Multidatabase Support for Object-Oriented, Multimedia Authoring Environments

by

Katherine Curtis

B.A., B.A.I., Trinity College, University of Dublin (1991)
S.M., Massachusetts Institute of Technology (1992)

Submitted to the Department of Civil and Environmental Engineering
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in the field of Information Technology

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1996

Author ... .
Department of Civil and Environmental Engineering
May 22, 1996

Certified by .....
Prof. Steven R. Lerman
Department of Civil and Environmental Engineering
Thesis Supervisor

Accepted by .....
Prof. Joseph M. Sussman
Chairman, Departmental Committee on Graduate Studies

Multidatabase Support for Object-Oriented, Multimedia Authoring Environments

by

Katherine Curtis

Submitted to the Department of Civil and Environmental Engineering on May 22, 1996, in partial fulfillment of the requirement for the degree of
Doctor of Philosophy
in the field of Information Technology

Abstract

This thesis examines the advantages of including a database component in an object-oriented multimedia application authoring environment. A database can provide superior methods for handling application data, particularly multimedia data, and ensuring data integrity. The importance of providing the application author with a common interface to a variety of available database systems is emphasized. An innovative design for a multidatabase system for use with an authoring system is developed. This system is implemented for the AthenaMuse® authoring tool and several applications, which have been developed using the multidatabase features, are evaluated. The multidatabase system provides an excellent foundation for further exploration into the role of database technology in application authoring environments. A new paradigm for application storage is developed by expanding the multidatabase system to handle the storage and retrieval of application objects. These abilities are integrated into an application editor for evaluation. The advantages of storing entire applications, not simply application data, in a database are discussed.

Thesis Supervisor: Steven R. Lerman
Title: Professor

Acknowledgments

To everyone at the MIT Center for Educational Computing Initiatives, without you this thesis would not have been possible.

To Jud, and all the members of the AthenaMuse development team, for their support during the development of the database module.

To Adam, Issam and Juan, fellow graduate students who were always willing to help out with suggestions, advice and encouragement.

To my advisor, Prof. Lerman, for his guidance during the writing of this thesis and his continual support at CECI.

To Prof. Logcher and Prof. Williams, members of my thesis committee, for their time, comments and insights.

To Alan, who was there through all the ups and downs of life at MIT.

Contents

Table of Figures

Introduction

The demand for multimedia applications has increased dramatically with the widespread availability of fast multimedia capable computers. Applications range from adventure games, to innovative learning environments, to business data processing aides. The design and implementation of these applications is carried out using one of the many available application authoring tools, or more often, a combination of these tools. Tools range from specialized image and video editing suites to media integration environments where a final application can be assembled.

This thesis examines the role that database systems can play in the development of complex applications, many of which require expert management of media or other data. The first step taken is to consider the demands that an application designer or user might make on a database system. As it quickly becomes apparent that an application must be able to interact, perhaps simultaneously, with a wide variety of existing database systems, the advantages of providing a generic database system interface to an application are discussed. A review of current database technologies is carried out in order to highlight the strengths and weaknesses of other multidatabase development initiatives.

In order to allow an application to take full advantage of a database system, an innovative multidatabase module is proposed for inclusion in an application authoring tool. The module is designed to allow complete access to a database system via a standard interface, consistent across all databases. The module provides for a set of standard data types and a standard query language and is designed using a layered architecture to allow new database types to be added with minimal effort. The validity of the proposed module is proved by incorporating it into an advanced object-oriented multimedia application authoring tool. A selection of applications are presented that have been developed using the tool and that rely on the multidatabase component to achieve significant improvements in functionality or performance.

The multidatabase system provides an excellent foundation for further research into both multidatabase systems, and innovative uses for the multidatabase within the application environment. One example, drawn from the latter field, is application storage. A typical

application can be divided into two parts, the application data and the application behavior. In the past, only the permanent storage of application data has been considered. A novel extension to the multidatabase system already proposed allows the application behavior to be captured in a database, enabling the development of a new paradigm in application storage. The advantages of this new concept in application storage are evaluated by their inclusion in an application editing tool.

Chapter 1

Motivation

Computer applications have long been popular in the fields of business, education and entertainment. These applications range, with infinite variety, from record keeping to history lessons to games. As computer hardware and software technology has advanced, these applications have become significantly more complex and now often incorporate large amounts of multimedia data to increase their appeal and usefulness. A good example of a complex application that references large amounts of data is a language training course that centers around a visit to a Spanish speaking city in South America. The application uses video and audio clips, as well as still images, to recreate the atmosphere of the city and to allow the user to listen and interact with native Spanish speakers. In addition, the application provides extensive information to the user about South American culture, economics and politics. In business, such multimedia applications are being looked to as a more effective and economical method of on-site training of employees. Companies use databases to store large amounts of important data such as customer records, financial accounts and stock information. Frequently, as a company expands, the number, type and complexity of database systems employed within the company increases. The databases are often located on different machines, running different operating systems, perhaps spread across the globe at a variety of company sites. Traditionally, separate applications have been developed to allow employees to access the information stored in each database. Combining data from different systems is not usually a trivial task. The ideal work environment would provide for a single application, able to communicate with all types of database, both legacy and current systems, and providing integrated information to the user.

Initially, all applications were written in a traditional programming language for a particular hardware platform. Any data needed by an application would be stored in external flat files or embedded inside the application itself. However, as applications have increased in complexity and the market for such applications has grown, a number of application authoring tools have

become available. Such tools provide an application author with the appropriate primitives and facilities to compose an application in a reasonable amount of time, without needing to be an expert programmer. However, none of these authoring environments has adequately explored the issue of the storage of complex data referenced by the application and subsequent convenient access to this data from the application. The data need not be stored locally, but could be located anywhere on an accessible network of computers and file servers.

1.1 The AthenaMuse® Authoring Environment

There are many application development tools now available, both commercially and as research prototypes. AthenaMuse® [Meehan et al. 1995] is a distributed multimedia application development environment created at the MIT Center for Educational Computing Initiatives. It serves as a good example of the way in which all development tools must manage application data and accommodate diverse user needs. For this reason, the discussion of the need to improve the handling of application data in the AthenaMuse® environment is relevant for all such environments. A brief description of AthenaMuse® is included here to aid in this discussion.

The AthenaMuse® environment (shown in Figure 1.1) provides an engine to control the execution of the application and an application description language (ADL) for use by the application author. In common with a number of emerging development tools, such as Java™ [van Hoff et al. 1995], AthenaMuse® is an object-oriented system. The ADL provides access to a set of base object classes that can be used as a framework for constructing a multimedia application. The user can also custom design his or her own classes to suit his or her particular needs. The base classes are implemented in one of the application modules of the system. Currently, media, interface and network modules exist. The media classes provide a standard interface to structured text, image, video and audio data elements and a means of controlling the presentation of these elements. The interface classes provide the screen elements, such as windows, menus and buttons that make up an application. The network classes provide an interface to various network protocols so that client server applications can be built easily or the application can gain easy access to the world wide web. The AthenaMuse® system was designed to appear platform independent to the application author and user. The ADL code written for an

application does not need to be altered to run correctly on different hardware systems. Currently the control engine has been implemented on UNIX, Windows and Macintosh systems.

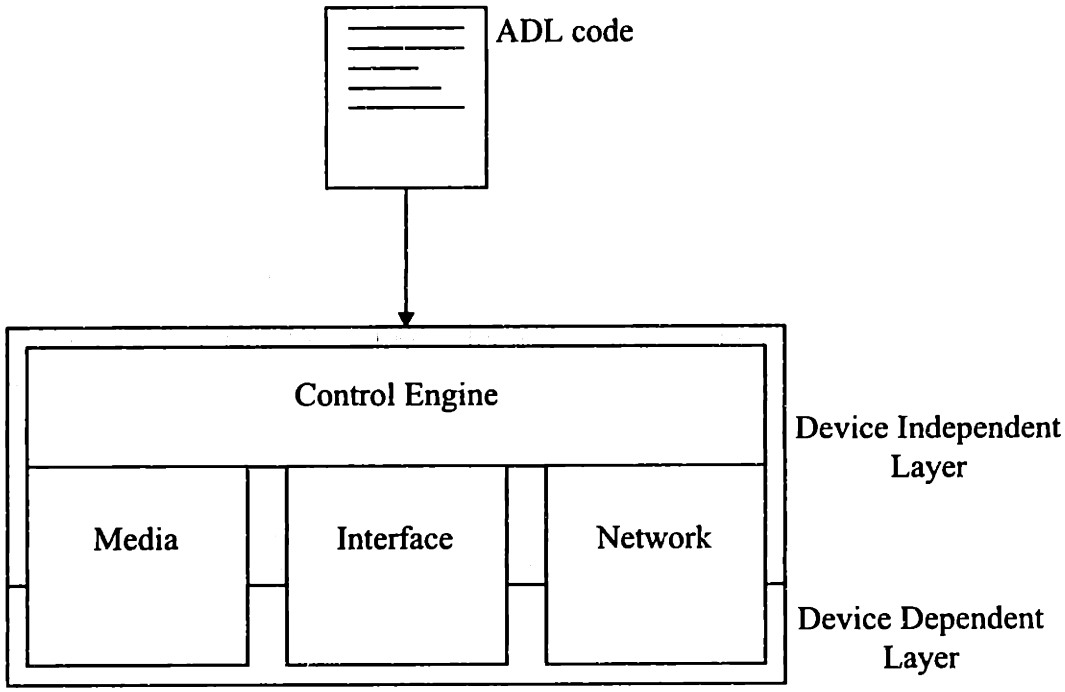


Figure 1.1 The AthenaMuse® Application Development Environment

All applications rely on the availability of large amounts of multimedia and other data. A great advantage to the application user would be if this data were available through a more sophisticated system than embedding external file names and data values into the application code. Data values hardwired into the application code in this way are very difficult to modify or delete. If a mistake is made, or if information needs to be updated, then the entire code must be manually searched for the data to be replaced and the new value entered. If an external file that is referenced by the code needs to be moved or renamed, then again the code must be manually altered. It is almost impossible to keep an orderly record of the data and files being used by any application at any given time. Thus, if an application needs to be moved, it becomes very hard to ensure that all the data resources used by the application are also moved at the same time. Similarly, without any centralized data control, if an application needs to be localized (e.g. changing the language in which text appears), it is very hard to ensure that all the needed modifications are made.

One obvious, but almost entirely untried, answer to this problem is to provide database support to such applications. The database provides a centralized environment in which data can be stored and accessed in an orderly fashion. By using a database access language, data can be specified independently of the file system and other environment variables. Data can be modified in the database without the need to modify the application code itself. Database systems are specifically designed for tasks such as data entry, deletion, or modification and provide appropriate interfaces for these functions. Additionally, database systems provide extensive facilities for data searching and querying that would not normally be available to an application. If localized data is required by an application, it can be stored in the database and a simple data query used to retrieve the correct data set. Once all application data is stored in a centralized location, it becomes simple to ensure that all data needed by an application is protected from accidental deletion and can be moved with the application as necessary.

1.2 The Need for a Multimedia Multidatabase System

Having declared the suitability of a database for control of the data needed by an application, questions remain concerning the type of database support that is necessary. Since the authoring environment is object-oriented, it would be an advantage if the database system used was also object-oriented. This would mean that any data objects created in the application could be stored directly in the database without the need for translation of data between the application and the database. The database should also have facilities for storing complex data as is required by multimedia and object data.

There are practical restraints that need to be placed upon the choice of database. In many applications, especially those aimed at the educational market, cost is an important factor. Such markets may not be prepared to pay the extra premium required to gain access to a commercial database system. One solution to this dilemma would be to write a proprietary database management system for the application authoring environment from scratch, while another would be to use a database system that resides in the public domain. The former solution requires a significant amount of effort and resources to be spent replicating work that has already been done elsewhere to good effect. Public domain databases tend to be poorly maintained, have limited functionality and cannot be relied on to keep up with technology changes such as hardware and

software system upgrades. However, some application developers and users may be willing to pay for the superior data handling capabilities of a sophisticated commercial database system. Alternatively, some organizations wishing to author an application may already have their application data stored in a database. For example, a museum wishing to write an application to access its art collection, may already have the collection stored in a database of their own choosing. Ideally, this data should be directly available inside the application environment and their should be no need to transfer the data elsewhere so that it can be accessed. The AthenaMuse® environment is portable across hardware platforms which indicates that the data access methods used in the application must also be portable. However, many database systems are written for specific platforms and few, if any, are available on all the platforms that are supported by AthenaMuse®. Thus the application may need to choose a different database depending on the available hardware platforms.

For all these reasons, the application environment calls strongly for a multidatabase solution where integrated access to a variety of databases is available. In this way, applications will be able to access data in pre-existing databases as well as in databases created during the application authoring process. Additionally, applications will be able to choose a database by weighing together economic, functional and performance criteria before deciding on the most suitable database for that particular application. It is envisioned that the multidatabase system would provide transparent support for access to database systems across a network. In this way, by combining the multidatabase with the multi-platform capabilities of an authoring environment, the database need not be constrained to run on the same machine, or even the same type of machine, as the application.

Every database system provides its own proprietary method for interfacing with an application. Obviously the application author does not want to have to learn how to construct a new interface for every database system than needs to be connected to an application. Therefore, the multidatabase system must offer a standard method of interfacing to different database systems. Unfortunately, no standard currently exists that covers the wide variety of database systems that it is anticipated will be needed in the authoring environment. Furthermore, the functional requirements for such a standard still need to be determined. At a minimum, the multidatabase system must provide a standard means for accessing data, but standard methods

for modifying data and accessing the database schema my also be appropriate. Since individual databases display different levels of functionality, the multidatabase system must provide a flexible interface that allows access to the extended functionality of some databases while providing extra functionality to compensate for the more primitive database systems.

1.3 The Storage of Application Objects

The incorporation of a multidatabase system into a multimedia authoring environment allows the environment to take advantage of the data management control and access facilities offered by a database system. While it quickly becomes evident that this is an excellent method of storing and retrieving application data, it is important not to overlook other possible uses of the multidatabase interface. The data forms only one component of the whole application. In an object-oriented application authoring environment, the application consists of a set of objects that together describe the behavior of the application, i.e., the way in which the data is presented to the user. Normally these objects would be specified in the application programming language and stored in a set of files. A novel alternative would be to store these application objects in a database. Then the entire application, behavior and data, could be stored together and a new way of constructing applications emerges. Since it remains undesirable to tie the authoring system to any one database system, the ability to store these application objects must reside in the multidatabase system. Some of the situations in which the ability to store application objects gives the system a significant advantage over rival authoring environments are described below.

Object Library. The application author may wish to provide the end user with a wide variety of choices inside the application. For example, the application may allow the user to choose whether or not to view a certain set of images. If the user chooses not to view the images, then there would have been little purpose to building an image viewer into the application. The viewer, although never used, would have increased the application complexity and size, possibly causing the application to perform more poorly than had the image viewer component been left out. A better solution might be for the application to request an image viewer object from the database only when it is needed by the user. In a similar way, if several image viewer objects are stored in the database, the application can search the database for the viewer most suited to

current user requirements without mandating the author to build a number of different image viewers into the application in advance. The ability to dynamically load objects into the application environment can significantly reduce the size of an application program and therefore the length of time taken and available memory needed for an application to initialize. Modifications and improvements made to the implementation of application objects stored in a database are automatically made available to applications which retrieve these objects, reducing the burden of application maintenance.

Application Archive. There is a tremendous amount of application code written which could be reused if it were readily accessible and documented. If application objects could be stored in a database, they could be indexed and made available to future authors in order to avoid unnecessary duplication of effort. The objects would be easier to find using database searching techniques than if they were simply stored somewhere in a file system. Once the desired object has been found, it becomes easy to incorporate the object into the new application, either by making a copy of the object for the application or by using it directly from the database. Such an archive mechanism, implemented by the multidatabase system, could also be used to keep track of object versions for application development purposes.

Object Sharing. If application objects could be stored in a database, then they would become available to multiple users who might wish to access the object at different times, using different applications. In this way an object, representing for example a complex engineering system, could be used by two or more engineers in a cooperative work environment. The database system can provide the transaction and locking mechanisms that must be used to maintain object consistency.

Object Validation. There is often a need to store the current state of an object for later reference and comparison with the same object after it has been modified. As database systems often include version control for data, it could be relatively simple to institute a method of versioning of application objects. An application object could then be compared at various stages of its development cycle. This may be especially important for runtime validation and error recovery

procedures. If a record of object versions has been kept, the object can be restored to its last valid state and the difference between versions may be helpful in pinpointing the error.

Application Editor. Combining the ideas of the archival, version control, sharing and validation of objects, the database could provide an excellent mechanism for controlling the development of object code for large applications. Any such system could also be used in the design of application editors. Editors are intended to aid the author in writing an application. They need to be able to store information about each new object the author wishes to create and to instantiate the object on demand. The author will wish to dynamically modify the objects by adding and deleting attributes, easy manipulations for a database.

1.4 Consideration of Previous Work

Previous research has tended to focus on the capabilities and development of complex multidatabase systems with little regard for the usefulness of such systems to application developers. Often such systems are designed for one particular application and are difficult to adapt for reuse in another application context. Little thought appears to have been given to the multidatabase interface requirements of an application authoring environment, where the multidatabase system must be flexible enough to serve the needs of a variety of designers and users. Similarly, object storage systems have been designed with the assumption that only one object database system, with its corresponding programming language, will be used. A new model for a multidatabase system is needed to overcome this inflexibility. A review of the literature reveals very few examples of applications that use a multidatabase interface or that give serious consideration to the general requirements for such a database interface. The three examples cited in the following paragraphs represent the published work in this area to date.

KMeD [Chu et al. 1995], developed at the University of California, is a distributed multimedia medical system providing an experimental knowledge base for a group of physicians. A set of independent databases, consisting of scientific, clinical and radiological data systems as well as a voice and picture archive systems, were combined in a multidatabase system. These databases were all based on the same data model which greatly simplified the integration procedure [Cardenas 1987]. The system also relied on the presence of an advanced object

database to function. The research carried out on the system focused primarily on the development of an advanced flexible approximate query language by combining a multidatabase system and a knowledge based system. However, the research also touched on some important issues in the management and processing of multimedia data such as the ability to relate stored images in space and in time. For example, a set of images can show the development of an illness over time or a set of images can together make up a complete picture of a human body.

A similar project, Telemed [Bucci et al. 1994], was undertaken at the University of Florence as part of an effort to establish a communication network among European medical centers. The system aimed to provide communal access to radiological images from a number of different centers for educational and scientific purposes. The participating image databases were again required to use the same data and query models, and in addition were required to have the same conceptual schema. This greatly reduced the control an individual institution could exercise over its own database. The research did emphasize the need for multiple hardware platform support, at least for the user interface, the ability to access database information across long distance networks and the importance of a graphical user interface as a means of communication with the database

An innovative graphical user interface to a database system known as 'dynamic queries' has been proposed in [Schneiderman 1994]. The interface is designed to allow the user to gain immediate and continuous visual feedback from queries submitted against a database. Instead of typing in traditional textual queries, the user is encourage to use sliders and buttons to adjust query parameters and the resulting change in the result set is displayed instantaneously. A real estate purchase application is proposed as an example of a use for such queries. Information about all potential properties for sale is kept in a database. In order to choose a property, prospective customers adjust the query for price, distance from workplace and number of bedrooms. Properties are dynamically added and removed from the display as these parameters are adjusted. According to the research, such applications proved much more popular than applications that required the queries to be input as text. One large disadvantage of the system was the amount of database system specific application programming that was needed, meaning that very few database systems were supported.

1.5 Objectives

These research examples serve to underline the need to consider both the technical requirements and the user interface requirements in the design of a multidatabase system for the authoring environment. On a technical level, the multidatabase system must allow convenient, integrated access to a wide range of component database systems, on multiple hardware platforms, locally and across networks. At the same time, the database interface must provide the application author with a simple interface for creating, modifying and retrieving data. As data will be retrieved from the database by the application user only when it is needed, not in advance, data retrieval must be flexible and fast to ensure a smooth running application. Once the multidatabase system is in place, storage of application objects will allow a new paradigm in application authoring to be explored. The objectives of this thesis can therefore be summarized as follows:

- Develop a multidatabase system, for use in an application authoring environment, that incorporates as wide a variety of the database systems currently available as possible. This includes:
    - Define the minimum set of functions that must be provided for each component database system.
    - Determine and provide the appropriate level of integration between databases in the multidatabase system.
- Design and implement an interface to the multidatabase system that allows the application developer appropriate access to application data.
- Provide a means of storing application objects in the multidatabase system
- Evaluate the multidatabase system by developing a variety of applications using the system.

Chapter 2

Background

There has been a significant amount of previous research carried out on multidatabase systems which must be taken into consideration during the design and development of a multidatabase component for an application authoring environment. In order to develop the multidatabase system, it is first necessary to understand the variety of data models on which database systems are constructed. Existing and proposed database system standards exist for the two most common data model types and whenever possible these standards will be incorporated into the multidatabase. Since the nature of a multimedia application authoring environment means that multimedia data will often need to be stored using the multidatabase system, the examination of previous research in the area of multimedia databases is important. The multidatabase system will need to successfully combine technologies from all these research areas to achieve its goals.

