For small-systems the creation and maintenance of a global integrated schema is feasible, but for larger scale systems this is a fundamental roadblock towards system scalability and evolution. In contrast, loose-coupling strategies favour zero schema integration. Loosely-coupled strategies have much less local-level control of repositories, and have a much more reduced global function set. These strategies typically access local repositories by applications (gateways, wrappers) which lie above the repository. Because there is no integrated schema, many of the responsibilities, such as semantic conflict resolution, may be left up to the user. The following sections will give a general overview of relevant research into major areas of multidatabase research and how they are similar to and different from the work presented in this thesis.

8.1 The Global Schema Approach

A classic approach [47, 48] for multidatabase management relies on building a single global schema to encompass the differences among the multiple local database schemas. Global schema integration approaches can be extreme such as in the case of a traditional *distributed database* where global and local functions share the same low-level internal interfaces and are so tightly integrated that there is little distinction between local and global functions. A more bottom-up design is that of a *global schema multidatabase* where global functions are implemented through the external interface of a local DBMS but there is still a global schema. All local sites must be diligent and cooperate closely to maintain the global schema.

Although the enforcement of a single global schema through data integration yields full transparency for uniform access, component DBMSs have much restricted autonomy, scalability and their evolution becomes difficult. Our project is an attempt to support the USECA properties. Thus, it is infeasible to maintain a global and monolithic schema in a large-scale interoperable environment.

8.2 The Federated Approach

Federated databases [24] are made up of component databases which are semi-autonomous and are a more loosely coupled subset of the global schema approach. Each local DBMS defines an export schema. Based on these export schemas, applications define import schemas which can be considered a global external view. Although they are semi-autonomous, each component database must still cooperate closely with the specific nodes it accesses. Changes to the local schemas may have profound impact on the global views and may require reworking of the global views.

This approach cannot scale well when new sources need to be added into an existing multidatabase system. Furthermore, the component schemas cannot evolve without the consent from the affected integrated schemas. Again a federated approach does not satisfy the USECA requirements.

8.3 The Distributed Object Management Approach

The distributed object management approach [37, 35], generalizes the federated approach by modeling heterogeneous databases of different levels of granularity as objects in a distributed object space. It requires the definition of a common object model and a common object query language. Recent activities in the OMG and the ODMG standard [15], which extends the OMG object model to include database interoperability, is an important standardization for continued work in distributed object management.

Research into DIOM [31] takes ODMG and adds extensions targeted to allow for more flexible and adaptive composition of interoperation interfaces, such as the addition of the abstraction mechanisms presented earlier in this thesis.

8.4 Multidatabase language systems

In a multidatabase language system, the global system supports all global functions by providing query language tools which aid in the integration of information from separate databases. Language tools include functions to map information from different representations into one the user is familiar with or into partial global schemas. User queries can then specify queries against local schemas of the nodes participating in the system. The mapping from each local schema to other partial or full global schema is often expressed in a common SQL-like language, such as HOSQL in the Pegasus System [1] or SQL/M in the UniSQL/M system [16]. The main weak point of this approach is the poor scalability since the users often need to know what are the information sources that are relevant or of interest to her queries.

The DIOM model allows for the creation of user interfaces through DIOM-IQL and DIOM-IDL without knowing the information sources. The DIOM query mediation services will automatically connect consumers' query requests to the relevant producers' source information.

8.5 Mediator Approach

Another approach, called the intelligent information integration (I^3) mediation [53] can be seen as a generic system architecture for information integration, with several projects funded by ARPA in the I^3 program [54]. For example, TSIMMIS [20] (The Stanford-IBM Manager of Multiple Information Sources) is one of the best known systems, which implements the mediators-based information integration architecture through a simple object exchange model. Another example is the Context Exchange project [22, 50] at MIT. It uses context knowledge in a context mediator to explicitly define the meaning of information provided by a source and that required by a receiver. Similarly, the Intelligent Agent Integration Project at the University of Maryland at Baltimore County uses the Knowledge Query and Manipulation Language [18] (KQML) to integrate heterogeneous databases. An example of a wrapper-based architecture is the Garlic [12] project at IBM Almaden Research Center, which aims at developing a system for the management of large quantities of heterogeneous multimedia information using repository wrappers. The SIMS project [2, 3] uses the LOOM knowledge representation system as the data model and query language to implement the agent-based integration. Other examples include University of Maryland [13, 19] and SRI [46], which in different ways integrate heterogeneous data and knowledge bases using a multiple F-logic object schema, via the KIF knowledge interchange logic.