2.1 Data Models

The most important property of any database system is the data model upon which the system is built. The data model is a factor that critically affects every other aspect of the system. An important role for any multidatabase system is to operate in an environment where multiple data models may coexist. Therefore, an understanding of each of the commonly used data models, and the vocabulary associated with each model, is necessary. Historically these models have evolved over time, and will presumably continue to do so. The first widely used data models were the hierarchical and network models. These models were replaced by the relational model which remains very popular today for commercial database systems. More recently, with the advent of object oriented design and programming techniques, an object data model has been adopted by some database systems. The object-relational hybrid model attempts to merge features from both the relational and object data model. All of these data models are in commercial use and could legitimately be expected to participate in a multidatabase system.

2.1.1 The Hierarchical and Network Data Models

The hierarchical data model [Hughes 1988] was used in the earliest database management systems. Data is represented by simple tree structures. Data record types for the system are established and one record type is chosen as superior to the other types. Each record of this type becomes a root of a tree. This root record type may have any number of dependent record types which may in turn have their own dependent record types. A simple hierarchical model for a project/employee database is shown in Figure 2.1. The database holds information concerning four projects and a group of employees, each of whom works on at least one project.

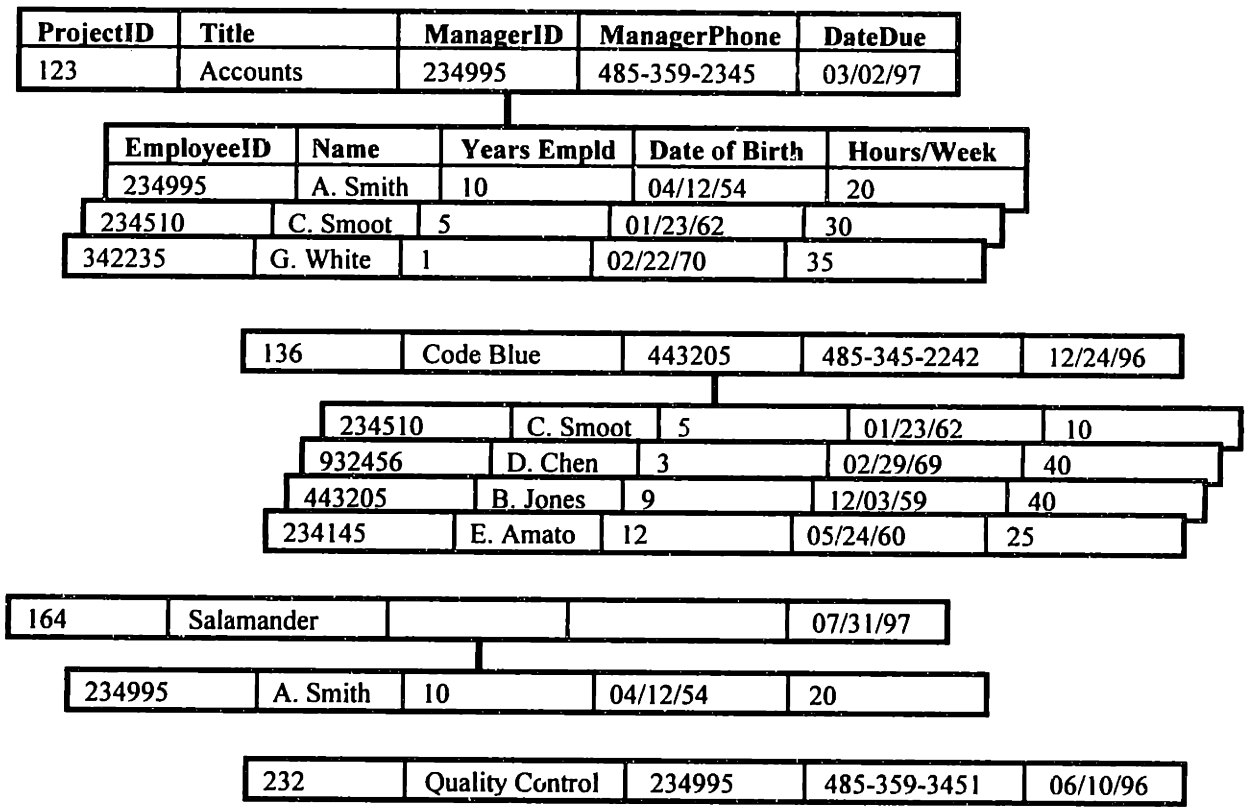


Figure 2.1 Project/Employee Database modeled using the Hierarchical Data Model

Fundamental to the hierarchical model, is the concept that any record is only given significance in the context of the tree it belongs to. Only the superior record type may exist in isolation. Dependent record types may not be created independently, but must be assigned a place in a tree. That no dependent record can be created unless it is assigned to a tree is one of the major disadvantages of the hierarchical model. Thus, in the example, no employee can exist in

the database unless he or she is currently assigned to a project. If a project is deleted from the database, then all the employees for that project are also automatically deleted. Another significant flaw in the model is that each employee exists in the database multiple times; once for each project that he or she is participating in. This leads to problems in keeping the database up to date, as multiple records must be modified when a change is made to a single employee.

The network data model was introduced in an attempt to overcome some of the shortcomings of the hierarchical model. Like the hierarchical model, the data in a network model is represented as a collection of records and links. However, unlike the hierarchical model which allows only a one to many relationship between a record and its dependent records, the network model allows a many to many relationship. Each record can have multiple links to both immediate superior records and immediate dependent records. The same project/employee database is shown modeled using the network approach in Figure 2.2.

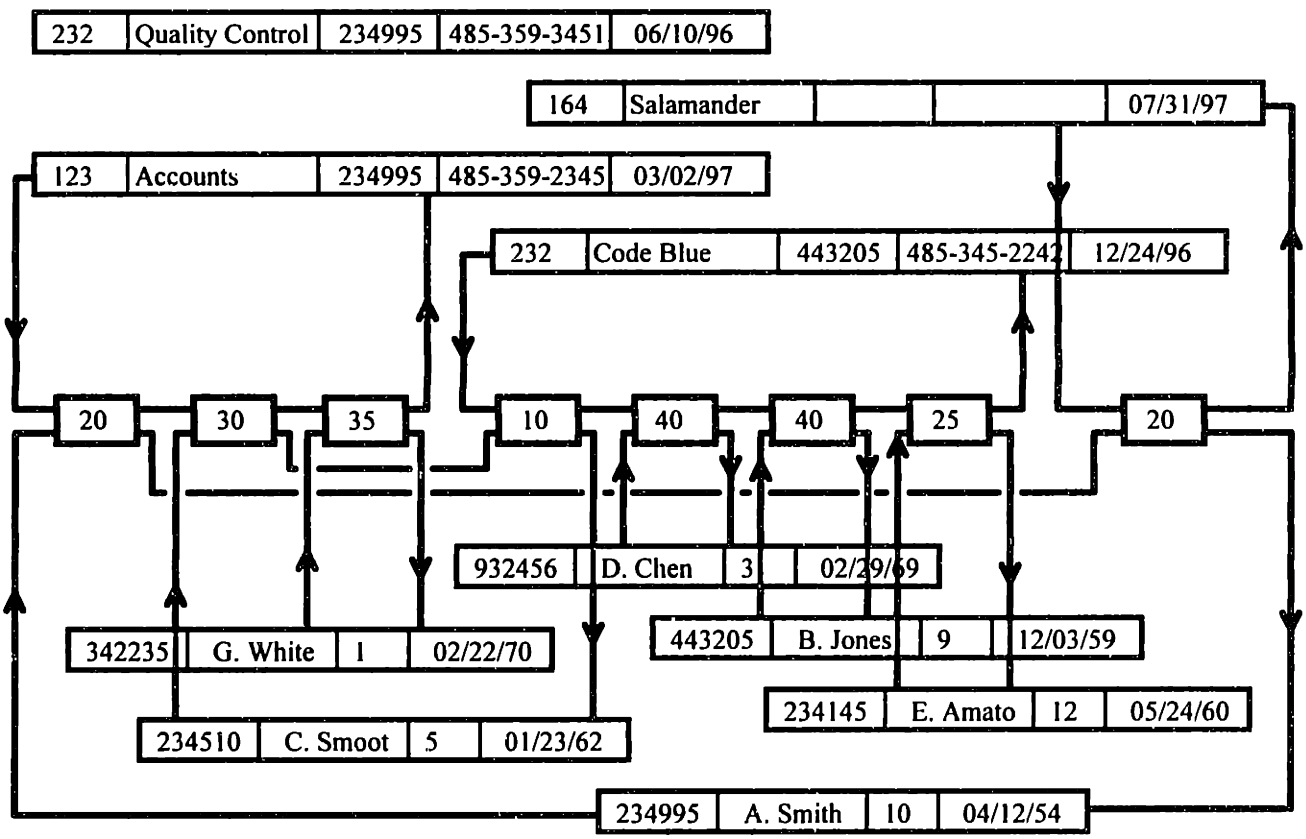


Figure 2.2 Project/Employee Database modeled using the Network Data Model

The network model [Borkin 1980] does indeed solve some of the problems associated with the hierarchical model. Records can exist in their own right, and do not rely on the existence of a superior record for validity. Duplication of records is eliminated. However, the main drawback of the model is seen in its complexity and the complexity of the queries that must be generated to answer simple questions such as finding the total number of hours worked by an employee. This complexity arises from the range of information bearing constructs, such as records and links that are in use. Accessing data is a way other than that dictated by the predefined relationships is slow and inefficient. Data modifications to both the hierarchical and network models are difficult, usually requiring the database to be shutdown before the changes can be made.

2.1.2 The Relational Data Model

The relational data model was first proposed by [Codd 1970] and remains the most popular data model for commercial database systems. The model addressed the problems associated with previous models by removing information about complex relationships from the database entirely and relying on the database query language to express these relationships [Date 1977]. The model is based on the concept of a relation; a set of ordered data tuples, such that each item in a tuple belongs to a specific data domain. This is perhaps best understood by example and the relational data model for the sample project/employee database is shown in Figure 2.3

Project

ProjectID	Title	ManagerID	ManagerPhone	DateDue
123	Accounts	234995	485-359-2345	03/02/97
136	Code Blue	443205	485-345-2242	12/24/96
164	Salamander	NULL	NULL	07/31/97
232	Quality Control	234995	485-359-3451	06/10/96

Employee

EmployeeID	Name	YearsEmpld	DateOfBirth
234995	A. Smith	10	04/12/54
443205	B. Jones	9	12/03/59
234510	C. Smoot	5	01/23/62
932456	D. Chen	3	02/29/69
234145	E. Amato	12	05/24/60
994321	F. Murphy	7	09/14/68
342235	G. White	1	02/22/70

Assignment

ProjectID	*EmployeeID*	Hours/Week
123	234995	20
123	234510	30
123	342235	35
136	443205	40
136	234145	25
136	932456	40
136	234510	10
232	234995	20

Figure 2.3 Project/Employee Database modeled using the Relational Data Model

A relation is also commonly known as a table. Each tuple within the relation is also referred to as a row. Each column, corresponding to an entry in the tuple, is known as an attribute of the table. Each column has a simple domain type, such as string, integer, real, date etc. or a subset of any of these types. Compound data types are not allowed as domain types which is perhaps the biggest disadvantage to the relational model.

The concept of a relation has a firm mathematical basis in set theory, the model can be subjected to rigorous mathematical techniques to aid in the resolution of schema ambiguities and inconsistencies and promote good schema design practices. It is usual that in a given relation there is one attribute, or a combination of attributes that uniquely identifies the tuples of the relation. This combination of attributes is known as the primary key of the relation and is used to distinguish between the tuples of a relation. There is no means of telling two tuples apart in a relation if they have identical attribute values. Hence, almost all relational databases enforce the notion of a unique primary key for every tuple in a relation. The relationships between tables are expressed in the form of foreign keys; that is the primary key data value of a row in one table is used to reference that row in another table by inserting the key value in a row of that other table.

Closely tied to the relational model is the concept of data control and data management languages which are provided with the model to allow the user to create, modify, delete and access data. The data management language is based on relational algebra which is composed of traditional set operations (union, intersection, difference) and special relational operations (selection, projection and join). The result is a simple, but very powerful, language for performing selective data retrieval update. Data control languages were also introduced to provide access control to the data, transaction management facilities and data concurrency control mechanisms. Structured Query Language (SQL) is a widely accepted standard for a data retrieval and management language for relational database systems.

2.1.3 The Object Data Model

Object-oriented database systems [Bertino and Martino 1993; Bancilhon et al. 1992] are software systems that integrate technology from the fields of database systems and object-oriented programming languages. The core concept of any object-oriented system is that each real world entity in the system is modeled as an object and the resulting set of objects communicate with each other via messages. An object-oriented database system provides a means of making these

objects persistent beyond the lifetime of the computer program in which they appear. While there is not yet a standard object model in use by object-oriented databases, certain generally accepted features of a typical model can be formed into a basic object model. Such a model includes the following concepts:

Complex Objects. Each object is made up of a set of attributes. The value of an attribute can either be a simple data type, or can be another object or set of objects. In this way complex objects can be built up. Each object in the system has a unique identity, regardless of the value of any of the object attributes.

Encapsulation. Each object also contains a set of methods with which it can be accessed and manipulated by other objects. In this way the object provides a well defined public interface through which all communication occurs and the implementation of the object behind this interface is kept hidden.

Classes. All objects which share the same set of attributes and methods are grouped together into a class. An object then becomes known as an instance of that class.

Inheritance. A class can be defined to be a subclass of one or more existing class, known as the superclass. The subclass inherits all the attributes and methods of the superclass and can then modify their behavior and composition as needed.

Polymorphism. Overloading, overriding and late binding of methods are allowed so that that the system can determine which method should be used to execute a particular operation at runtime.

The principle advantages of the object data model are object identity and extensibility. New object classes can be defined from the primitive system types and existing classes. However, the lack of a formal mathematical base and a standard for the object model has been seen as a disadvantage. Early object-oriented database systems were closely tied with an object-oriented programming language such as C++ or Smalltalk. Object-oriented database systems were closer to an extension of the programming language to provide persistence of objects than an independent system. Only recently have systems tended to develop more independence between the programming language and the database. Object databases have tended to be criticized for

poor performance and reliability, and thus there has been a reluctance to use them in large commercial applications. There is also no clear standard for an object-oriented query language to be used in conjunction with the object data model. In addition to accessing data by query, using a form of the relational query language, object data can also be accessed by navigation, exploiting object identifiers and aggregation hierarchies. The project/employee database is shown modeled using the object data model in Figure 2.4.

Database Object Classes

```
class Employee {
  string Name
  integer YearsEmpld
  date DateOfBirth
}

class Manager : Employee{
  string Phone
}

class Project {
  string Title
  Manager m
  date DateDue
  set(Assignment) employees
}

class Assignment {
  Employee e
  Project p
  integer Hours
}
```

Database Objects For Accounts Project

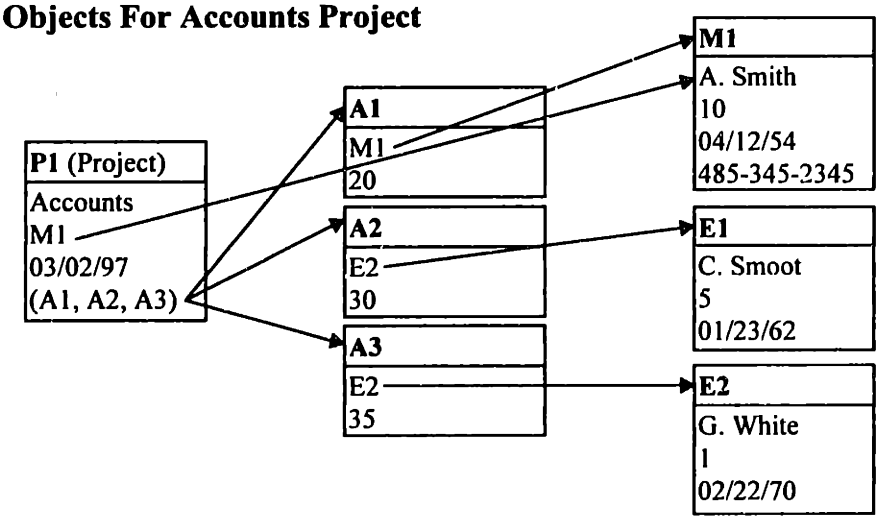


Figure 2.4 Project/Employee Database modeled using the Object Data Model

2.1.4 The Object-Relational Model

In order to overcome the disadvantage of the relational model in not allowing complex data types, the object-relational [Stonebraker 1995] or extended relational [Cattell 1991] data model has been developed. This data model generally allows user-defined types and non-tabular

structures, such as nested tuples. New features such as inheritance, methods and tuple identification may also be added. In this way the data model looks almost identical to the object model. However, by building from existing relational database systems, object-relational systems benefit from the many years of research and development with the relational model. Users can continue to make use of facilities such as view generation, automatic query optimization, transaction management, concurrency control and dynamic schema alteration. An extended relational query language is used to access and modify data. Unlike traditional object-oriented systems, object-relational databases are not tied to a particular programming language.

2.1.5 Comparison of Models

It would be incorrect to assume that each new model supersedes its predecessors and becomes the model of choice for database systems developed from that point onwards. When choosing a data model, it is important to consider the data types that will need to be stored in the system and the applications that will be developed using the system. Relational database systems are generally useful for storing simple data for which the user requires complicated query facilities. Object database systems have traditionally been better suited to storing complex data where the query capabilities of the system have been unimportant. Object relational databases have been proposed for systems that require both complex data storage and query facilities. The boundaries between the types of database systems are also becoming less clear as relational database vendors extend their system to include the definition of complex data types, including type inheritance, and object database vendors introduce improved query and interface functionalities, together with improved performance.

Database systems under current development do not generally make use of the hierarchical and network models. These models, however, remain important as they correspond closely to the model of a file system or the World Wide Web, both of which can be considered as large data repositories and should not be ignored by the multidatabase designer.

2.2 Existing Standards

Database standards are a very important issue, and one that is continually being addressed by database vendors and users. The availability of a standard interface to a class of database systems can significantly reduce the work of creating a multidatabase environment that included that class

of systems. There are four separate standards that are worth mentioning in the context of the development of a multidatabase system for an object-oriented authoring environment; the Structured Query Language (SQL), the Open Database Connectivity™ (ODBC) interface for relational database systems, the work of the Object Management Group Inc. (OMG) on the Common Object Request Broker Architecture™ (CORBA) and the Object Database Modeling Group (ODMG) proposed standard for object database systems.

2.2.1 SQL

SQL is a widely accepted standard for a data management language for relational database systems. The most recent specification is the SQL-92 ANSI standard [ANSI 1992]. The language provides a standard set of commands for the creation, update and deletion of data according to the relational data mode, as well as providing a standard interface for access protection and transaction control. Similarly, the query language follows a standard syntax using the SELECT, FROM, WHERE, SORT BY keywords. For example, in answer to the query 'Find the name of the manager for each project name' from the project/employee example database, the SQL query would be

```
SELECT title, name
FROM project, employee
WHERE managerID = employeeID
```

SQL3 is the code name for the new draft of the SQL standard [Mattos and DeMichiel 1994] that is likely to replace the current standard in 1998. SQL3 contains several object-oriented extensions: the introduction of abstract data types that will allow the definition of complex data types, a type inheritance hierarchy and function overloading. The standard is being designed to be backward compatible with the SQL-92 standard which is an important consideration for existing database systems. Object-relational database system vendors have already each implemented their own versions of object extended SQL in anticipation of the standard, but have agreed to comply with the standard once it has been approved. Generally extended SQL allows for the inclusion of object attribute paths and object methods in the query.

2.2.2 ODBC

The ODBC interface [Microsoft 1994] developed by Microsoft®, but partially based on the recommendations of the X/Open™ and SQL Access Group Call Level Interface specification, allows applications to access data in database management systems that use SQL as their data accessing protocol. The interface permits and promotes maximum interoperability so that a single application can be written that is able to access different database systems without needing to recompile or change the application in any way.

The ODBC interface defines a set of function calls that allows an application to connect and disconnect from a database system, determine the database schema, execute query statements and retrieve the results of these query statements. The query statements must be given in standard SQL, but both static and dynamic queries are allowed. Dynamic queries allow the late binding of query variables at runtime. The interface defines a standard representation for the common database types and carries out any needed conversion between native database types and the ODBC standard types. Similarly the interface will manage any inconsistencies due to small variations in the implementation of SQL between native databases. A standard set of error codes also forms part of the interface.

Typically the ODBC architecture has four components: the application, the driver manager, the driver and the data source as is shown in Figure 2.5. The driver processes the ODBC function calls, submits SQL requests to a specific data source and returns results to the application. When necessary, it is the driver that modifies an application's request so that the request conforms to syntax supported by the native data source.

The advantage to an application using the ODBC interface is that it can ignore underlying data communication protocols between it and the native database system. Data values are sent and retrieved in a standard format. The major disadvantage of the interface is that it was designed specifically for relational database systems. It is assumed that the underlying data model is relational and the application programming interface reflects this in the nature of its calls. It does not provide any extensions to allow a connection to a hierarchical or object data model that would allow the pertinent features of these models to be retained. Although ODBC drivers have been developed for some object-relational databases, it is at the cost of losing the ability to interact with the data in an object-oriented manner. Most significantly, ODBC does not provide

for any extended SQL commands nor for the translation of complex data types between the data source and the application. However, since the system does unify the programming interface to relational database systems, it may be possible to remove one level of complexity from a multidatabase system by only needing to provide one type of access interface for relational databases.

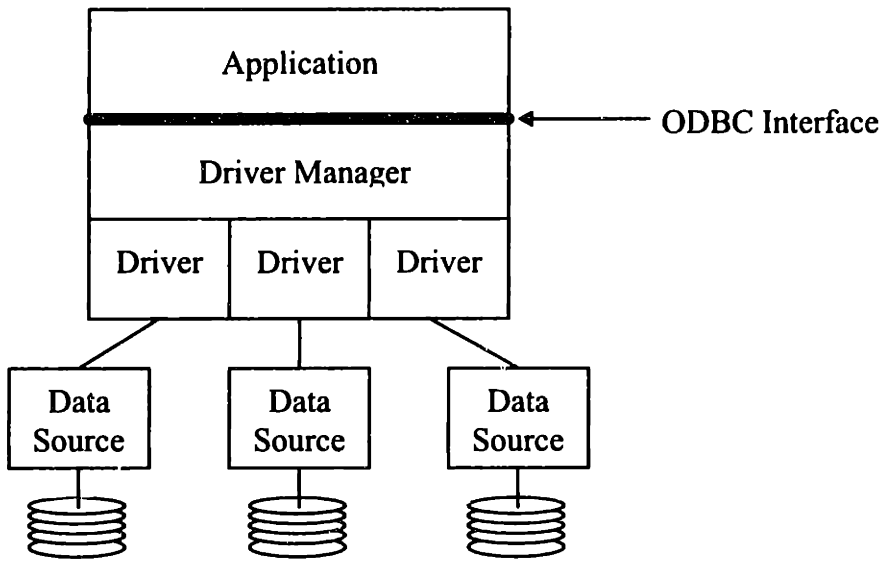


Figure 2.5 The ODBC System Architecture

2.2.3 OMG/CORBA

The OMG has been responsible for the promotion of the CORBA standard [Orfali et al. 1996] that enables distributed object programming. Components of a CORBA compliant system are objects with interfaces defined in a standard object-oriented interface definition language (IDL) that uses a common object model. Once the object is defined, a programmer can make calls to the object, using the object interface, no matter where the object is actually located in the system. The object could reside on any machine available via a network, but this complexity is hidden from the programmer by the CORBA system. Object Linking and Embedding® (OLE) from Microsoft™ and OpenDoc® from Apple™ are similar systems that provide an interface to distributed object programming.

Orbix® [Iona 1995] is one particular implementation of the CORBA standard. They have proposed an integration between the ObjectStore™ object-oriented database and Orbix which

would enable a programmer to make Orbix objects persistent by storing them in an ObjectStore database and make ObjectStore objects accessible remotely by making them CORBA compliant.

2.2.4 ODMG

The primary goal of the ODMG [Cattell 1994; Cattell 1995] is to put forward a set of standards that will allow a database application programmer to construct applications that are portable across object database systems. These standards encompass the definition of the object schema, the programming language bindings needed for object data definition and manipulation, and the definition of an object query language. This has directly resulted in the three separate, but related, components of the standard; a definition language for the proposed standard object model, the object query language, and C++ and Smalltalk language bindings for data definition and manipulation procedures.

The object data model suggested by the ODMG is based heavily on previous work carried out by the OMG. One immediate advantage is that such objects could be easily integrated into a CORBA compliant system at a later date. According to this object model, objects are categorized into types (equivalent to classes in other definitions) and all objects of a given type exhibit common behavior and a common range of states. The behavior of the objects of a particular type is defined by a set of operations that can be executed on an object of that type. These are equivalent to class methods. The state of an object is defined by the values of a set of properties (equivalent to attributes) of the object and by the values of a set of relationships between the object and one or more other objects. The fundamental data types allowed by the model are integer, float, character, boolean, date, time, timestamp and interval. Fundamental collection types are set, bag, list and array. Complex types are built up from these fundamental types. The standard includes an object definition language (ODL) in which objects can be defined in accordance with this object model.

The second component of the ODMG standard is a declarative object query language (OQL) for querying database objects. The language is based on the SQL standard but is not a superset of SQL. Most significantly, the OQL does not include facilities for the creation, modification or deletion of data from the database. No language constructs for dynamic schema alteration or view creation exist either. Like extended SQL, OQL does extend the query language so that attribute paths and method calls can be included in the query. The OQL does introduce the concept of

typing query results. In SQL query results are always returned as a relation. In OQL the result can be any valid database type, simple or compound. Common sense dictates that it is in the best interest of both the application developers and the database vendors to ensure that the OQL standard and the SQL3 standard are merged to produce one coherent standard for an object query language. The two standards bodies involved are in communication with each other, but the outcome of these negotiations remains to be determined. The object model used by ODMG is more advanced than that proposed by SQL, including as it does the concept of class methods and relationships, in addition to class attributes, and this makes SQL3 an unacceptable direct substitute for OQL. It has been suggested [Kim 1994] that SQL3 could profit by adopting the OQL system of managing compound data and the idea of strong typing of query results.

Since the aim of the ODMG standard is to provide a single, integrated environment for programming and data manipulation in a manner that is portable across object database systems, the C++ and Smalltalk language bindings proposed are seen as the most important aspect of the standard. The bindings specify how the object model can be defined in that particular language and the methodology for creating and deleting persistent objects and updating the state of an object. In common with many object databases, the questions of runtime schema identification, retrieval and modification are ignored so that no standard interface for these features is considered. This greatly reduces, if not completely destroys, the suitability of the interface for use in a multidatabase system. In a multidatabase system, the system would need to be recompiled every time a component object database was added, deleted or modified in the system, usually an unacceptable situation. In addition, the ODMG standards have not yet been widely implemented by object database vendors, unlike the ODBC standard which is available for all major relational database systems.

2.3 Multimedia Databases

The number of requests for multimedia information systems has been growing rapidly in a huge variety of fields from business to education and from entertainment to engineering. Multimedia data includes images, audio, video, text and any combination of these types of data. It is popular because it provides information with a much higher semantic content than is possible with purely character based interfaces. Applications that contain multimedia data can portray information in a

manner that is easier and faster to comprehend, and the applications themselves can become significantly easier to use. Improvements in computer hardware and in display technology have made such applications generally affordable. Due to the increase in demand for multimedia data for use in applications there has been significant research carried out into optimal methods for storing and retrieving multimedia data. This research has led to a class of databases known as multimedia databases that have been especially adapted or created to store multimedia data [Ghafoor and Berra 1995].

There are four important issues, proposed in [Ghafoor 1995] that are faced by the developers of multimedia database management systems.

1. The development of a data model capable of appropriately representing multimedia data.
2. The design of a query language that enables the intelligent retrieval of multimedia data from its semantic content.
3. The design of indexing or other data organizational techniques.
4. The development of efficient storage techniques for the data.

A multidatabase system must be primarily concerned with the first two of these issues. Organizational and storage techniques [Buddhikot et al. 1994] are generally determined by the underlying component database and are outside the sphere of influence of the multidatabase. Obviously, for the best performance within the multidatabase, a component multimedia database that can take advantage of the latest techniques in high speed data storage and retrieval will be an advantage.

The ideal multimedia data model should be able to model the semantic complexities of multimedia, encompassing the different elements of video, still image, audio and other data. The model should be capable of representing the spatial and temporal relationships between data elements to ensure correct data synchronization [Little and Ghafoor 1991]. For example, a video clip may have an accompanying audio track or text subtitles. A collection of images may be arranged so as to form a larger mosaic. This spatial synchronization is particularly important in computer aided engineering and drawing applications. Typically relational database systems have resorted to creating one extra database system type, the binary large object, to cope with

multimedia data. This type is simply a means to store a fixed size block of binary data and cannot distinguish between types of multimedia. The relational model also makes it very difficult to store more complex semantic information about the data such as synchronization constraints and to standardize the capture of this information across database systems. The object data model provides a much better means of modeling multimedia data [Browne 1993]. Object classes can be created for each type of multimedia data and common data features can be abstracted into an inheritance hierarchy of data classes. Each object can contain temporal information that can be used to calculate precedence and synchronization between objects, or container objects can be specified that model the spatial layout of a collection of objects. In addition to the object model, graphical models, pert-net models and language base models have also been suggested as suitable models for multimedia data. However, with the advent of object database systems, most multimedia database systems have chosen to use the object model.

A typical multimedia application user might like to query a collection of images in a database in the following way: 'Please give me the titles of all images that feature a sunset scene'. Unfortunately, for the database system, this is a non-trivial task [Yoshitaka et al. 1994]. There is no direct way to tell if the image is of a sunset from the binary data form it is most likely stored in. Traditionally databases have relied on a set of key words, free text, or a title that is entered manually for each image in order to match a user's request. Obviously this method is unreliable, depending as it does on a limited interpretation of the image carried out when the image is first placed in the database, and cannot hope to fully anticipate the needs of future users. Some of the most successful attempts to develop an intelligent query language for multimedia data have taken advantage of the object model. Object methods can be defined on media data types that can determine the answer to whether, for example, an image features a sunset, and these methods are easy to incorporate into the extended query language used by object databases. In a multidatabase system it is possible that such media processing facilities could be optionally provided at the multidatabase system level, dramatically increasing the capabilities of component databases.

2.4 Multidatabase Systems

Information is a key resource for organizations in the world today, vital to the continuing function of business, government and education. A significant proportion of this information is

stored in a huge variety of database systems that were originally designed to be independent islands of knowledge. Today, with the advent of network technology, application users are demanding simultaneous integrated access to these databases. Unfortunately the multitude of existing databases each have their own access protocol and storage paradigm. While it is unreasonable to expect a user to learn the data access method for each individual database, it is also unreasonable to expect an organization to immediately convert all its database systems to a common standard, even if such a conversion were possible [Ram 1991].

A multidatabase system is one solution to the problem of shared access to heterogeneous data resources, and it is expected to become a key component in distributed global technology systems in the future [Hurson et al. 1994]. A multidatabase, as shown in Figure 2.6, allows access to pre-existing databases by providing the user or application with a common way of accessing and integrating the data in these participating component databases.

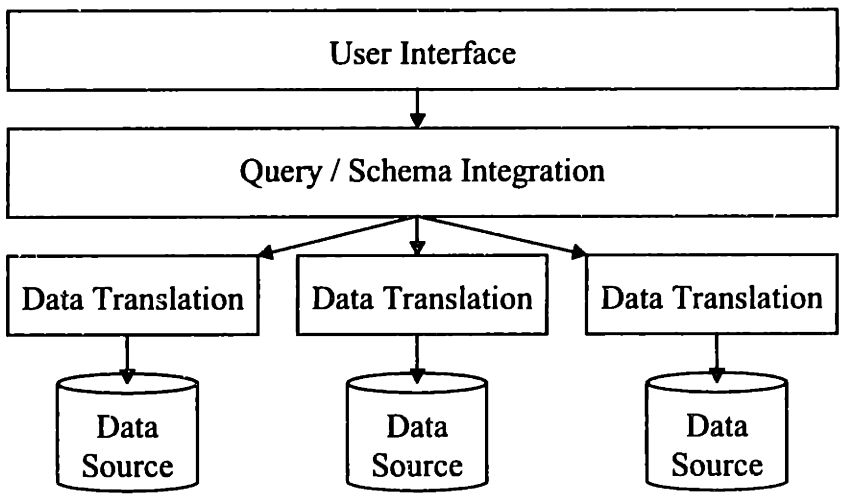


Figure 2.6 A Typical Multidatabase System

2.4.1 Types of Multidatabase Systems

A wide variety of multidatabase system designs have been proposed, each focusing on different objectives and providing different functionality to the system user [Bright et al. 1992]. The important distinctions between the systems are generally to be found in the autonomy allowed to the component databases, the level of heterogeneity allowed among the component databases and the existence of a global schema or query language that serves to integrate data from the

component systems. On this basis, multidatabase systems can be divided into five categories according to the services they offer: distributed databases, global schema multidatabases, federated databases, multidatabase language systems and interoperable systems [Sheth and Larson, 1990]. In reality a multidatabase system falls somewhere along the continuum from distributed database to interoperable system. Also, the terminology used in multidatabase systems is not exact. Words such as federated, distributed and multidatabase are used interchangeably in the literature so that it is not always possible to accurately discern the features of a system from its description.

A distributed database is very closely coupled to its component databases. The global and local databases are often designed and built in parallel. The global system is given access to low level local database functionality and is given control over local database processing. The local databases are usually homogeneous (at least with respect to the data model chosen) and although they may be implemented on different machine types, they provide the identical interface on each platform [Jeffery 1994]. The global system will maintain a global schema, found by integrating the local database schemas, and the global user will interact with this global schema, often unaware that the data is coming from distributed local databases. The tight coupling of the local and global systems enables the efficient processing of all global database operations. However, the cost of this tight coupling is the loss of local database autonomy, and the local user becomes subject to global control. Indeed, in such systems the number of local database users is often very small or non-existent.

Global schema multidatabases are more loosely coupled than distributed databases because global functions access local information through the external user interface of the local database system and have no privileged access to internal database functionality. It is the responsibility of the multidatabase system to integrate the schemas of the component database systems into a coherent global schema to be used by the global user. This global database [Lee and Moon 1993] has no control over the modifications made at the local database level, but must incorporate these changes into the global schema regardless. The local databases may be heterogeneous, that is they may not share a common data model, in which case the task of schema integration becomes much more difficult. The advantage of global schema systems is that they can be more readily designed to incorporate pre-existing component databases without any need for modification. In

return, the systems sacrifice the processing efficiency of the tightly coupled distributed database system. The successful integration of the data is determined by the construction of the global schema and so this is the central problem tackled by research on these type of systems. Queries against the global schema are automatically broken into component queries against the local databases by the system and the results reassembled into a global result.

Federated database systems are a still more loosely coupled subset of global schema multidatabase systems. No single global schema is maintained. Instead, each participating local database maintains an export and an import schema. The export schema is a description of the information the local database is willing to share with the global system. The import schema is a description of the information that may be accessed locally from the global database system. In this way the amount of data information that must be integrated at any one time is reduced and the complexity of the global system is lessened. There is no concept of the global database user. Instead, the local user is able to access all data from the local database and a subset of information from the other databases participating in the federation.

Multidatabase language systems are considered to be even more loosely coupled, as no global schema of any kind is maintained. Instead, the multidatabase supports global database functions by means of a global query language. In this way local database users can query for information from any combination of the participating databases. The global system provides a global name space and special functionality for translating data between the different database models that may be present in the system. Local database system autonomy is guaranteed and the global system has no control over the local systems. Like global schema databases, multidatabase language systems are designed to integrated pre-existing, heterogeneous database systems without modification. The absence of a global schema shifts the burden for data integration towards the user who must specify the global queries and away from the multidatabase system.

Interoperable systems are the most loosely coupled of all information sharing systems and the global functionality of such systems is limited to simple data exchange. The global system does not necessarily exhibit any database functionality at all. One important consequence of this deficiency is that these systems can more easily share information between traditional database systems and other sources of information such as expert systems, file systems and knowledge

based systems. The main role of the global system is to provide data translation and access protocols.

2.4.2 Issues in Multidatabase Systems

There are a number of important issues that must be understood and resolved in any multidatabase system in order for it to function correctly. A brief overview of these issues and possible proposed solutions is given here.

Site autonomy. The key difference between distributed database and multidatabase systems is that in a multidatabase, each local database system maintains complete control over local data processing, whereas in a distributed database system the global system imposes control over a local site. In an autonomous multidatabase system, each local database determines when to enter and leave the global system, how its local data will be presented and which of the global data requests it will service. No global changes, such as the addition or loss of a component database, affect the local database. Local database autonomy may be critical in some organizations that have invested heavily in existing hardware, software and user training which makes the modification of existing databases impossible, especially where those databases service a critical business function. In addition, site autonomy enables a local site to protect portions of data by not making it available to the global system. The disadvantage of local autonomy is that it places a much greater burden on the global system which cannot rely on the local system to be always present or cooperative. Global optimization of queries becomes significantly harder when the local databases give priority to local optimization.

Semantic heterogeneity. There are many ways to model a real world object in a database, but in a global database system, it is preferable for each object to have only one common representation. Each local database, developed independently, is unlikely to conform to the common representation, and so the global system must provide translation functions between the local data representation and the global data representation [Bright et al 1994]. The simplest difference between representations is name difference, when the same data item is referred to by different names in the local databases. The global database must choose a standard name for the item, recognize the existence of the item in each local database, and maintain a global mapping

between the local and global names of the item. Format differences include differences in data domain type, scale and precision. For example a part number could be defined as an integer in one local database and as an alphanumeric string in another. In this case the global database normally provides translation functions to the global representation. However, the translation from global to local database format may be non-trivial. Structural differences mean that the data object may be structured differently in different local databases. For example, an address in one local database may be represented as a string of characters, whereas in another database it may be represented as a series of three character strings, each representing one component of the address; street, city and zip.

Data model heterogeneity. Multidatabase systems must be able to translate between the local and global data models and the local and global data access language [Won and Jungun 1991]. Typically, a relational or object model is chosen as the global data model. Supporting heterogeneous systems is often seen as a trade-off between supporting a particular system and the amount of translation code that must be provided by the global system. Often, functionality missing at the local database level must be provided by the global system, while extra functionality found at the local database level is unavailable to the global user.

Global query processing. The basics of global query processing are consistent across most multidatabases. A user employs the global schema or global query language to submit a query, and this query is decomposed into a set of subqueries by the global system. The act of decomposition also includes any data translation that may be necessary. Each subquery is executed by a local database and the results are merged by the global system for presentation to the user. A significant amount of research has been carried out into the process of global query optimization, which aims to structure query parsing, decomposition and results recombination so that the communication and processing costs are minimal. Multidatabases benefit from the parallelism inherent in the processing of subqueries at local database sites. In addition, each local database subquery can be individually optimized for that particular database. If several participating databases contain the same information, then the subquery can be redirected to the closest local databases or the local database with the best retrieval performance.

Global constraints. Different local databases may represent semantically equivalent or related data and so the global system must have some method for specifying and enforcing integrity constraints on interdatabase dependencies and relationships [Rusinkiewicz 1991]. For example, consider a situation in which a list of employees, together with their identity numbers and addresses is kept in one local database and a list of identity numbers, together with corresponding addresses and telephone numbers is kept in another. In a multidatabase system that includes both these systems, if an employee's address is updated in one of the databases, then it should automatically be updated in the other as well. However, such automatic updates may violate a requirement for local database autonomy, if the databases were participating in a read only manner in the multidatabase system. In this case, the global system must decide whether to accept the update, violating global data integrity, or to refuse to carry out the update. The global constraints on a system are often created and maintained as part of the global schema.

Concurrency control. Concurrency control is the coordination of concurrent processes that operate on shared data that may interfere with each other. Typically, database systems use a lock on an item of data to indicate that the item is potentially being modified and other processes must wait before accessing that data. The traditional concept of a transaction is a series of reads and writes on the database system, utilizing the internal lock mechanism of the database, that occurs in a short time frame and either completes successfully or fails and leaves the database unaltered. In either scenario, the database is left in a consistent state. However, the traditional methods of concurrency control do not translate well into a multidatabase system. Multidatabase transactions tend to need to be spread out over a longer time, both because communication and coordination between component databases takes significantly more time than in a local database and because multidatabase systems, such as a computer aided design application where a design may be worked on for a significant period of time by one user, tend to be inherently longer. Multidatabase system queries may not need to be atomic. It may be acceptable for a portion of the query to succeed and for another portion to fail if appropriate notice is given to the user. Obviously the global constraints imposed on the system also make concurrency control significantly more difficult. Since the concurrency control in a local database is often managed internally, without providing an interface to the normal user, it can be hard for the global

multidatabase system to control transactions in a component database. Also, each database may have its own, proprietary, control mechanism so that the global system needs to be able to work with a wide variety of concurrency control paradigms which may be incompatible. A wide variety of solutions to the problem have been proposed, from the simplest, restricting multidatabase systems to being read only, to more complicated schemes that involve tree graphs for coordinating local locking mechanisms, to methods of relaxing the restrictions on serialisability and atomicity for global transactions.