The DIOM architecture is motivated mainly by the mediator architecture. For example, this architecture incorporates the notion of a logical mediator which keeps knowledge on a specific domain and uses wrappers to bridge the gap between the interoperable database system and the individual component repositories. TSIMMIS uses a very simple object model (Object Exchange Model) which does not provide explicit support for data abstraction mechanisms such as in DIOM. DIOM utilizes a rich set of object-oriented mechanisms enabling the incremental construction of new interfaces from existing ones. Furthermore, among these projects, only a few have dealt with the issues of query processing and query optimization. For example, TSIMMIS query processing is based on pre-defined query pattern matching. The query reformulation and simplification are performed by using logic-based rewriting rules against pre-defined query patterns. The query processing in the information mediation project at the university of Maryland and SRI uses F-logic to define a set of heterogeneous object equivalence rules to facilitate the mapping between the multidatabase and the local schemas. The query processing in SIMS is done through query reformulation and query optimization by the LOOM reasoning module. These related projects have not addressed all the requirements of the USECA properties in a systematic manner, which is the goal of our approach.

8.6 Enabling Technologies

A number of proposed and implemented systems have emerged as basic enabling technologies for implementing interoperable objects in a distributed and dynamic object environment. Examples include Microsoft's Object Linking and Embedding (OLE), IBM's System Object Model (SOM) and its distributed version (DSOM), OMG's Common Object Request Broker Architecture (CORBA) [23], and CI labs OpenDoc.

The research and emergence of these technologies demonstrates that object-oriented programming and object-oriented database management systems are evolving and are focussing towards language-independence in a distributed object computing environment. Although these proposals are early practical and important, they focus primarily on the software interface problem, not the USECA properties for a large scale system.

DIOM can be seen as the glue which spans and integrates these interface models at a higher level. For example, unlike in CORBA, ODMG, and DSOM where only specialization/generalization are considered in interface construction, DIOM also supports the import and aggregation mechanisms for interface abstraction. These abstractions provide more flexibility and convenience in interface composition and also increase the robustness, scalability, and adaptiveness of compound interfaces in the presence of component schema changes and system evolution.

All of the above offer systems of integrating heterogeneous sources and offer different advantages and disadvantages. In an interoperable world of the future, we contend that the DIOM system has the ability to incorporate the above systems by creating wrappers to each. In this way all systems can interoperate which is the initial design of DIOM.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

This thesis describes the Rainbow prototype implementation of the DIOM query mediation methodology, which offers adaptive querying capabilities over multiple heterogeneous data sources. The term, *adaptive*, refers to the extensibility, the scalability, and the composibility of the services with respect to the growing number of sources and the evolving requirements of producers' source models and consumers' query models. This tool was designed as a prototype to illustrate the functionality of the DIOM interoperable object model, the system architecture and the DIOM query mediation method. The target environment was the Internet and the example application was a Job Search Assistant application. This thesis provided an extended overview of architecture, and presented the design and implementation effort in prototyping the DIOM query mediation method.

The main objective of this implementation is to design and implement the framework to allow for future development. The main components of the Rainbow prototype include:

- The design of a graphical user interface offering major system functions including producer and consumer metadata functions and querying functions.
- The design of underlying database schema needed to capture and maintain metadata on producers and consumers.
- Rainbow Services Module to process client requests and return results from real sources.
- The wrapper design and implementation including the partition between the general wrapper interface from the local functions which aids in the reuse and generation of wrappers.

Rainbow was implemented as a WWW application. Its interface was created using HTML and Perl cgi-scripts. The individual wrappers were also created as Perl programs. Linking to the underlying database was primarily done through Oraperl. Many versions of the prototype were built during development. The user interface and the Rainbow system were refined as requirements and functionality became clearer.

9.2 Ongoing and Future Implementation

A prototype system will require much more work and research. Currently, several of the underlying sub-system components are *simulated*. For Rainbow to become a production-quality system more effort is needed to make it fully functional. Here is a list of the general sub-divisions within the DIOM project which will require larger investments of time and were not in the scope of this thesis:

Front End
A better interface to capture consumer domain knowledge and to allow for consumers to compose new interfaces is needed. Continued work needs to be done in refining the WWW interface and incorporating Java to allow for more versatility and richer functionality.

Metadata Management/Distributed Catalog Service
All metadata has to be stored efficiently, methods for maintaining the metadata and replication strategies need to be designed and implemented, including distributed algorithms for caching/replication and dealing with issues of performance and availability.

Distributed Query Mediation Services
The ongoing DIOM research includes efforts in several areas, for example, the formal development of a distributed object query algebra, including joins involving multiple information sources, the design and validation of algorithms for query decomposition, and the design and validation of algorithms for generation of optimal parallel access plans. Optimal access plans should take into account the efficient processing of aggregate functions such as SUM, COUNT, MAX, and MIN. Research into the prototype implementation of continuous query [33] support is also open.

In the prototype, query tracing is simulated and not fully functional. The Rainbow Services Manager needs more work to fully support the DIOM query decomposition algorithms and the query assembly algorithm.

Heterogeneity Management
This area includes semantic matching of consumer interface descriptions with producer source metadata and utilization of the state-of-art research in ontology to improve the resolution of heterogeneity problems and to provide quality interconnection between information consumers and producer sources.