Security. The requirements of security are often fundamentally at odds with the purpose of a multidatabase system, which is to share information. Distributed system security is difficult as additional steps must be taken to ensure the security of the communication links, to check the security at each of the participating local sites and the validation of all global users. Typically multidatabases have relied on the security of the underlying hardware and system software to meet their security requirements. Local database autonomy helps in this case, as the local database administrator still retains ultimate control over the access to local database information. Typically a global user will need to pass some form of security identification test, usually a username and corresponding password, in order to include a local database in the multidatabase system.

2.4.3 Data Models in Multidatabase Systems

The choice of global data model has a major impact on the power and ease of development of a multidatabase system. The global data model must be expressive enough to capture the meaning of all component databases that are, or will in the future, be present in the global system. It must be possible to translate between local and global data models with relative ease. In addition, it must be possible to translate the data access methods associated with the local model to and from the global access methods. The global data model must also provide the global data types to which local data types can be mapped.

When multidatabase research was first begun in the 1980s, the relational model was the most expressive and powerful model that had achieved widespread acceptance. It was therefore a natural choice for a global data model. Correspondingly, the global data access model was based closely on SQL, which had become the standard access language for relational database systems.

Translation between local relational database models and the global relational model proved straightforward as there were usually only minor differences between the models and data types employed by the different component systems. Research tended to concentrate on schema integration techniques and on the optimization of global query processing [Salza 1995; Lee et al. 1995].

However, since the early 1990s, the object data model has been receiving more and more attention among researchers into multidatabase design [Bukhres et al. 1996; Pitoura et al. 1995]. Multiple hierarchy class structures and inheritance allow the object model to capture more of the semantics and complexities of data entities than the relational model. Class methods and polymorphism enable a rich set of functions to be applied to data objects and give designers significant freedom in defining the function of the global data model. The ability of objects to encapsulate their implementation means that the object model is very good for providing translation services to and from other data models. Complex types are easily introduced into the object model to handle, for example, multimedia data. Object models are also more useful for integrating component systems where the information is not stored in conventional database systems. For example, weather information systems, airline reservation systems and library catalogue systems may use a proprietary data model designed for their special needs.

In a heterogeneous system that consists of component object database systems and component relational database systems, a global object model is the natural choice. If the component object database systems had to be translated to a global relational model, most of the object functionality would be lost and unavailable to the global user. It is much simpler to conceive of the relational data model as a simplified subset of the object model, although much more significant differences are encountered when integrating data access protocols. One major drawback of the object model in this situation is the lack of widely implemented object database standards, especially in the area of object data access languages.

2.4.4 Review of Existing Multidatabase Systems

Although many research prototype multidatabase systems have been developed, the number of commercially available systems remains close to zero. The cross section of systems considered here represents the wide range of functionality and innovation that can be found in multidatabase system design.

Initial Systems. As described previously, almost all multidatabase systems were initially based around the relational data model and served to integrate component databases that also followed the relational model. Most early prototype systems, such as Multistar [Litwin et al. 1990], attempted to create a global schema that the global user could query against. However, the global schema provided only minimal help in resolving schema interdependencies. The definition of the global schema was typically not done automatically, but by the manual definition of global database views across the component databases. Alternatively, systems such as Calida [Jakobson et al. 1988], concentrated on the development of a proprietary multidatabase language that made use of database names to resolve naming conflicts and even allowed updates as well as elementary queries. Initial commercial offerings, from Sybase™, Ingres™, and Oracle™, were based on the distributed database model and did not integrate pre-existing databases into the multidatabase system. However, such systems did allow users to access a distributed system of homogeneous databases on a variety of hardware platforms and to define queries and views that spanned this set of databases.

Pegasus. Pegasus [Ahmed et al. 1991] is a heterogeneous multidatabase management system that was developed by Hewlett-Packard Laboratories and represents one of the most effective attempts at implementing a complete multidatabase system. Pegasus takes full advantage of object-oriented data modeling and programming techniques, using abstraction to deal with mapping and integration problems. The system is composed of native Pegasus databases as well as providing support for a small variety of external data sources.

The Pegasus object data model has three basic constructs: named types which represent collections of objects with common characteristics, uniquely identifiable objects which hold data values, and functions which provide mappings between objects as well as between objects and the underlying component data structure. Subtyping (similar to class inheritance) is allowed. The system provides mapping facilities that creates a Pegasus schema for all underlying component databases that give these external systems the appearance of a Pegasus database. The user can interact with these databases in the same way that he or she would interact with a native Pegasus database. In addition Pegasus uses a unified data definition and data manipulation language called Heterogeneous Object Structured Query Language (HOSQL). This language provides the

ability to query data across databases and is used to create types, functions and objects in Pegasus and the underlying databases. The language can also be used to import types from underlying databases so that the types can be integrated into a partial global schema. This global schema relies on user interaction to solve domain and schema mismatches and to specify object equivalence.

In general the Pegasus system can be divided into three functional layers: a local data access layer that manages schema mapping; data translation and local query processing; a cooperative information management layer that deals with schema integration and global query processing; and an intelligent information access layer that provides user services such as information mining and data and schema browsers. Pegasus is an important system in several respects; the use of an object data model, the introduction of a way to cope with schema and data conflicts and the use of global query optimization techniques. However, although an object data model is employed, the component databases themselves are still generally assumed to be based on the relational model. A way of automatically importing relational databases to the system was also developed in [Albert et al. 1993].

Other Object Based Systems. ViewSystem [Kaul et al. 1991] is a multidatabase system that also uses an object data model. Integration between component databases is done by defining virtual external and derived classes. External classes are the direct representation of data in a component database. Derived classes are formed using special class constructor definitions from a number of external classes and enable an integrated global view of the data. Similar to this technique, integration of database systems by defining views through queries is found in many multidatabase systems such as GTE's distributed object management system [Manola et al. 1992] and Xerox's engineering information system [Pathak et al. 1991]. An alternative integration technique is used in the Comandos Integration System [Bertino et al. 1989] which concentrates on defining correspondences between operations, rather than between data elements. A set of operations is defined on a local database, and all global multidatabase functions interact through this set. Other systems such as Object View Broker [Durand et al. 1994] and InterBase [Buhkres et al. 1993] do not attempt any global integration of data at all, simply allowing the user access to component databases in a standardized object-oriented manner. While all these systems do use

some form of object model as their common database model, little or no attention is paid to the mechanics of data translation from the local model to the common model. Furthermore, almost all the systems imply that they expect the local data model to be relational.

The Jupiter system [Murphy and Grimson 1994] is unique among multidatabase systems in that it takes advantage of the CORBA distributed object management services in its implementation. The global data is represented as a collection of distributed objects, and the CORBA implementation enables the system to take advantage of transparent networking capabilities, as well as data replication facilities in a heterogeneous environment. Jupiter introduces the concept of a negotiation protocol, where the user can request either direct access to federation data or can opt to work with a local copy of the data. In the latter case, conditions are specified as to how out of date the data can become before it must be refreshed by the system.

Commercial Systems. UniSQL/M® is a heterogeneous database system being developed at UniSQL Inc., that allows the integration of SQL-based relational database systems and the UniSQL/X® [Kim 1994] object relational database system. Its importance stems from its commercial availability. The common data model used is the object-relational model used in all UniSQL databases. Since this model is a superset of the relational model, model translation is straightforward. A multidatabase query language, SQL/M, an extension of SQL, is supported. Database integration is carried out according to an individual user's specification using the extended query language to define virtual class views over the existing classes. The language also provides a means of resolving schema and format conflicts that arise in the definition of these virtual classes. Unfortunately the system only provides automatic support for a limited number of popular commercial relational databases and so will not be suitable for all situations.

This survey of existing prototype multidatabase system did not reveal an immediate match to the needs of an application authoring system, due to both the lack of appropriate component database support and the lack of in-depth consideration of a sophisticated object-oriented user interface for the systems.

Chapter 3

The Common Data Model

The first step in developing a multidatabase system suitable for inclusion in a multimedia application authoring environment is to choose a common data model and associated common query language. Following directly from this choice, the issues arising from the need to translate each component database system model to and from the common model must be addressed. The required level of database interoperability must also be determined and then implemented. This chapter explains the theoretical foundations for the multidatabase system developed for the AthenaMuse® authoring system.

3.1 The Choice of a Common Data Model

The choice of a common data model for the multidatabase system will greatly influence the ease with which certain component database systems can be integrated into the system. The choice will also affect the design of the user interface for the system. Therefore, it is very important to elucidate the end requirements and expected use patterns for the system before a final choice is made.

3.1.1 System Requirements

The multidatabase system will consist of component databases representing at least the three main database types; relational, object-relational and object-oriented. The common data model must be able to capture the essential components of each of these different data models. In addition, the common model must be able to capture the semantics of specialized component data sources, such as file systems. The model must also be extensible so as to allow additional component database types to be added later.

The common data model must provide a way to adequately represent complex application data, in particular multimedia data. The AthenaMuse® authoring tool is object-oriented, and so

the chosen data model must fit comfortably into this environment. It seems natural to consider an object data model as the common model in these circumstances. The object model is the most advanced of the component data models under consideration and has already been shown to be appropriate for modeling complex data [Booch 1994].

It is necessary for the component database systems to maintain complete autonomy. Often, applications are designed to allow multiple users simultaneous access to large amounts of sensitive data. In such cases, in order to ensure the data integrity of the system, the application user will generally be allowed only read access to the database. Therefore, it must be possible to translate from any component system data model to the common model without altering or adding to the component database itself. Since the component database may already exist and be in use within other application environments, relying on cooperation from the component database system administrator in modifying the database schema in any way is not a viable option.

Within the context of a multimedia application environment, the user of a multidatabase system is primarily concerned with data retrieval. In order to carry out this data retrieval intelligently, the user needs to be able to obtain information about the component database schema, which describes the organization of data within the database. This knowledge is needed to construct meaningful data queries. Therefore, the ability to retrieve schema information and data values is a high priority for the multidatabase system. The typical user is less concerned with the ability to input large amounts of data into the system or the initial construction of a database schema. This may be done more simply and efficiently by the component database system administrator using a direct connection to the database. However, the multidatabase user may still wish to perform selective data updates and schema alterations, and so this functionality must be provided for in the multidatabase system.

3.1.2 An Object Model as the Common Data Model

One of the main features of object-orientation is the separation of interface from implementation. In addition to its ability to model complex data, this makes the object data model eminently suitable as the common data model,. The access protocol to the common model can remain constant, while the implementation of the model varies according to the component database system involved. In this way, the incompatible internal details of component systems are kept

hidden by the principle of data encapsulation. The object model can take advantage of inheritance to define a class hierarchy that ensures the appropriate translation procedure is carried out for each of the different types of component systems. Classes can be specialized to carry out the translation process for relational database systems, and then further refined to carry out the translation for a particular relational database. The object model is flexible enough to allow continuous development of the model without necessitating alteration of the base classes of the hierarchy. This flexibility also makes it possible to translate other, more rigid data models to the object model. The reverse procedure would be significantly more difficult if the multidatabase user is to be allowed full access to the advanced features of object database systems.

The ODMG-93 standard has already proposed a common object data model for use in all object database systems. Compliance with database standards is an important way forward for multidatabase systems, and so it is crucial to at least consider this model as a guide. Use of the ODMG object model as the common model should greatly alleviate translation impedance between component object databases, which are expected to move towards compliance with the standard in the near term future, and the common model. It could also mean that the multidatabase system itself would be considered an ODMG compliant system and implies probable compatibility with CORBA distributed object management systems.

The ODMG object model can be summarized as follows:

- The basic modeling primitive is the object.
- Objects can be categorized into types. All objects of a given type exhibit common behavior and a common range of states. A type itself has the properties supertypes, extent and keys. A type inherits all the functionality of its supertypes, and then adds to or modifies this functionality. The extent of a type is the set of all objects of that type. The keys of an object specify which attributes of the type can uniquely identify an object of that type. This is in addition to the unique identifier automatically associated with every object. The keys property may be empty.
- The behavior of an object is determined by the set of operations that is defined for the type of that object.

- The state of an object is defined by the values the object carries for a set of properties. These properties are either attributes of the object itself, or relationships between the object and one or more other objects.

The interface definition for a type must therefore include the supertypes and keys for the type, as well as the specification for the interface to the operations defined on the type and for the attributes and relationships that characterize the objects of that type. The user interacts with database objects solely through the appropriate type interface. The implementation of the interface does not concern the user.

The common data model chosen for the multidatabase system is a slightly simplified version of this ODMG model and uses alternative terminology for some components of the model. Again, the object is the central construct of the model. Each object is an instance of one particular class (a class is equivalent to a type) and objects of the same class exhibit similar behavior. A class definition has four components:

- A set of superclasses from which the class inherits behavior. A class may have zero, one, or multiple superclasses. The class is then termed a subclass of each of these superclasses.
- A set of attributes which define the state of an object of that class. Each attribute has an attribute name and an attribute type associated with it.
- A set of method definitions (operations) which are available for that class. These methods can be executed on all objects of the class. A method definition consist of the name of the method, the return type of the method, and the number and type of each parameter that must be supplied when the method is executed.
- A set of keys that determines the combination of attributes by which the object can be uniquely identified. A class can have zero, one, or multiple keys.

It is not necessary to explicitly include the extent property in the class definition. The extent of a class can be found by querying the database, and the multidatabase system will provide a

means by which this query is automatically constructed and executed so that the extent is easily accessible to the user. This is simpler and less error prone than requiring the system to track the extent of all classes continuously, especially in cases where the component database system does not directly provide this functionality.

The ability to define relationships on the object is also not included in the model. The purpose of a relationship is to specify an object or set of objects that is somehow connected to the initial object. There is then an inverse relationship that exists between these objects and the initial object. As an example, assume that two object classes, one representing a teacher, and one representing a lesson, have been defined. In the teacher class, a one-to-many relationship *teaches* could be defined between a teacher object and a set of lesson objects. The inverse relationship in the lesson class is *taught_by* which is a one-to-one relationship between a lesson object and a teacher object. Each relationship is made up of four components, the name of the relationship, the type of object involved in the relationship, whether the relationship is to an individual object of this type or a set of such objects and the name of the corresponding inverse relationship. The practical reason for excluding relationships from the common model is that no anticipated component database system currently implements object relationships. There would be no immediate benefit to including relationships and no way in which to test this aspect of the model. It is anticipated that the ability to define relationships could easily be added to the model at a later date by providing a data structure to store the components of the relationship and providing operations to access each relationship using its name.

3.1.3 Definition of Common Data Types

A key component of any database system is the set of fundamental data types from which class attribute and method definitions are built. An allowable data type in one of these definitions is either a fundamental data type or an existing class. The multidatabase system must define a set of common data types and is then responsible for translating the fundamental types of the individual component database systems to and from the common types. The common data types for the multidatabase are defined in this section.

There are four common atomic data types. An atomic type is one which does not need an explicit create operation defined for it. All instances of an atomic type implicitly pre-exist and have unique identity.

integer

Corresponds to a long integer and is represented with at least 32 bits.

real

Corresponds to a double. Maximum and minimum values are platform dependent but should accommodate a range of at least 10^{-38} to 10^{38}.

string

Represents an ASCII string. The maximum length of the string is also platform dependent but should be at least 65535 bytes.

boolean

Must have either the value TRUE or FALSE

There are also eleven common data types in the form of predefined object classes. Each class has a published interface through which the user can manipulate object data. The implementation of each class is encapsulated so that translation between component database types and the common type representation is hidden.

date

Provides a simple representation of a date consisting of a designator for the day, the month and the year. Class methods are provided for simple arithmetic involving dates.

time

Provides a representation of a time consisting of a designator for the hour, the minute and the second. Class methods are provided for simple arithmetic involving times.

timestamp

A means of jointly specifying a time and a date.

monetary

Provides a representation of a monetary value together with its currency specification.

set

Models an unordered collection of data types that allows duplicates. The elements of the set must be of the same data type, or belong to the same class hierarchy. This is in line

with the constraints placed on collection types in the ODMG-93 and draft SQL3 standards.

sequence

Models an ordered collection of data types that allows duplicates. Similarly to the set data type, the elements must be of the same base type.

binary

Models a fixed length block of binary data.

media

This class inherits from the binary class and is used to specify binary data as media data. This would include binary data that is used to store image, video, audio or text data types. The class includes an optional field to specify the media type.

image

This class inherits from the media class. It provides additional methods for image processing and access.

Image is the only media class currently implemented for the multidatabase system. However, it is expected that video, audio and other media types will be added as the system is expanded. Meanwhile, the binary type is sufficient for the needs of most application designers.

3.2 Data Model Translation

The next step in implementing the multidatabase system is to provide a methodology for translating each component data model to and from the common model. In keeping with the object model, the multidatabase user explores a database schema by examining the database class hierarchy and then the properties of individual classes within this hierarchy. The translation process is responsible for creating a class hierarchy that adequately represents a component schema. Once the hierarchy is established, the translation process is also responsible for providing access to the data stored within the database. This data is presented to the user as object instances belonging to a particular class. Where necessary, type conversion from fundamental component data types to common data types must be carried out. All these

translation operations are transparent to the user who uses the same interface to access schema information and data objects for all component systems.

Some component systems do not support notions such as the definition of operations on data types or keys. Some systems do not support the full range of fundamental data types allowed in the common multidatabase system. In some cases, the multidatabase is able to make up for these deficiencies by providing supplemental data translation capabilities between the multidatabase system and the component database. In all cases, such deficiencies must be treated gracefully and should not interfere with the operation of the multidatabase.

The translation process between an object-oriented or object-relational database and the common data model is addressed first. As might be supposed, this case requires the least translation effort. Next, methods are proposed for translating from the relational and hierarchical data models to the common model. General translation techniques are suggested for each type of database system. Small changes in the process may be necessary for individual databases, or some databases may require the combination of techniques from different categories.

3.2.1 The Object Data Model as a Component Model

Object-oriented and object-relational database systems share almost identical data models and can be treated equivalently for the purposes of translation to the common model. In both systems the database schema is composed of a class hierarchy, with each class consisting of at least a set of attributes. Each class in the hierarchy is imported directly, using a one-to-one correspondence, as a class in the common model. The attribute, method and superclass properties for each class are copied directly from the component system. Where necessary, references to component system fundamental data types are replaced by references to one of the common model data types. Some object database systems distinguish between a class and a tuple data type. A tuple is a structured data type, defined by a set of attributes, each attribute having a name and data type. An instance of a tuple is a set of values, one value for each attribute. In the common data model, a tuple is interpreted as a class for which only the attribute property is used. The class has no methods or other properties defined for it. An instance of a tuple type is then interpreted as an object.

A very small number of advanced object-oriented database systems, including the ODMG standard, provide extent, relationship or key definition capabilities as part of a class specification.

In systems that provide for a definition of the extent of the class, the implementation of the component database translation protocol has a choice. The implementor can either use the built-in multidatabase operation that uses query functionality to find the extent, or this default operation can be replaced by component database-specific operations. If key definition information is included in the class specification, then this information is recorded in the common model class specification; otherwise the key property of a class is left empty. Generally key information is used only with component relational database systems. As described earlier in the chapter, it is straightforward to modify the common model to include information about relationships defined on a class. The component object database remains responsible for creating and maintaining the relationships, and the common model need only provide standard operations for accessing this information. Alternatively, a relationship defined on a class can be considered as an additional attribute for the class. For example, the relationship *teaches* in the class teacher, between a teacher object and a set of lesson objects, could alternatively be considered as an attribute with the name *teaches*. The attribute type would be a set of lesson objects. The disadvantage of this implementation is that information about the inverse relationship is unavailable to the user of the common model. However, this is a more acceptable solution than neglecting the relationship entirely.

Some object-oriented database systems allow the user to associate a persistent name with a particular database object. The object can then later be retrieved from the database using this name. The multidatabase provides this functionality only for these component systems and not for all systems. In order to provide this functionality on all component systems, write access to the component system must be guaranteed.

The major difference between object-oriented and object-relational databases is the application programming interface that is made available for the implementation of the common model in the multidatabase system. Object-relational databases, due largely to their evolution from relational systems, tend to provide excellent runtime schema identification and manipulation tools. It is generally possible to extract all details concerning the class hierarchy and to alter this hierarchy at runtime. No knowledge of the database schema is required prior to execution time. Object-oriented databases systems, evolving from persistent programming language implementations, often assume that all schema information is available at program

compile time and have not provided a means to programmatically access schema information at runtime. Since the multidatabase system requires that component database systems can be added or removed from the multidatabase system at runtime, and that no prior knowledge other than the name, location and type of the database is needed, this restriction is very limiting. Object-oriented systems that do not provide some means of performing runtime schema identification are not suitable as component database systems. Fortunately, object-oriented database designers, including the ODMG, have realized this shortcoming and are beginning to provide runtime access to schema definitions. Equally important is the ability to programmatically manipulate data objects in a way that does not require advance knowledge of the type of that object at compile time.

Object-oriented database systems also often distinguish between persistent and temporary objects. Persistent objects are those objects that are stored in the database, whereas temporary objects are used in database programs and class methods and are not stored permanently in the database. The multidatabase system is concerned only with persistent objects and does not provide operations to create or manipulate temporary objects.

3.2.2 Relational Data Model

A relational database schema consists of a set of table definitions. Each table is identified by a table name and contains a set of fields, each with its own field name and associated data type. In the most simplistic scenario, every table in the relational schema can be translated as a class in the common object model. The class name is taken from the table name. Each field in the table is represented as an attribute. The attribute name and data type are taken directly the field it represents. Since relational database systems do not support the definition of complex data types, the data type of a field will always be one of the fundamental data types of the component system. This type is converted to one of the fundamental types of the common model to find the attribute type. There is no inheritance hierarchy explicitly defined among the tables in a relational database, and so the class hierarchy that is created for the common model is completely flat, with no class having any superclasses. Primary key information for the table is also stored in the class definition of the common model.

A data row in a relational table is transformed into a database object instance by assigning the field values of the row to the attribute values for the class that corresponds to that table. One

problem associated with relational systems is that each data row within a table does not automatically have a unique value associated with it. If two rows contain identical data values in each field, then there is no way to distinguish between the rows. These rows would be translated as two objects with identical attribute values, but there is then no way to distinguish between them when it becomes necessary to translate the objects back into the relational system. If a primary key is defined for a table, then the value of this key can be used as an object identifier and all objects can be correctly translated back to the relational system by comparing primary keys. Changes in the value of a primary key are tracked by an object during its lifetime so that the correct correspondence between a table row and the object is maintained. This method of imposing object identifiers on data rows must also consider the possibility of data being altered outside the control of the multidatabase system. This applies only to multi-user database systems. In such systems, the multidatabase object must obtain a lock on the data row that it currently represents. It is the responsibility of the object to release these locks when the object is no longer needed by the system.

One improvement that can be made in the translation process from table definition to class definition is the use of secondary keys defined for a table to deduce a container relationship between two tables [Castellanos et al. 1994]. A field in a table is designated as a secondary key when the field is used to store the primary key data value from another table. For example, Figure 3.1 shows two tables, one representing a person, and one representing a project. The primary key of the person table is the *SSN* (social security number) field, while the primary key of the project table is the *projectID* field. The *projectID* field also appears as a secondary key in the person table to indicate the project to which a person is currently assigned. The corresponding class definitions for the tables show how the relationship is translated in the common object model by making the *projectID* attribute a reference to a project object. In order to implement this functionality, the person object stores the database query that will retrieve the project object together with the other attribute data values.

Other algorithms have been developed to derive superclass relationships between tables by looking for common field definitions in the tables [Papazoglou et al. 1988]. However, this refinement is not necessary for the multidatabase user to be able to use the component database to store and retrieve application data. The algorithms usually necessitate some user interaction

and are not guaranteed to derive accurate relationships, thus possibly leading to user confusion and data misrepresentation. Therefore, it was decided to exclude this functionality from the multidatabase system.

Relational database systems sometimes include additional functionality such as the ability to define data triggers. A data trigger is a way to specify a database action that should be taken when a specified data insert, deletion or update is carried out. These data triggers are usually implemented and maintained by the database administrator. It should be unnecessary for the multidatabase user to need access to information about these triggers. The triggers will operate as programmed during multidatabase use.

Person

SSN	*name*	*salary*	*projectID*
234-23-4324	Smith	10,000	243
254-45-2234	Jones	8,000	243

Project

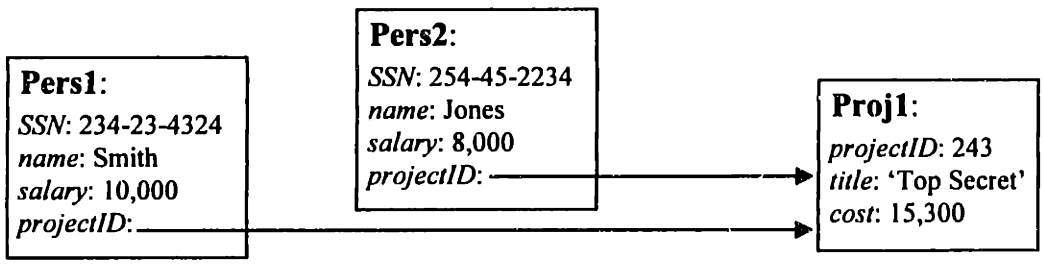
projectID	*title*	*cost*
243	'TopSecret'	15,300

(a) Relational table definitions, together with sample data

```
class Person{
  string SSN;
  string name;
  real salary;
  Project projectID;
}
```

```
class Project{
  integer projectID;
  string title;
  real cost;
}
```

(b) Corresponding class definitions for the common model

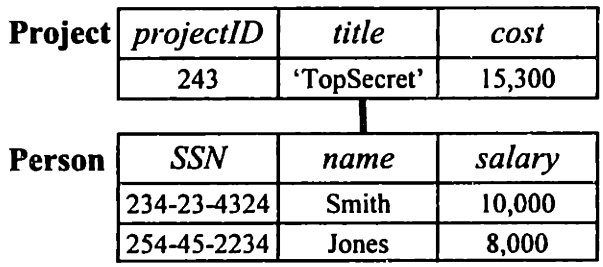


(c) Database objects created in the common model

Figure 3.1 Translation of relational tables to the common object model

3.2.3 Hierarchical Data Model

The hierarchical data model is composed of data elements arranged in a simple tree structure. A set of data element types is defined for the database, with each type consisting of a set of data fields. In the common object model, each element type is considered to be a class, and the data fields are interpreted as attributes. As in the relational model, the type associated with each field will be translated into a fundamental type belonging to the common model. Each data element becomes an object.

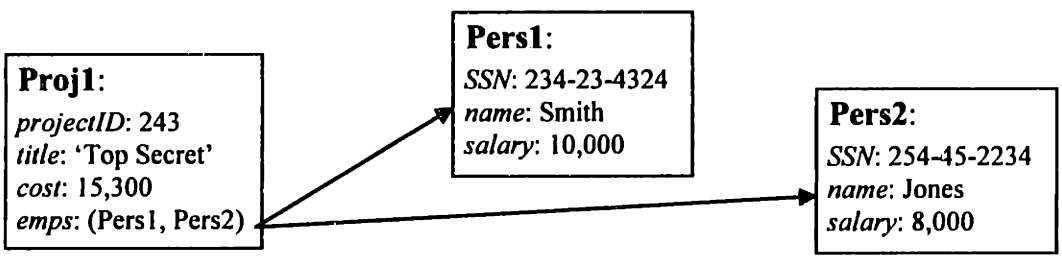


(a) Sample hierarchical data

```
class Project{
  integer projectID;
  string title;
  real cost;
  set(Person) emps;
}
```

```
class Person{
  string SSN;
  string name;
  real salary;
}
```

(b) Corresponding class definitions for the common model



(c) Database objects created in the common model

Figure 3.2 Translation of hierarchical data to the common object model

In order to model the relationship between the structures, each class contains one or more extra attributes of type set. This attribute is used by an object to hold the objects which are rooted

from it. This mechanism is best illustrated by the example in Figure 3.2. Similar to the case of relational systems, there are serious problems with providing each object in the common model with a unique identifier. The same data element may appear several times in a hierarchical database. However, if it is assumed that all data elements that contain the same field values do represent the same physical entity, then the multidatabase model can implement a system of references to objects that prevents multiple copies of the same object from being produced. This can actually remove some of the problems with data update that are traditionally associated with the hierarchical data model.

One of the most common data sources to be modeled as a hierarchical database system is a file system. A file system is made up of two types of data elements, the file and the directory. A file element includes data fields for the file name and whether the user has write permission for the file. A directory element is a specialized type of file element that also contains a relationship to a set of files and sub-directories. A directory element provides a root for the system. Figure 3.3 shows the file and directory classes as they appear in the common model.

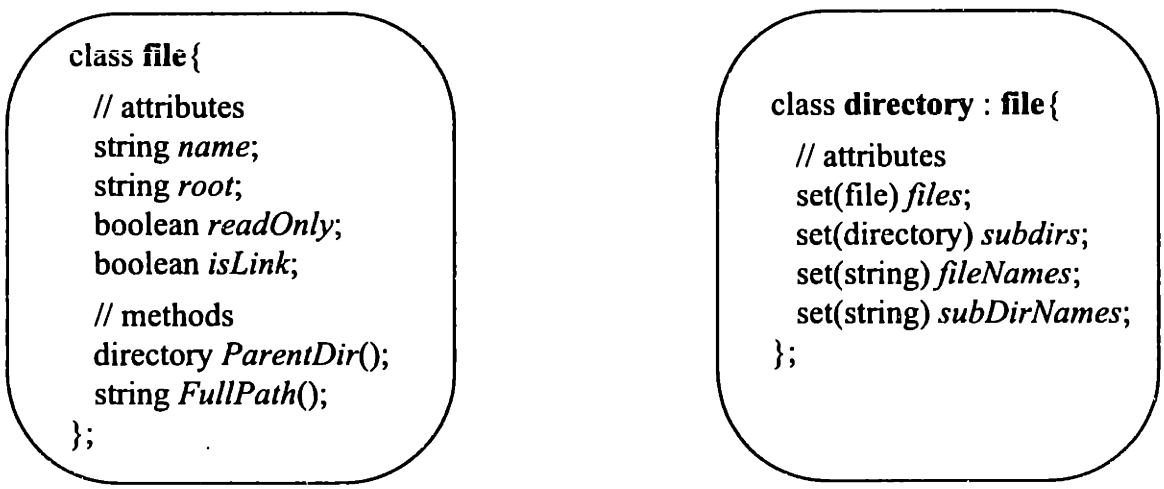


Figure 3.3 Representation of a File System

The implementation of the file and directory class types and objects uses information taken directly from the file system. The multidatabase user is asked to specify the root of the file system that should be used as the data source. The multidatabase system is responsible for creating the object hierarchy as it is needed by the user. Some file systems use the notion of links to represent files or directories. The multidatabase user is informed when a file is a link and is

given a choice as to whether to retrieve the corresponding file or directory object to which the link refers. The multidatabase is also responsible for ensuring that if the object is already in existence, it is not recreated unnecessarily.

The root of the file system, represented as a directory object, is made a persistent root of the system. This persistent root is named 'root'. The user can retrieve the root directory object using this persistent root and use this object as a starting point for exploring the data source. Convenience methods to return the parent directory of a file and the full path specification for a file are added to the file class for user convenience.

3.3 A Common Query Language

In order to retrieve data stored in the component database systems, the common data model must provide an accompanying query language. This query language must reflect the object-oriented nature of the common model by allowing full access to the attributes and methods defined on database objects. The multidatabase system is responsible for translating between the common query language and the language used by the component databases. The system must also present the query results in a standardized manner.

3.3.1 Choice of Extended SQL as the Common Query Language

Both the SQL3 and ODMG standards have chosen to base their query language standards on SQL-92. In fact, it is widely expected that these two proposed standards, SQL3 and OQL, will be merged to produce one standard query language in the future. Both these query languages can be referred to as extended SQL, as they implement a version of SQL that has been extended to take advantage of the object data model. In SQL3, this means that attribute path specifications are allowed in the query where ordinarily only a field name would be valid. OQL allows for the use of both attribute path and method specifications in the query.

In anticipation of the adoption of extended SQL as a database standard, the common model provides an extended query language that is based heavily on the anticipated SQL3 standard while also incorporating some of the more advanced object features of OQL. The common query language for the multidatabase is concerned with data retrieval only. Data and schema manipulation is possible from the multidatabase system, but is done directly through operations

defined on the multidatabase objects, not through the query language. There are four parts to a query as defined in the common query language:

select property (, property2, property3, ...)
from class (, class2, class3, ...)
where condition (**and** | **or** condition2 ....)
order by field (, field2 ....)

where the query variables are defined as follows:

property Any valid attribute path, method call or class name that is taken from one of the classes listed in the **from** clause.

class Any valid class defined for the database

condition Any valid condition involving a combination of properties and constant values

field Any valid field position corresponding to the position of properties in the list given in the select clause

The query language is best illustrated by example, as shown in Figure 3.4.

```
class Hotel{
  City location;
  real roomrate;
  integer rating;
  string name;
  boolean HasVacancy();
}
```

```
class City{
  string name;
  string country;
}
```

(a) Sample class definitions

```
select hotel, name, location.name
from hotel
where HasVacancy() = TRUE and rating > 3
order by 2
```

(b) Sample query in the extended SQL common query language

Figure 3.4 Example of Common Query Language

Assuming the class definitions for hotel and city as given, the query retrieves the hotel object, hotel name and city name for every hotel in the database that currently has a vacancy and is rated above 3. The query results are returned in alphabetical order by hotel name.

The results of the query are returned as a set of result rows. Each row contains a value for each of the fields specified in the select clause of the query. The data values are all instances of the fundamental data types or class types of the common model, allowing standardized access to the results. In order to allow the multidatabase user the most flexibility and functionality possible, the user can choose to execute a query specified in the common query language or in the native query language. In either case, the results are returned in the standardized format.

3.3.2 Object Database Systems

Object-relational database systems generally implement a version of extended SQL that is very similar, except for minor syntactical differences, to the proposed common query language. These syntactical differences are taken into account during the translation process from component system languages to the common language. Many older object-oriented database systems offer only very limited query facilities, since their primary goal was simply to provide object persistence. However, almost all of the newer object databases offer some form of object query language. The major difference between some object query languages and relational query languages is that the object query result is typed. The query result is a single instance of one of the fundamental data types, which includes collection types, or a data object. If necessary, the query specification will be used to define a new, temporary, class type to which the object belongs. In these cases, the multidatabase is responsible for extracting the individual data items from the result and presenting it to the user in the expected standard form.

3.3.3 Relational Database Systems

Relational database systems usually implement SQL as their database query language. This is certainly true of all popular commercial database systems. Normally SQL does not allow attribute paths to be specified in the query, because they are not needed. However, with the inclusion of object references in classes, stemming from the interpretation of secondary keys, this extension must be added. The attribute paths are interpreted by the multidatabase to produce the equivalent SQL query, and the results translated into the expected form. Similarly, a class name can be

specified in the select clause, and the multidatabase system will ensure that an object of that class is returned in the result.

3.3.4 Hierarchical Database Systems

When using a file system as a component data source, no query language of any kind is supplied by the system. Therefore the multidatabase is entirely responsible for implementing the query language. It allows a user to query the attributes of the file and directory objects, so that the queries such as the following become valid.

```
select files, directories
from directory
where name = '/mit/katc/thesis'
```

The query languages for other hierarchical data sources must be customized individually for use in the multidatabase system.

3.4 Multidatabase Functionality

The ability to connect to a variety of component databases and retrieve data from each of them using a standard interface is often all that is required by application users. This functionality alone enables an application designer to use sophisticated data management techniques within the application, in addition to being able to write application code that is independent of any one database system.

However, the formulation of the common data model and associated query language provides an excellent base for the development of some of the more advanced multidatabase features. These features can be used by application designers to coordinate data transfer and data integration among a group of component database systems. However, in order to implement them, the multidatabase must either construct a global schema or provide a multidatabase query language. The creation and maintenance of a global schema is a complex task that requires intelligent deductions from the system as well as user input and intervention. A multidatabase query language is simpler to implement and allows the user to determine the ways in which the individual database schemas are united in the global system. The multidatabase query language is

best suited for application designers, who generally have a more complete understanding of the database systems they are controlling, and can translate this knowledge into an appropriate query.

In order to facilitate data transfer between component database systems, the multidatabase must also provide a way to specify equivalence between class types so that some of the semantic heterogeneities in the system can be resolved.

3.4.1 Multidatabase Query Language

A common query language has already been developed for the multidatabase system, and this serves as an excellent base for the development of a multidatabase query language. The purpose of a multidatabase language is to allow data to be retrieved from a combination of classes spread out among the component database systems. The multidatabase system must break down the query into the appropriate component queries, one for each database. The system is responsible for the execution of each query, and combining the query results for presentation to the user.

As an example, assume that there is a class person in the first component database with attributes name, age and social security number. The second component database has a class employee, which has attributes employee social security number and salary. The multidatabase user might wish to formulate a query to find the name and salary of all the employees under the age of 30. The multidatabase query language is based on SQL and is written as

```
select name, salary
from dbase1.person, dbase2.employee
where dbase1.person.age < 30
    and dbase1.person.SSN = dbase2.employee.employeeSSN
```

The user specifies the equivalence between the SSN attribute of class person and the employeeSSN attribute of the employee class. The multidatabase splits the query into its two constituent components. The first query, executed on the first database, is given by:

```
select name, ssn
from person
where age < 30
```

The names returned by this query form the first half of the multidatabase query result. The second half of the result set is found by executing the following query, on the second database, for each social security number returned in the first query:

```
select salary
from employee
where employeeSSN = ssn
```

There is a wealth of existing research on the decomposition of multidatabase queries and the recombination of the component query results [Kühn 1993]. Any of the existing algorithms for multidatabase query execution can be used in this multidatabase system, provided that the component queries produced are compatible with the common query language syntax proposed in the previous section. Since query execution and result retrieval are carried out using a standard interface, the multidatabase query execution algorithm does not need to consider schema or data translation issues. This flexibility means that the multidatabase query language facilities can be incrementally improved as new techniques are developed.

3.4.2 Type Equivalence

It is important for the multidatabase user to be able to take advantage of class type equivalencies between component database systems, particularly among multimedia database types. For example, a class in one database may have an attribute of type string that represents a reference to an external file that holds image data. Another class in a different database may contain an attribute of the binary data type that also represents an image. It would be advantageous to the user if both these attributes reflected that they were actually of type image. Once the attributes have been identified as the same type, the data values can be compared or transferred from one database to the other.

It is impossible for the multidatabase system to deduce these relationships without assistance from the component database system administrators or multidatabase users. Therefore the multidatabase system provides a means of defining data equivalence within each component system. A special class is created in the component database for this purpose. Each object of the class contains information about the correspondence between an attribute of a certain class and a

multimedia data type. The multidatabase uses the information contained in these objects to correctly represent the class attributes to the user. This extra step in data translation is transparent to the user once the necessary information has been stored in the component database systems.

The ability to specify type equivalence is currently restricted to multimedia data types because this is most important to the target audience of the multidatabase system. However, the same system could be used to specify many other forms of data equivalence. The advantage to storing the equivalence information in the database is that the information is directly under the control of the database administrator who is most familiar with the component system. Also, since all database objects are represented in a standard manner in the multidatabase system, the translation procedures necessary to present the data correctly are common to all component systems.

3.5 Applicability of the Multidatabase System

The multidatabase system described in this chapter does not address all the issues that have been raised in the field of multidatabase systems research. Instead, it concentrates on providing a multidatabase system that is ideal for use in an application authoring environment and provides for the vast majority of needs of the application designer and user. Special emphasis is placed on the ability to handle multimedia data. The system concentrates on providing a solution to the problem of data heterogeneity while allowing each component system to maintain complete autonomy. Less attention is paid to the issues of concurrency control and the observance of global constraints. Individual component systems will of course impose their own concurrency controls over native database operations which is often all that is necessary. The multidatabase system provides enough functionality for the application designer to implement their own constraint mechanism. Often, this is the simplest and best solution, as the designer is most familiar with the semantic content of each database.

The main sacrifice made in order to achieve this wide ranging and flexible functionality is one of processing efficiency, and therefore speed. Routines implemented at the multidatabase system level will not generally be as fast as those routines implemented by the native database. However, much of the processing overhead occurs when the connection to a database is first

made and does not seriously impede database performance throughout the execution of the application.

Chapter 4

Multidatabase System Implementation

The interface to the multidatabase system that is presented to the application designer, and by extension to the application user, is extremely important in determining the contribution that the multidatabase can make to improving the functionality of the application. This chapter addresses the practical implementation of the multidatabase system and highlights the efforts made to provide a powerful, flexible, yet straightforward interface to the system, built on the theoretical foundations discussed in the previous chapter. A number of applications, designed and implemented by staff at the MIT Center for Educational Computing Initiatives, are described which illustrate the benefits that these applications have derived from the addition of multidatabase capabilities. The component data sources currently supported by the multidatabase system are the object-oriented database O2®, the object-relational database UNISQL/X®, the relational databases mSQL® and Postgres95®, any ODBC compliant database and the UNIX® and Windows™ file systems.

4.1 Multidatabase Interface Structure

The multidatabase interface provided to the application designer takes the form of a set of predefined database classes. This fits in appropriately with the object-oriented nature of the AthenaMuse® application authoring environment. The designer can use these classes directly, or can derive more specialized classes for database manipulation through subclassing. The interface can be divided into two main parts. The first part is a set of classes that represent the fundamental data types available in the multidatabase system, while the second part is a set of classes that represent a connection to a component database and provide operations for schema manipulation and query construction and execution.

A connection to a database system is represented by an instance of the DBdatabase class. This class contains references to instances of four other classes; DBstructure, DBconnection,

DBconversion and DBexecution. The DBconnection, DBconversion and DBexecution classes serve as virtual base classes. Subclasses are derived from these classes for each type of component database supported in the multidatabase system. In this way, the system uses polymorphism to hide the implementation details of the classes for individual database systems and common functionality can be provided at the superclass level. When a connection to a database is opened, instances of the appropriate subclasses are automatically created and attached to the instance of the DBdatabase class. The relationship between these classes is shown schematically in Figure 4.1. The shaded area of the figure represents the database dependent portions of the class structure, while the unshaded area represents the database independent portion which is kept as small as possible.