Interconnections of Information Sources
Wrappers to other types of sources need to be created. Currently, wrappers are designed for a small generic function suite to Oracle, HTML-based information tools, and Usenet sources. Assistance is needed in generating and installing wrappers for various other sources with emerging tools such as JDBC and DBperl.

Bibliography

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

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

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

[4] K. Arnold and J. Gosling. *The Java(tm) Programming Language*. Addison-Wesley Publishing Company Corporate and Professional Publishing Group, 1996.

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

[6] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform resource locators (URL) (RFC 1738). http://ds.internic.net/rfc/rfc1738.txt, October 1994.

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

[8] N. Borenstein. RFC 1521: MIME (multipurpose internet mail extensions) part one: Mechanisms for specifying and describing the format of internet message bodies. ftp://ds.internic.net/rfc/rfc1521.txt, September 1993.

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

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

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

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

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

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

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

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

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

[18] T. Finin, D. McKay, R. Fritzsonand, and R. McEntire. KQML: An information and knowledge exchange protocol. *Knowledge Building and Knowledge Sharing*, 1994.

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

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

[21] Georgia Tech Research Corporation. GVU's 4th WWW user survey home page. http://www.cc.gatech.edu/gvu/user_surveys/survey-10-1995/, 1995.

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

[23] O. M. Group. The common object request broker: Architecture and specification. OMG Document Number 91.12.1, Revision 1.1, 492 Old Connecticut Path, Framingham, MA 01701, December 1991.

[24] D. Heimbigner and D. McLeod. A federated architecture for information management. *ACM Trans. Comput. Syst.*, 3(3):253–278, July 1985.

[25] Hermetica. DBI – a database interface module for perl5. http://www.hermetica.com/technologia/DBI/index.html, 1995.

[26] M. Horton. RFC 1036: Standard for interchange of USENET messages. ftp://ds.internic.net/rfc/rfc1036.txt, December 1987.

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

[28] J. Kahng and D. McLeod. Dynamic classification ontologies for discovery in cooperative federated databases. In *Proceedings of the International Conference on Coopertive Information Systems*, pages 26–35, Brussels, June 19-21 1996. IEEE Press.

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

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

[31] L. Liu and C. Pu. The DIOM approach to large-scale interoperable information systems. Technical report, TR95-16, Department of Computing Science, University of Alberta, Edmonton, Alberta, March 1995.

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

[33] L. Liu, C. Pu, R. Barga, and T. Zhou. Differential evaluation of continual queries. In *IEEE Proceedings of the 16th International Conference on Distributed Computing Systems*, Hong Kong, May 1996.

[34] L. Liu, C. Pu, and Y. Lee. An adaptive approach to query mediation across heterogeneous databases. In *Proceedings of the International Conference on Coopertive Information Systems*, pages 144–156, Brussels, June 19-21 1996. IEEE Press.

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

[36] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, 1991.

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

[38] M. L. Mauldin and J. R. R. Leavitt. Web agent related research at the center for machine translation. Center for Machine Translation, Carnegie Mellon University, August 1994.

[39] E. Mena, E. Kashyap, A. Sheth, and A. Illarramendi. OBSERVER: an approach for query processing in global information systems based on interoperation across pre-existing ontologies. In *Proceedings of the International Conference on Coopertive Information Systems*, pages 14–25, Brussels, June 19-21 1996. IEEE Press.

[40] K. Moore. RFC 1522: MIME (multipurpose internet mail extensions) part two: Message header extensions for non-ascii text. ftp://ds.internic.net/rfc/rfc1522.txt, September 1993.

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

[42] Netscape Communications Corp. Netscape homepage. http://home.netscape.com/.

[43] A. M. Ouksel and I. Ahmed. Coordinating knowledge elicitation to support context construction in cooperative information systems. In *Proceedings of the International Conference on Coopertive Information Systems*, pages 4–13, Brussels, June 19-21 1996. IEEE Press.

[44] J. Perry and J. Lateer. *Understanding Oracle.* Sybex Inc., 2021 Challenger Drive #100, Alameda, CA, 1989.

[45] T. Portfolio. *PL/SQL User's guide and reference.* Oracle Corporation, Dec. 1992.

[46] L. Qian and L. Raschid. Query interoperation among relational and object-oriented databases. In *Proceedings of the International Conference on Data Engineering*, TaiPei, 1995.

[47] S. Ram, editor. *Special Issue on Heterogeneous Distributed Databases Systems*, volume 24:12 of *IEEE Computer Magazine.* IEEE Computer Society, December 1991.

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

[49] A. Sheth and J. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

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

[51] Sun Microsystems, Inc. The JDBC(tm) database access API. http://splash.javasoft.com/jdbc/, 1996.

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

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

[54] G. Wiederhold. I^3 glossary. *Draft 7*, March 16 1995.

[55] R. W. Wiggins. Growth of the internet: An overview of a complicated subject. http://www.msu.edu//staff/rww/netgrow.html, 1995.

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