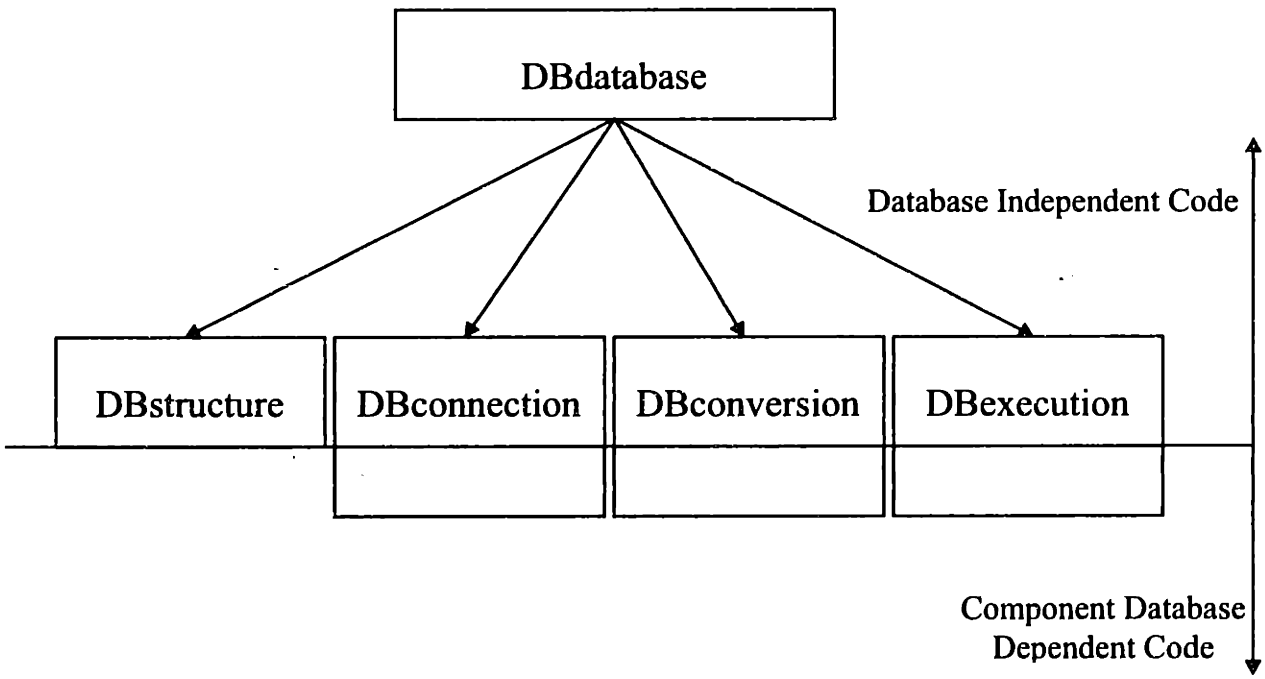


Figure 4.1 Schematic showing implementation of multidatabase system.

The user need learn only the standard interface to the DBdatabase class. Operations on this class are automatically sent to instances of the other classes for execution when necessary, and this complication is kept hidden from the user. The user does also need to interact directly with the DBclass class, which contains schema information, the DBquery class, that represents a query formulated using the common query language, and the DBcursor class, that provides the user

with a way to access the results returned by a query. The remainder of this section discusses the implemcntation of each database class in detail. The complete user interface specification for the database classes, including those classes that represent the common data types, is given in Appendix A.

4.1.1 DBconnection Class

The DBconnection class is responsible for opening and closing a connection to a named database system. In order to open a connection, a minimum of the name and type of the database system that is to be connected must be supplied. Provision is made for the optional specification of the database location, both in terms of a machine location and a location within the file system of a machine. When a connection is opened, the connection class is responsible for carrying out schema identification operations and providing this schema information to the DBstructure class. The class is also responsible for implementing schema manipulation operations, such as creating or deleting a class type in the database, or adding or dropping a class attribute. These operations make use of the DBconversion class to translate between component database data types and the multidatabase common data types. This ensures that all schema manipulation operations are standardized. In addition, if the component database system supports transaction management, the class provides operations to begin and commit transactions.

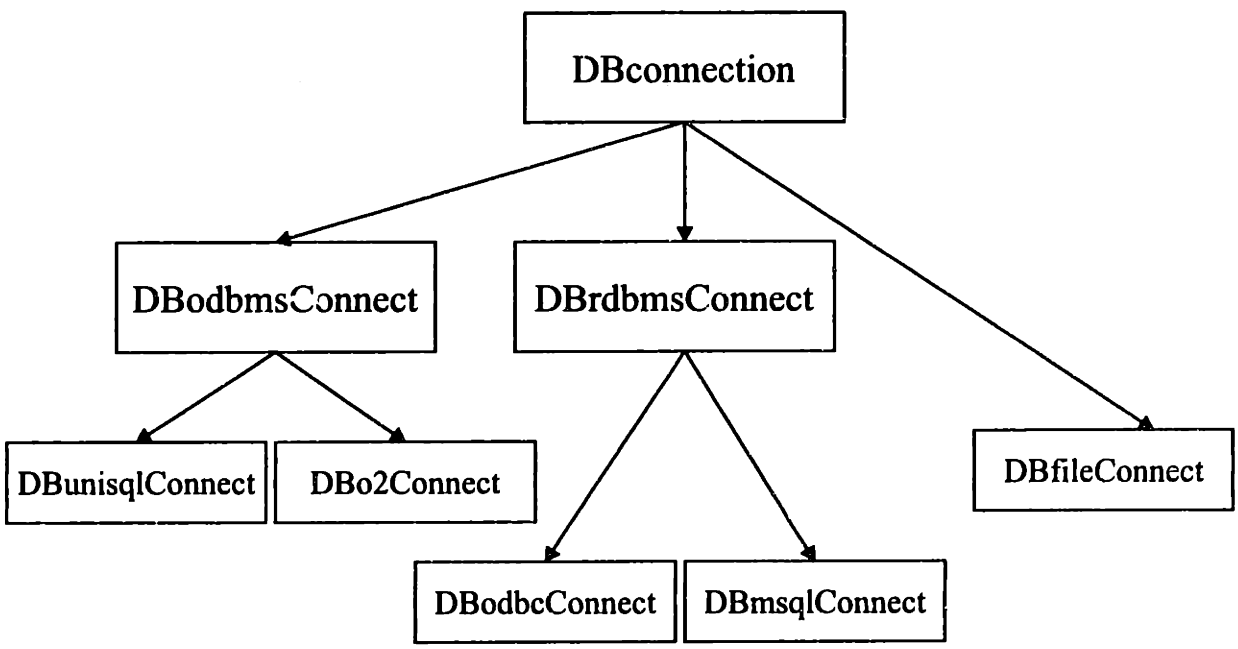


Figure 4.2 DBconnection Class Hierarchy.

Each individual component database system will need the majority of this functionality implemented in a manner that is tailored specifically for that system. However, some operations can be generalized for all database systems or for a certain type of database system. Therefore a hierarchy of classes, shown in Figure 4.2, is constructed, with DBconnection as the virtual base class of the hierarchy. A subclass of DBconnection is created for each of the three main component system types; object, relational and file. Subclasses of these classes are then created for each particular component database system, for example an ODBC subclass from the relational connection subclass. In this way the hierarchy minimizes the amount of work that is required to add a new component database to the multidatabase system and eliminates the need to write nearly identical code for similar component databases. The type of database system that is specified in the database open operation determines which of these classes is instantiated to represent the database connection. Similar class hierarchies exist for the DBconversion and DBexecution classes.

4.1.2 DBstructure Class

The DBstructure class is responsible for maintaining information about the component database schema in a standard manner. The initial information about the schema is obtained from the DBconnection class when the connection to the database is opened. The DBconnection class also passes updated information to the DBstructure class when schema modifications are made. The schema information is kept in a tree structure made up of instances of the DBclass type. A DBclass contains all the information about the attribute, method and key properties of a particular class type. The arrangement of the instances in the tree reflects the sub/super class relationships between the class types.

The schema information can be retrieved by the user through the interfaces to the DBdatabase and DBclass classes. Operations on the DBdatabase class provide information about the names of classes in the schema and a means of retrieving the DBclass instance corresponding to a named class. Operations on DBclass provide information about the properties of that class. However, the information is also used internally by the multidatabase system to perform error checking and to aid in data type and query translation. Information concerning data type equivalence is also stored internally in the DBclass instance for use by the multidatabase system. The ability to look up schema information locally, without needing to query the database directly, can provide

significant time savings, particularly when the database is located on a network and not on the local machine where the application is running. However, the time taken to open a database connection may be longer to allow the schema information to be retrieved.

4.1.3 DBconversion Class

The DBconversion class is responsible for carrying out the mechanics of data translation between the data types of the component database systems and the common data types defined in the multidatabase system. Similar to the DBconnection class, the DBconversion class serves as a virtual base class for a hierarchy of classes that implement the class interface for the various component database systems. The DBconversion class is used by almost all the other classes in the system to provide data in a standard format to the user.

The class works in conjunction with the DBnativeType class hierarchy which provides a mechanism for holding a data value from a component database system. There are two principle operations defined on the DBconversion class; one to convert from an instance of DBnativeType to an instance of one of the common data types, and the second to carry out the reverse procedure.

4.1.4 DBexecution Class

The DBexecution class hierarchy is responsible for submitting queries to the database system for execution and for providing access to the query results in a standard format. The query can be specified in either of two ways, as a character string or as an instance of the DBquery class. The results of the query execution are presented to the user as a reference to an instance of the DBcursor class.

If the query is submitted in the form of a string, the query string is passed directly to the component database system for execution. No attempt is made to translate or modify the query in any way, which may be advantageous to the user in certain circumstances. For example, it may allow the user to take advantage of some very specific component database functionality. Alternatively the user may choose to build a query specification using the DBquery class. This class allows the user to build up the query from its component parts and to perform late binding of query variables. This is advantageous to any user who wishes to execute a series of related queries, where only minor modifications are made to the query specification each time. The

DBquery class interface provides significant improvements over manipulating the query string directly in such cases.

The other important feature of queries submitted using the DBquery class is that they are assumed to be specified according to the common query language defined for the multidatabase. The query is translated, as appropriate, for the target component database system. This allow the extended query language to be used with those component databases, such as relational systems, that would not normally support this functionality. Additionally, it allows data type equivalence information to be used in the translation process.

The DBcursor class provides information about the type of data values in the result set, and the number of result rows returned in the set. It also provides a mechanism to move forward and backwards through the result set and retrieve individual result rows. All data values returned in a result row are instances of one of the common data value types.

4.1.5 Data Type Classes

The DBobject class represents the object common data type, the most important of the common types. An instance of the DBobject class represents an object from a component database system. The object may correspond to a row in a relational table, a database object, or a file specification, depending on the type of the component system. The DBobject class provides a standard interface to the object in all cases. The class provides operations for retrieving and modifying the value of an object attribute and executing any methods that may be defined on the object class. The class also provides operations to delete the object from the database and to retrieve a reference to the DBclass object, that will provide the class type information for the object.

There is a class to represent each of the complex common data types, e.g. DBdate, DBimage. Each of the classes provides a constructor so that a data value of that type can be created, and operations so that the data value can be manipulated. For example, the DBdate class provides a constructor that takes the values for month, day and year as arguments. The class provides operations to retrieve the date information as a string or a list and to perform simple arithmetic involving dates.

4.1.6 DBmultidatabase Class

The DBmultidatabase class provides a means of executing multidatabase queries on a set of component database systems. The class provides operations to add and remove systems from this set, and to associate a unique name with each system. Each component database is represented as a reference to a DBdatabase object. Multidatabase queries are specified as character strings and the results returned in the normal manner as a reference to a DBcursor object.

4.2 Network Accessible Database Systems

Often an application user will want to open a connection with a database system that is not located on the local machine, but can be reached across the network. There are two ways in which this functionality can be achieved in the multidatabase system. The first method takes advantage of the network capabilities of individual database systems and the second takes advantage of the network functionality built into the application development environment.

Many database systems, e.g. Oracle®, UNISQL/X® and SQL Server®, are designed to take advantage of client server technology. The main database system is installed on a machine known as the database server and a database client component is installed on any machine that wishes to connect the system. Database users on client machines can connect to the database by specifying the name of the database server in the connection specification. The multidatabase can take direct advantage of such systems when an application is run on one of these client machines. Provision is made for the server name to be specified when a connection to a database is opened.

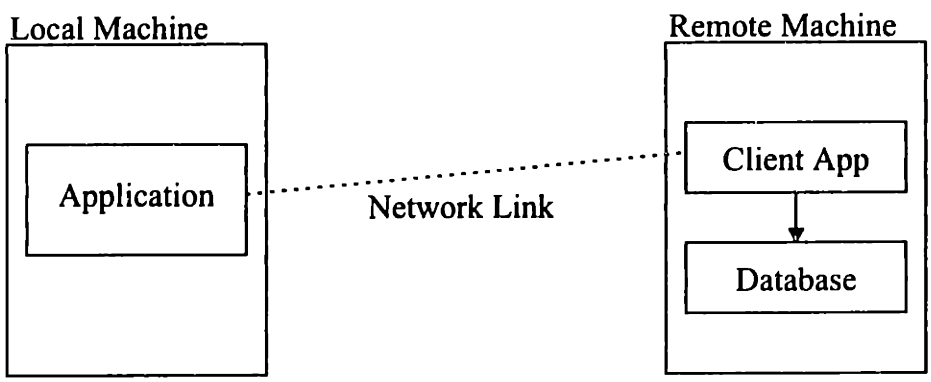


Figure 4.3 Network Access to Component Database Systems using AthenaMuse®

If the database system does not supply any networking capability, then the network functionality of AthenaMuse® can be used to imitate a client server database system. An AthenaMuse client application is built to run on the machine where the database system resides and communicates with the main application via one of the network protocols available within AthenaMuse®, as shown in Figure 4.3.

This client application will accept requests from the main application to execute database operations and return the results across the network. The application user need not be aware of the additional steps necessary to connect to the database. Due to the multi-platform support available in AthenaMuse®, this technique for connecting to database systems across the network is also applicable when the machine types on which the application is needed to run and the database system is located are different.

4.3 Sample Applications

There is an endless variety of AthenaMuse® applications that could be designed to take advantage of the multidatabase capabilities. A sample of the applications developed to date are described here in order to provide an indication of the diverse database systems which are of use to application designers. The purpose, complexity and target audience of the applications all vary widely.

4.3.1 The MIT Museum Edgerton Collection

The MIT Museum recently completed a project to provide electronic access to the collected works of Harold Edgerton, famous as a pioneer of strobe photography. The collection was cataloged according to the US MARC standard using the object-relational database UNISQL/X®. The standard is remarkably complex, allowing for the possible specification of hundreds of data fields. The museum hopes to use the system to catalog other portions of its collections, and therefore the standard was implemented in full in the database. The ability of UNISQL/X® to model complex data structures was of immense importance to the success of the project, which would have been very difficult, if not impossible, to complete using traditional relational database systems. The object-relational database also provides better facilities for storing image data, either in the database for small thumbnail images, or as references to external files for larger images.

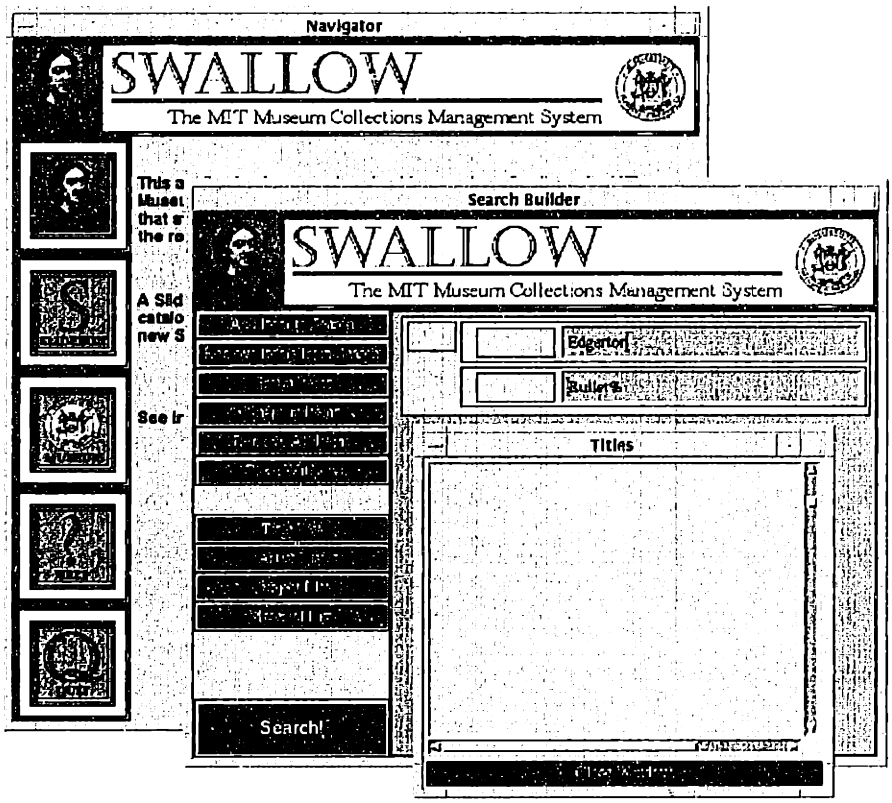


Figure 4.4 Constructing a Query on the Edgerton Database

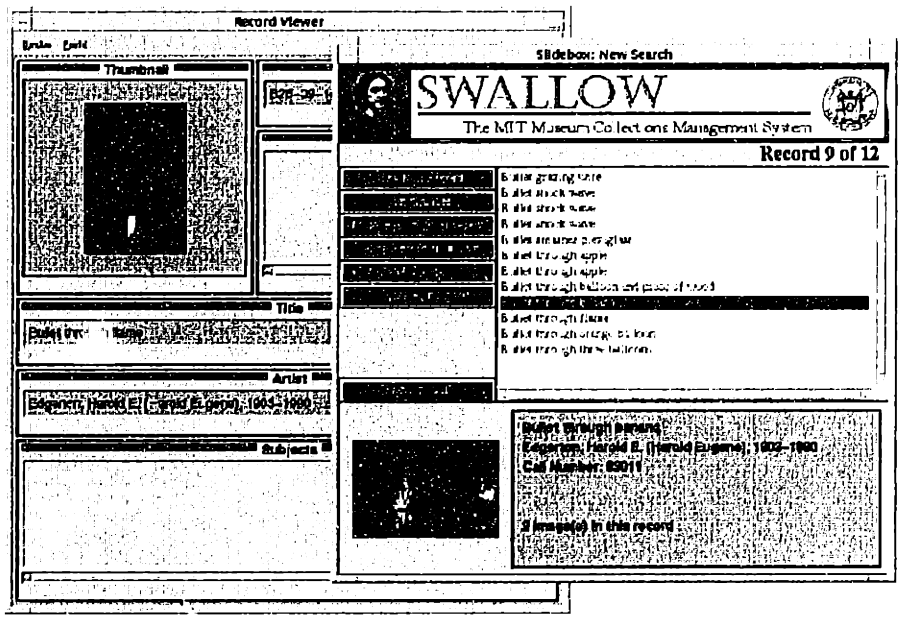


Figure 4.5 Search Results from the Edgerton database

In addition to using the database catalog for internal bookkeeping and reference purposes, the museum also wished to provide an interface to the database so that a museum patron could browse Edgerton's photographs. There are over 7,000 photographs in the collection and it would be physically impossible to display all of the photographs together. By providing a patron with database access to the photographs, the patron can explore the entire collection more easily, searching for photographs of a particular subject that interests him or her, or just browsing the catalog at random.

UNISQL/X® does not provide an application generation tool with their database system, so an external authoring tool needed to be found for constructing the application. No commercial application builder is available that can be linked to UNISQL/X® and writing the application directly on top of a windowing system would require significant effort. A much better solution was to use the AthenaMuse® authoring environment with its multidatabase module to link to the UNISQL/X® database. Kara Schneiderman and Edgar Ngwenya were responsible for the development of the application which can be divided into two main parts. Figure 4.4 shows how the user enters a number of search parameters that will determine the query performed on the database. Figure 4.5 shows how the matching data records are returned to the patron, who can then choose to view any particular record in more detail.

4.3.2 New England Aquarium Exhibit

A design team, led by Lestra Litchfield with Justin Lapierre and Philip Bailey, has used AthenaMuse® to author a kiosk exhibit featuring information about ponds in the New England area for the New England Aquarium. Facts concerning the fish and algae levels in each pond, and the size and health of the pond are stored in a very simple relational database. When the application user selects a particular pond, the information relevant to that pond is retrieved from the database and displayed as shown in Figure 4.6. The database commands used by the application designer are identical to those used in the previous application, the benefit of the standard multidatabase interface.

There are many advantages to using the database to store pond information. The database was already in existence at the beginning of the project, so no extra work was needed to modify the data for use in the application. Updates made to the information in the database are immediately

and automatically reflected in the application. In order to add a pond to the application, a large percentage of the work is done by simply adding the pond to the database.

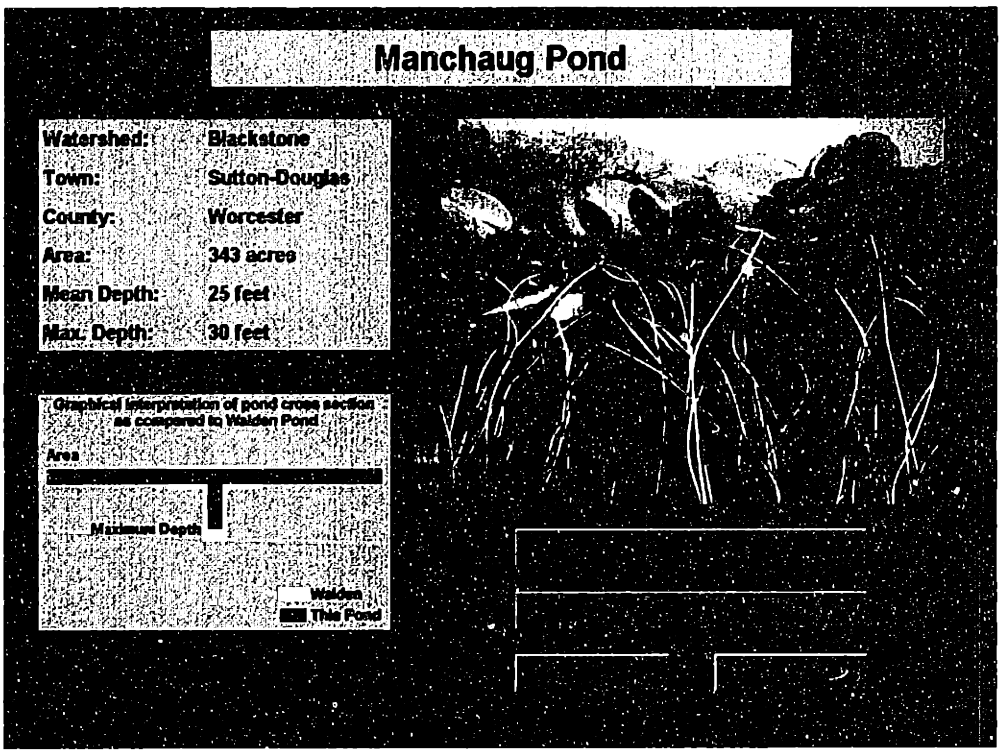


Figure 4.6 The Ponds Exhibit of the New England Aquarium

4.3.3 Operación Futuro

Operación Futuro is an application designed by Ana Beatriz Chiquito to teach intermediate Spanish to college level students. It uses images, audio and video clips, and text to take the student on a two day visit to the Colombian city of Medellin. The application uses a relational database to store many diverse pieces of information that are needed to ensure its smooth functioning.

The subtitles for all the video and audio clips are stored in a database table. This enables the subtitles to be centrally maintained and updated. It also means that a simple alteration to one database query can change the language in which the subtitles appear. Another database table implements a simple dictionary for the words that occur in these subtitles. In the future it is planned to store references to supplemental historical, geographical and social information in the

database. As the functionality of the application is expanded, it is envisioned that the database system will be used to customize the learning experience for each user individually. The teacher will be able to determine which sections of the city tour are made available to the student or whether subtitles appear with the media clips by modifying student profiles stored in the database. The database will also be used to store information about each student's journey so that his or her progress can be evaluated.

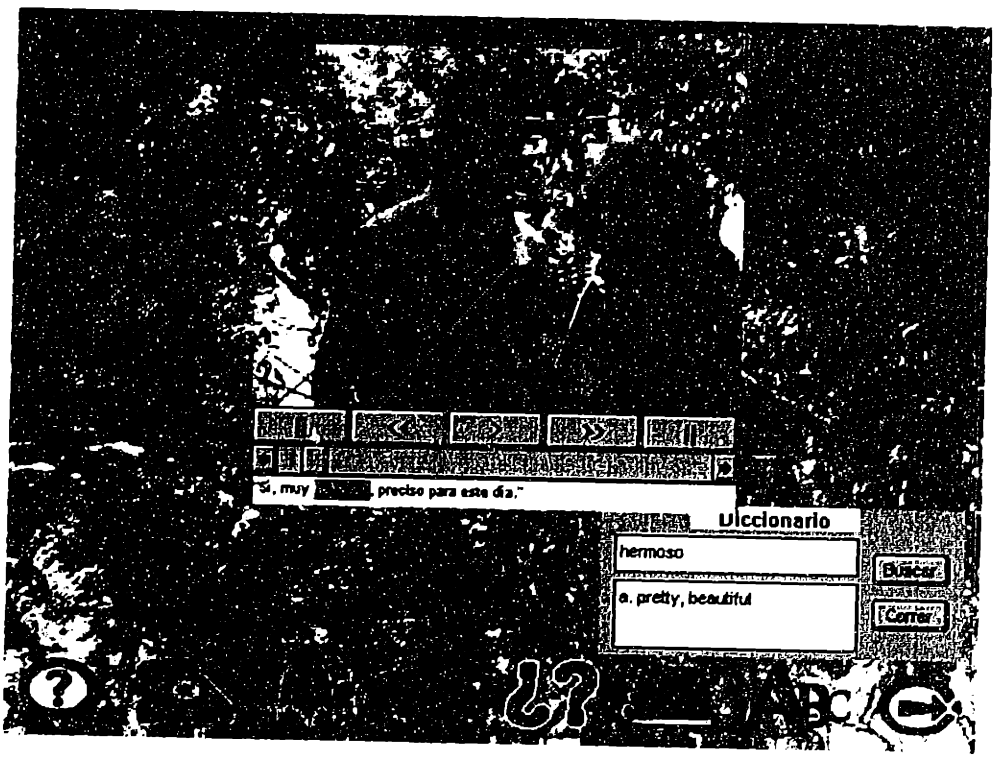


Figure 4.7 Subtitle and Dictionary Information for Operación Futuro

4.3.4 Additional Applications

The response from application designers and users to the addition of a multidatabase component to the AthenaMuse authoring environment was extremely positive. Designers typically take one of two approaches to application authoring using the multidatabase component. In one approach, characterized by the Edgerton application, the database forms the core of the application, and the primary goal of the application is to implement a browser type interface to the database. The second approach, characterized by Operación Futuro, makes use of a database as an organizational tool to allow greater innovation and flexibility within the application.

Several other major projects are currently under construction that also rely on the multidatabase technology. Electricité de France is planning to use the authoring environment to create an interface to a multimedia documentation archive. The archive is part of a digital library project that aims to improve information flow within the company and employees will be able to use the interface to store and retrieve reference material. The documents will be stored in an O2 object-oriented database. The 'China Project' is an exciting joint venture between The Long Bow Group, Inc., Prof. Peter Perdue of the MIT History Faculty and the MIT Center for Educational Computing Initiatives to develop a multimedia interface to the huge media archive on China collected by Long Bow. A database accompanies the archive and records particulars about each media item. The prototype interface is divided into two modules; the first offers direct access to the database to search for media and the second offers a series of guided tours through the media. Tremendous effort has already been spent in creating the database and indexing the media by date, title, subject keywords, etc. so an authoring system that provides convenient database access is essential for the creation of the interface and to allow the construction of complex queries to locate media.

Several smaller scale applications have been implemented to highlight particular features of the multidatabase system. A generic database browser has been constructed that can connect to any component database supported by the system. The browser displays database schema information graphically and allows the user to execute queries and retrieve data using a menu driven interface. The graphical presentation is especially useful to users who are unfamiliar with database systems. A simple media file viewer was built to highlight the use of a file system as a component database. The viewer was then extended to include the ability to search for media in a selection of databases. The user is then given an option of storing a copy or reference to a chosen media item in a local database. This viewer is intended to model the needs of a student carrying out research for a project. The student will need to be able to connect to and search a variety of sources to retrieve relevant information which can then be stored locally for use in the final project presentation. The local database can be a simple relational database which fits within a student's limited resources. It is important to remember that the multidatabase interface allows the application designer to provide access to multiple databases without expending significant effort.

Several of the applications described in this section would have been significantly harder to implement, if not impossible, without the existence of a multidatabase component in the authoring system. The popularity of the component is a good indication that it would be a significant improving addition to any authoring environment.

Chapter 5

Storage of Application Objects

The multidatabase system developed in this thesis so far has provided excellent data management facilities to the application designer. However, the application data represents only half of the application. The behavior of the application comprises the other, equally important half. In object-oriented authoring tools, this behavior is captured in the object classes and instances that together make up the application. The ability to store these application objects in a database would encourage new and innovative ideas for application construction, modification and reuse, as well as providing support for the implementation of collaborative work and learning environments. The user should be able to choose the database in which the objects will be stored from as wide a range of database types as possible. The multidatabase system therefore provides an excellent foundation for the construction of application object storage and retrieval capabilities.

It is envisioned that the ability to store and retrieve application objects will be advantageous in all object-oriented application authoring system environments. Throughout the development of the storage and retrieval methodologies, care is taken to recognize and consider the possible requirements of individual authoring systems, their idiosyncrasies and potential incompatibilities. The AthenaMuse® authoring environment is again taken as a representative of these systems for the purposes of implementing the methodology, so that it can be used and evaluated. In order to implement object storage, the AthenaMuse® authoring tool must supply a base set of functionality that allows runtime access to object information. By examining this set, a list of functional requirements for other authoring environments can be drawn up.

The first step in creating a procedure for the storage of application objects is the determination of the properties of a typical object. Armed with this knowledge, multidatabase system operations can be utilized to store objects in a database. A strategy that enables intelligent retrieval of these objects from the database and their incorporation into a new application

environment is then developed. Finally, in order to demonstrate the benefits that can be gained from including these capabilities in an authoring environment, a prototype application editor is modified to make use of the object storage facilities.

5.1 An Application Object

The AthenaMuse® authoring environment gives the application designer access to a set of predefined object classes, as well as the ability to define new classes using the application description language (ADL). An application is composed of instances of these classes, and is itself an instance of the application class. A user may wish to store the entire application object in the database, or only a part of the application represented by a constituent object. In some circumstances the user may wish to store only a class specification and is unconcerned with preserving any instance information. The terms instance and object are used here interchangeably to refer to the same concept. The term application object is used to refer to any object component of the application, including the application itself.

5.1.1 Basic Object Class Properties

Within the AthenaMuse® environment, an object class is composed of the three features that are common across almost all object-oriented systems, including the common object model that is used in the multidatabase system.

- A set of base classes from which the class inherits behavior
- A set of class data members, with each member having an associated member name and member type. The values of these members determine the state of the object.
- A set of class methods. These methods determine the behavior of an object of this class type.

Every object is an instance of one particular class and is uniquely identifiable. The most important difference between an application class definition and a class definition from the multidatabase system is found by examining the set of valid member types. (A member of an application class is equivalent to an attribute of a database class.) Each authoring environment is responsible for defining its own set of base types. A valid member type is one of these base

types, or a previously defined class. For example, AthenaMuse® defines ten base types, six simple and four compound types.

The simple types are:

integer

real

string An ASCII string.

boolean TRUE or FALSE.

handle Reference to a base type, compound type or object.

any Can be any simple or compound type.

The compound types are:

list Sequence of data values. Each value can be of simple or compound type.

time Ordered tuple of four integers representing a period of time.

interval Ordered pair of integers or real numbers with an associate open or closed condition for each half of the pair.

array All elements of an array must be of the same simple or compound type, excluding the array type itself. Each element is referenced by a set of keys. The number and type of the keys must be defined when an array variable is declared.

It is immediately obvious that these types do not match one-to-one with the base type set developed for the multidatabase system. The choice of base types for the multidatabase was heavily influenced by the needs of traditional database users. The base types used in application authoring environments, on the other hand, have been influenced by the data types found in many programming languages and known to be useful to programmers. It is important for the differences between the sets to be resolved so that an application object can be stored using the multidatabase system.

The application class specification includes not only the set of method interface definitions, but also the application code that implements each interface. As is typical in object-oriented systems, the method set can include special methods known as constructors. Constructor methods are used to create an instance of the class and are distinguished from each other only by the number and type of method arguments each takes. The default constructor is a constructor

method that takes no arguments and is used when no other constructor is specified. The object model used for the multidatabase system includes a set of method interface definitions but is unconcerned with the implementation of the method which is taken care of by the component database system to which the object class belongs.

5.1.2 Extended Class Properties

The syntax for specifying a class definition will obviously vary among application environments. Each environment frequently provides extensions to the three basic class properties described in the previous section. The aim of these extensions is to make the programming environment more convenient for the application designer. In order for the application class to be recreated exactly as it was originally specified, the complete class definition, including these extensions, must be stored in the database. The extensions might include the ability to include default values for class members or to declare immediate object members following a class definition. The AthenaMuse® environment introduces the concept of the class 'anonymous' for use in situations where all instances of a class are declared as immediate members. A user does not need to specify a name for an anonymous class, creating a potential problem for the storage of the class.

Within a typical database system, all classes are defined at the same level, meaning that all classes are visible from all other classes. However, within an application environment it may be possible to define a class within the scope of another class. A class defined in this way is known as a nested class. A nested class is available for use, perhaps as a member type, within its enclosing class only. Other classes within the application will not have access to its definition. The application environment may also provide a way of defining variables with a global scope which are accessible from any object in the application. Such scope rules must be included in the class specification that is stored in the database.

5.1.3 Predefined Object Classes

An object-oriented authoring environment will usually provide a set of predefined classes for the convenience of the application designer. These classes provide functionality that is expected to be useful to a significant number of designers. For example, AthenaMuse® provides classes that represent items such as images, window system components and networking protocols, so that

the designer can quickly build a complex application with minimal effort. The designer can take advantage of the inheritance mechanism to customize the behavior of any of these classes.

From the point of view of an application designer, it is possible to treat predefined and user defined classes identically. The origin of the class is insignificant. However, it may be important to consider the classes separately for the purpose of storing the class definitions in a database. A predefined class can be assumed to be permanently available in the authoring environment. There is no need to store the class details in the database and waste storage space with information that can always be retrieved directly from the application. Often predefined classes embody complex functionality and would require significant storage space. A mechanism for storing objects of these predefined class types must still exist.

There are several other reasons why it may be advantageous to rely on the continual availability of the predefined classes within the authoring environment. The authoring environment may be revised, and the efficiency and functionality of the predefined classes improved. Applications can take automatic advantage of these improvements if the version current in the application environment is always used. However, this may produce inconsistencies between stored objects of predefined classes and the current class specification if significant alterations are made to the class interface, especially object member definitions. While the two class types, predefined and user defined, may be designed to appear identical to the user, they may be implemented differently and require separate treatment for preparation for storage. The AthenaMuse® environment provides excellent facilities for determining the properties of user defined classes, but does not provide the same level of access to predefined classes.

5.1.4 Widgets

A central part of any application development environment is the ability it provides to construct and customize graphical layouts. As part of its collection of predefined classes, the AthenaMuse® environment provides classes which represent elements of a windowing system. The user then creates objects of these classes and combines them to create the application interface. Traditionally, these window elements have been known as widgets. A widget might be a window, a menu bar, a button or a text field. A class that represents a widget has several unique features which will affect the storage and retrieval of such objects in a database.

All widget objects are assigned a place in the widget parent/child hierarchy. A widget representing a base window forms the root of the hierarchy, and is the only widget which does not require a parent widget. All other widgets must be created as a child of a parent widget. For example, if a user wishes to place a button on a window, the button object must be created with the specification that the window object is the parent of the button. This ensures that the button is attached to the window. The parent is specified in a special member of the child widget object and a valid parent widget must be provided at the creation time of the child object. Widget object classes also possess an additional set of properties called attributes, each attribute having a name and an associated type. Attributes differ from members in that a change to an attribute indicates the need to redraw the widget; for example, a change in the background color. Each widget class has a different set of possible attributes representing characteristics such as width, height or color. Each object can provide information about the attributes which have been set for that object, but is unaware of the existence of attributes which have not been specifically set for the object.

Widgets are an example of a set of classes and objects that require special handling to ensure their correct creation after storage in a database. Any given application environment may have other sets of classes that also require individual attention

5.1.5 Sample AthenaMuse® Application

A partial implementation, written in ADL, of a simple rolodex application is shown in Figure 5.1. The code is meant to illustrate the use of both user defined and predefined classes and is not intended to address all the design issues of such an application. The purpose of the application is to allow the storage and retrieval of basic contact information for a group of students. The advantages of adding storage capabilities to the rolodex will be considered later in this chapter.

ADL syntax is very similar to C++ syntax. The major difference is the way in which methods are invoked on objects. In ADL, a method is invoked on an object using the following syntax:

returnValue = {`MethodName, argument1, argument2, ... } => object;

Method declarations are prefaced by the keyword **on**. Constructor methods are prefaced by the keyword **upon**. In the code samples that follow in this chapter, all keywords and predefined class

names appear in boldface and all comments in italic. The keyword **class** indicates the beginning of a class definition which must be enclosed in brackets.

```
class student{
    class address{
        string dormName;
        integer roomNumber;
    }
    string lastName;
    string firstName;
    integer dormPhone;
    handle moreInfo              // handle to another object
    address dormAddress;
    upon Create : string lname, string fname, integer phone    // non-default constructor
    {
        lastName = lname;  firstName = fname; dormPhone = phone;
    }
};

class rolodex : XFtop{          // XFtop is predefined and represents a main window
    XFlabel name, phone, address;      // XFlabel, XFtextField and XFbutton are
    XFtextField searchOnName;          // all predefined widget types
    XFbutton search, exit;
    upon Construct{                                  // default constructor
        search.Pressed = {'FindStudent, self};       // determines the behavior of the
        exit.Pressed = {'Exit, 'TheApp};             // application when button pressed
    }
    on FindStudent{                                  // locate a particular student instance
        // insert code here to retrieve an instance of student from a list, file
        // or database using criterion in searchOnName.text
        {'Refresh, hFoundStudent} => self;
    }
    on Refresh : handle theStudent{                  // update the rolodex information
        name.label = theStudent->firstName + " " + theStudent->lastName;
        phone.label = toString(theStudent->dormPhone);
    }
}
```

Figure 5.1 Sample ADL Application to Implement a Primitive Rolodex

5.2 Object Storage

The primary goal of object storage is to store sufficient information about an object in a database so that it can later be recreated as part of an application using only information retrieved from that database. It is advantageous if the object can be stored in such a way that the object data is meaningful to a user who has access to the database but who does not wish to recreate the object within the application environment. The validity of object data outside the application environment will of course depend on the nature of the object itself. For example, if an object is designed to represent a person, then information such as the name and age of the person could usefully be accessed and modified outside the application environment. However, an object representing a user interface component may be meaningless when viewed from outside this environment. There are two stages involved in storing an object in a database; first the class specification must be stored in the database, and second the state of the individual object must be recorded in the database.

5.2.1 Storing the Class Definition

It is impossible to recreate an object in an application environment without first recreating the class to which the object belongs. The class information need only be stored once in each database in which objects of that class type are to be stored. When the multidatabase system receives a request to store a particular object in a database, the first step taken is to determine whether the class information for the object has previously been stored. If the class information does not already exist in the database then the class specification will be stored.

During the storage procedure for a particular class definition, other class definitions may be encountered in the form of base classes, nested classes, member types or method arguments. Each new class definition is checked to see if it has already been entered in the database and is added when necessary. In this way, all the class definitions required to completely specify a class are recursively stored in the database. If a class is detected to be one of those classes provided by the application, then there is no need to store the class specification, since there will be no need to recreate the class. It is unnecessary to recursively gather information about base classes or nested classes of predefined classes, as these classes must also be predefined.

The system can provide an option to store only the class definition. This option is convenient for application designers who wish to store object behavior for reuse at a later date, but will create only new instances of the class at that time. In this case, the user is uninterested in preserving the state of individual objects. An example might be an object class representing an image viewer, where the application designer wishes to reuse the behavior of this interface component. However, if the application designer creates a class to represent the characteristics of a customer, then the designer is interested in storing each customer object individually so that the objects can be reused over time in various applications to build up important information about the customer base. This functionality is also required by applications which wish to store the class definitions and the object instances in separate databases. The ability to store class definitions in one central database allows the application to better control access to the definitions and prevent unauthorized modification of the classes.

The significant differences between the common object model used in the multidatabase system and the application object model have already been described. The application model is more complicated, encompassing the need to distinguish between certain types of classes, i.e. widgets and the need to remember member and attribute default value information and method implementation code. The complexity of the application class means that it is impossible to map directly from an application class to the database common object model without the loss of information.

In order to overcome this problem, a special class is created in each database for the storage of application class specifications. Each instance of this class will represent an application object class. The instance will hold information about class bases, methods, members and attributes, the default values for class members and any other application specific information that is needed to recreate the class. In addition, for each application class, a database class is created that is capable of storing the state of an application object of that class. These database classes are created by the multidatabase system using information about the inheritance structure and member properties for the application class.

Usually each database class is given the same name as the application class it represents, and the database class attribute names are copied from the object member names. The attribute type is similarly determined from the member type. The integer, real, string and boolean application

base types translate directly into database base types, as do class types. For each of the compound application base types, a database class is developed to represent the compound data structure. These special database classes are created the first time an object storage request is received. A member of compound type generates an attribute in the database class with the corresponding database class type. The database class is designed to mirror the application class as closely as possible so that the information may be usefully interpreted outside of the application environment.

The major source of class storage errors is expected to come from namespace clashes within the database system. For example, two separate applications may each have a unique implementation of a class named person, and may both wish to store the class in the same database. Clearly, the first application will be able to store the class person, but an attempt to store the class by the second application will result in an error because the class person already exists in the database. This naming problem is often exacerbated by database systems which use case insensitive naming techniques and are unable to distinguish between certain application class names, i.e. person and Person. One way in which to avoid the majority of clashes is to store the class name as used in the application as part of the class specification and, when necessary, to assign a different class name to represent the class in the database. These class names might be derived by appending a version number to the name of the class. A similar problem may arise with the storage of nested classes. Two application classes may both define nested classes with the same name which are distinguished by their scope within the application but must be renamed to distinguish them in the database.

Due to the necessity of recording class specification information, the first object instance belonging to a particular class may incur significant processing overhead. The application can avoid this overhead only by storing the class definitions in the database in advance. Obviously, this is only possible where it is known beforehand which application objects will be stored in the database.

5.2.2 Storing the Object

In order to store an application object in a database, the state of the object must be captured and recorded. This state is given by the current values for each class member of the object. Therefore, storing an application object is equivalent to storing its member values. In the case of widget

objects, the attribute values of each object must also be recorded. The database class, created to represent the application class during the storage of the class specification, is used to store the member value information. When a request is made for an application object to be stored, a new instance of the database class is created. The system iterates over the list of object members and sets each of the database object attributes accordingly. The member types can be divided into four categories for the purposes of describing how values of these types are stored in the database; simple types, complex types, class types and handle types.

The simple application types are represented by the equivalent database base types. The value of the member is simply assigned to the value of the attribute. The multidatabase system takes care of any conversion issues that may arise. The complex application types; list, array and any, are represented by specially created database classes. When a member of one of these types is encountered, a database object of the appropriate class type is created. The attribute values of this object are assigned according to the object member values. The database object representing the complex value is then assigned to the appropriate attribute of the database object.

Similarly, for members whose type is another object class, an object of the appropriate database class that represents this object class is created and assigned to the appropriate attribute of the initial database object. If necessary, this process is carried out recursively so that all object members are stored in the database. A list of objects that have been stored in the database in response to a request to store one particular object is maintained, so that multiple references to a single object can be easily resolved. The list ensures that only one database object is created for each application object and the storage procedure cannot become entrapped in an infinite storage loop. However, if the same application object is encountered during two different storage requests from the user, then that object will be stored twice in the database and will be represented by two different database objects. This happens regardless of whether or not the members or attributes of the object have been modified.

The handle type represents a reference to another piece of data of any valid application type. It is the equivalent to the pointer type used in many programming languages. The handle type is represented by a special handle class created in the database in a manner similar to that for complex types. The importance of the handle type is that the system can choose whether or not to store the piece of data that is referenced by the handle. If the system chooses not to store the

referenced data then the value of the handle is set to null. If the handle references a simple data type, for example a string, then little extra work must be done by the system to store the string. However, the handle could reference another object, which might require significant effort for it to be stored in the database. The suggested system default behavior is to expand all handle references in application objects and store all data referenced by a handle. The storage procedure is again recursive if necessary so that a deep copy of the object is made. Ideally the system would allow the application user to specify whether an object is stored using deep copy semantics. The user could significantly improve performance by choosing not to store handle references and is in the best position to judge which storage method is most appropriate.

In order to aid object retrieval, the system provides the user with the option of storing an identifying name with each object. This name, together with the object class name, can be used to uniquely identify the object in the database.

5.3 Object Retrieval

After an application object has been successfully stored in a database, the system must provide a procedure for recreating the object in the application environment. In order to recreate the object, it may first be necessary to reload the class specification for the object into the environment. Alternatively, the user may wish to only load a class specification from the database so that he or she can create new objects of this class type in the application. The multidatabase system allows objects and classes to be simultaneously retrieved by the same application from multiple database systems.

5.3.1 Class Definition Retrieval

The retrieval of a class definition from a database is triggered when an object of a class type not currently defined in the application is requested from the database or when a direct request for the class definition is made to the database. Once the retrieval request has been made, the system uses the information stored in the special application class object in the database to recreate the class. During the class creation process, all nested classes and base classes will also be created if they are not already present in the application. Similarly, classes that are necessary for the definition of members and methods will also be automatically retrieved from the database. If any of these required classes cannot be found within either the application or the database, the system

returns an error to the user and no modifications are made to the application class structure. Such errors can be caused by database modifications that occur outside the control of the application.

The most critical problem faced by the system is the resolution of name space clashes between existing application classes and those classes retrieved from the database. This problem is compounded by the recursive nature of class creation which can cause the creation of many more classes than the user specifically requests in order the create a valid class specification. One partial solution is to allow the user to specify a name to be used for the newly created class in the application in addition to the class name under which it was stored in the database. However, since the user cannot necessarily know in advance the names of all the classes that will be created in response to a request, it may be unrealistic to provide alternative names for all classes. The system checks to see if a class exists in the application before creating the class from the database. If the class exists in the application, it is assumed to be the correct class to use, even though there is no guarantee that this is the intention of the application user or that the class specification in the application and the database are compatible. For example, if the application user is retrieving a set of objects of the same class type from the database, then the retrieval of the first object will trigger the retrieval of the class specification. However, each following object will find the class specification already exists in the application and will not attempt to retrieve it again. This is the correct behavior under these circumstances. The system will detect an error if an attempt is made to retrieve an object using an incompatible class definition. The system is then faced with two alternatives, it can abandon the object retrieval process, or it can search the database system for an alternative class specification. If such a specification is found, the system can define an alternative class name for that class and continue to attempt to recreate the object. References to this class within the retrieved object must also be appropriately modified to use the alternative name.

A class specification may also include references to global variables. In this case, the variable must already exist in the application for the class to be successfully loaded. It is possible for the application user to query a class object in the database to determine the class properties so that the nature of the class can be determined before it is imported into the application. The member names and type and existence of nested classes and global variables can all be determined using

the multidatabase query language. Armed with this information, the application user can create global variables and make other preparations for the importation of the class from the database.

If the user chooses to import only a class specification, then the system must provide a special interface procedure to allow the creation of instances of this class during application runtime. Normally new object instances are created by calling a class constructor with the name of the class. The class name is recognized as a key word in the programming language when the application is created. However, the names of classes retrieved from the database are not available to the application at compile time and so an alternative way to create application objects must be provided. The user must be able to specify arguments to this object creation method. The number and type of the arguments must match one of the class constructor methods. The matching constructor will be used to create the object instance.

5.3.2 Object Retrieval

When an object is requested for retrieval from the database the first step taken is to ensure that the class specification exists in the application environment. Then, an instance of the object is created using the default constructor. Finally, the application object state is restored by assigning object members the values recorded in the corresponding database object. The assignment of member values is carried out recursively, so that if a member value is another object, this object is retrieved from the database and its member values set appropriately. If for any reason, a member value is not found in the database, that member value is set to null.

The retrieval procedure for widget objects must necessarily be modified since the default constructor for such objects requires a handle to the parent widget as an argument. There are two possibilities; the parent widget object can be found in the database, or the parent can be specified by the user as part of the retrieval request. Normally, the system will give priority to the parent specified by the user. If no parent is specified, then the system will look in the database for a parent object that was stored with the object and recreate this parent object. In this case, it is more appropriate to return the parent widget object to the user rather than the child widget requested, together with an appropriate warning indicator that the returned object is not the one requested. The application user should be aware that widget objects cannot be created without the specification of a parent widget and should be prepared for such behavior. If no parent widget can be identified, then the system will not be able to recreate the object and an error message will

be returned to the user. Widget objects also require special attention in that the attribute values for the object must be set according to the values stored in the database.

In general there are two ways in which an application user can identify an object in the database for retrieval. The first method uses the optional identifying name that was stored with the object as part of the storage procedure. The user can mark an object for retrieval by specifying its class name together with its identifying object name. The second method uses the member values of the object as identifying features. This allows the user to build their own proprietary identification features into any class. The objects are retrieved using the normal multidatabase query language. The system then provides a special translation method to convert a database object to an application object.

5.4 Implementation of the Object Storage System

The system for storing application objects as described above is implemented for the AthenaMuse® application authoring environment by building on the multidatabase facilities already developed for the environment. Although the multidatabase system provides an interface to hierarchical, relational and object database systems, the hierarchical system is not sufficiently flexible to allow the storage of application objects without significant effort. This effort is not justified by the low number of anticipated users. Hierarchical databases are usually legacy systems to which it is not desirable to add new data. Due to their popularity and availability, it is however important to ensure that the multidatabase system allows the storage of application objects in both relational and object databases.

5.4.1 Authoring Environment Support

In order to implement an object storage system within a particular authoring environment, it is important that the environment can provide all the base functionality needed. It must be possible for the environment to provide information, at runtime, concerning class definitions and object member values. This information is needed in order to store an object. In order to successfully recreate an object, the environment must provide a means of creating classes and instances at runtime. AthenaMuse® has implemented a system for the incremental parsing of applications in order to achieve this.

5.4.2 Implementation within the AthenaMuse® Environment

The existing multidatabase system is sufficient for almost all the data translation and representation needs of the object storage system. A new object storage class is added to the multidatabase to implement the storage and retrieval behavior. This new class is responsible for implementing the routines to extract class and object information and for storing the information in the database. It creates the special database classes that are needed to represent an application class specification and the compound base types. The storage class is responsible for the translation of data values between these database classes and application objects. The class also coordinates the recreation of class specifications and objects from the information stored in the database.

The internal details of the classes created especially for object storage will vary among database types. Relational database systems lack the object and set types that are usually found in object systems and so the representation of object and set application base types must be adjusted accordingly. The multidatabase storage class has a database independent component and a database dependent component similar to the other multidatabase connection and query classes. Translation routines to and from the application types and their equivalent database representation must be written for each component database. However, the remainder of the object storage and retrieval procedure is implemented at the database independent level. Much of the translation can be abstracted for a particular database type and so a relatively small amount of work is needed to add application storage capabilities to a particular database.

Object database systems require the least translation effort. Object member types can be directly translated to database object attribute types. Object databases generally include a set or sequence base type, which is used in representing the list and array data types. The any data type may pose a challenge as it represents a data instance of any of the application base types. However, this can be solved by providing an attribute in the database any class for each possible data type and an additional attribute to indicate which type is currently set. Individual databases may provide a more elegant solution. Generally object databases provide a generic object pointer which can be used to store references to objects of an arbitrary class type as may be needed to facilitate the storage of handle data types.

Relational database systems do not provide such a wealth of fundamental data types upon which to build. The multidatabase system ensures that each table definition in the system is viewed as a class specification and the extended multidatabase query language can be used to retrieve objects from the relational database. A combination of these techniques is used to store the application base types. A special class (table) is created for each of the simple base types integer, real, string and boolean, which is capable of storing the data information for that type. Every data row in these tables is assigned a unique identifier by the system which is stored as part of the row. Special classes are also created to store the remaining base types. However, the data in these more complex base types is represented by a database query. For example, a handle value that references a string is represented by a query that extracts that string, stored in the special string data class, from the database. The database class representing the handle type provides a text attribute for the storage of this query.

The database class created to represent an application class is modified to cope with the restrictions imposed by the relational data model. Each attribute of the database class corresponds to an object class member, except that the type of each field is set to be a text string. When an application object is stored in the table, a query that represents each data member is stored in the appropriate row and field of the table. The creation and interpretation of the query is determined by the translation routines for the relational database type. An object that contains a reference to another object also uses a query to store that reference in the database. The translation routine simply executes the query on the database to retrieve the object when necessary.

The most difficult type to represent in a relational database is the list. The number of fields in a database table is expected to be set in advance and cannot always be expanded at runtime. Each element in the list can be represented as a simple database query. For example, a string element may be referenced as 'select theString from theString where oid = 10'. These queries cannot be stored together in some form of delimited text block, as the length of the block might exceed the field length limits for some relational database systems. Instead, each query is stored in a separate field, with the maximum number of fields predetermined at the time of table creation. This has the undesirable effect of imposing a maximum size on the lists which can be

stored in a database. The array type is represented in a relational database by decomposing the array into a list of lists and storing the array value in the form of a query to retrieve this list.

Class inheritance also poses a representational problem for relational database systems. This is solved by replicating the fields in each table as they would be expected to be found according to the inheritance hierarchy. The list of base classes for each class is stored with the general class information so that each of the classes can be reconstructed correctly.

The object database implementation is clearly significantly more efficient than that of the relational database system, but the easy availability and lower cost of relational database systems often compensate for their slower performance. The other disadvantages of the relational system include the limit on the number of elements that can be stored in a list that must be predetermined in the structure of the table that represents the list data type. It is also nearly impossible for useful data to be retrieved from the database by a user outside of the application authoring environment since the translation process renders the data almost unreadable. However, by storing object references in the form of database queries, the system is taking advantage of the speed of data lookup that database systems are optimized to provide.

5.4.3 User Interface to the Object Storage System

The application user interface to the object storage system is via a group of specialized methods in the database class of the multidatabase system. All operations are carried out on the component database that is represented by the instance of the database class on which the methods are executed. There are three methods concerned with object storage:

StoreClassOnly(string *className*): boolean

This method stores the class definition for the class *className* and returns TRUE if the class is successfully stored.

Store(handle *hObject*) : boolean

hObject is a handle to an application object. This method stores the referenced object in the database and returns TRUE if the object is successfully stored.

StoreAs(handle *hObject*, string *objectName*) : boolean

This method stores the referenced object in the database together with an identifying name that can later be used to retrieve the object.

For example, in the rolodex application presented earlier in the chapter, the application user might want to store the class definition for the class rolodex in a database so that it is available for use by other users. There is little extra to be gained from storing the complete rolodex object in the database, except the user's preference in widget attributes such as width, height and color. The class does not contain any of the student information data. The code necessary to store the class is shown in Figure 5.2. The definition for class rolodex includes references to only predefined classes whose definitions do not need to be stored in the database. If the class student were to be stored in the database, the nested class address would also be automatically stored.

```
class rolodex{
        // class definition from Figure 5.1
};
DBdatabase dbase;
boolean success;
success = {'Open, "applications", "", "", 'UNISQL} => dbase;
if (success) {
        success = {'StoreClassOnly, 'rolodex} => dbase;
}
```

Figure 5.2 Sample ADL Code to Store a Class

The rolodex application can gain significant extra functionality from storing the instances of class student in a database. The query capabilities of the database can be used by the rolodex to extract a particular student object. It also allows other applications, not necessarily constructed using AthenaMuse® to access the student information. Sample code to create a student object and store it in a database is shown in Figure 5.3. It is chosen to store the student information in a separate database from the rolodex class. The address object which is automatically created for the dormAddress member of the student class will be stored in the database together with the student object. If, as in the example given, a valid object has been assigned to the member moreInfo, which is of type handle, then this object must also be stored in the database. This may mean storing the class definition for the information class if it is the first time that an object of this class has been stored in the database.

```
class student{
        // class definition from Figure 5.1
};
class information{
        string info;
};
DBdatabase dbase;
boolean success;
handle hStudent;
information theInfo;
hStudent = new {'Create, 'John, 'Smith, 52334} => student;
theInfo.info = "Active in the swim club";
hStudent->moreInfo = &theInfo;
success = {'Open, "students", "", "", 'UNISQL} => dbase;
if (success) {
        success = {'Store, hStudent} => dbase;
}
```

Figure 5.3 Sample Code to Store an Object

There are four methods concerned with object retrieval:

RetrieveClassOnly(string *className*) : handle

This method imports the class definition for the class *className* into the database environment and returns a handle to the class.

Retrieve(handle *hObject*, list *theList*) : handle

hObject is a handle to a database object of multidatabase class type DBobject. This method returns a handle to the newly created application object. The list argument is used to hold special constructor arguments, such as a handle to a parent widget when a widget object is being retrieved. Typically however this argument will be an empty list. The argument is chosen to be of type list to give flexibility for later development.

RetrieveByName(string *className*, string *objectName*, list *theList*) : handle

This method returns a handle to the application object of type *className* stored with the identifying name *objectName*. The list argument is again used to hold special constructor arguments.

CreateAppObject(string *className*, list *arguments*) : handle

This method returns a handle to an application object of type *className* constructed on the heap using the values in *arguments*. This method assumes that the class type has previously been imported from the database.

Continuing with the sample rolodex application, the sample code to retrieve the rolodex class from one database and search for a student object in another database is shown in Figure 5.4.

```
DBdatabase appsBase, infoBase;
boolean success1, success2;
handle hStudent, hRoloClass, hRolodex, hCursor;
string query;
success1 = {'Open, "applications", "", "", 'UNISQL} => appsBase;
success2 = {'Open, "students", "", "", 'UNISQL} => infoBase;
if (success1 && success2) {
    // recreate the class from information in the database
    hRoloClass = {'RetrieveClassOnly, 'rolodex} => appsBase;
    // create an instance of class rolodex using default constructor
    hRolo = {'CreateAppObject, 'rolodex, {}} => appsBase;
    // find the correct student in the database
    query = "select student from student where lastName = 'Smith'";
    hCursor = {'ExecuteStr, query} => infoBase;
    hObj = at (1, 'Next => hCursor);
    // convert database object to application object
    hStudent = {'Retrieve, hObj, {}} => infoBase;
    // display the student in the rolodex
    {'Refresh, hStudent} => hRolodex;
}
```

Figure 5.4 Sample Code to Retrieve a Class and Object

When a student object is retrieved from the database, the address object corresponding to the dormAddress member is also automatically retrieved. If the student object has been assigned a valid object for the member moreInfo, then this object will also be automatically retrieved and the moreInfo member handle set to reference this object. In this example, this means that an object of class type information would be retrieved from the database.

5.5 The Application Editor

In order to evaluate the advantages of providing for the storage of application objects, facilities for object storage are incorporated into the AthenaMuse® application editor that has been designed by Prof. Steven Lerman to provide an application author with automated help in creating his or her application. The editor is itself written in ADL, the AthenaMuse® authoring language and so the object storage techniques used in the editor are valid for any application.

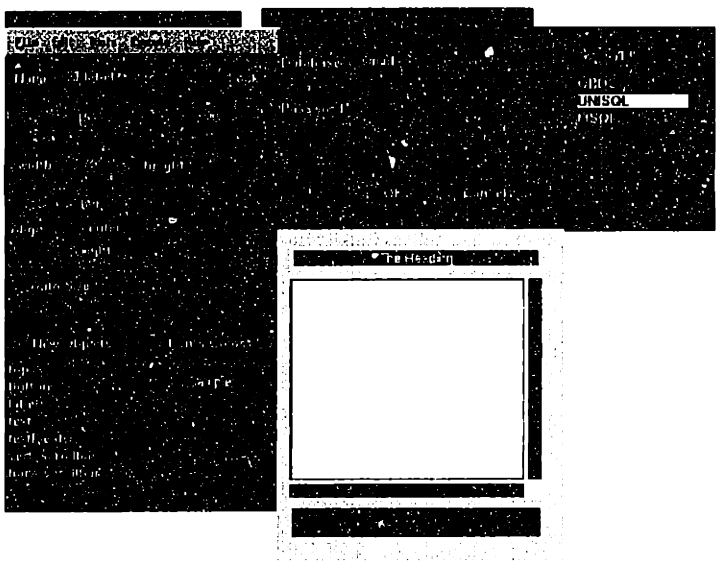


Figure 5.5 The Application Editor

The editor is composed of a set of 'smart classes' each of which represents a key element in an application environment, for example a base window, a button or an image holder. The application under construction in an editor is composed of a set of object instances of these smart classes. Storing this set of objects in a database is equivalent to storing the application. A container class with one member of type list is created to store the object set. Each application is stored as an instances of this class. The user is asked to choose a unique identifying name to associate with the application so that the object can be easily identified in the database. The editor provides the user with an option to load an application from a database, in which case the user specifies the identifying name of the application and each object in the corresponding list is retrieved and recreated in the editor environment.

Previous to including the option of database storage in the editor, all objects had to be translated into text and stored in a file. The user could supply a file name to the editor in order to load an application. The ability to store the application in a database provides significantly improved protection against accidental modification or deletion of the application. The database can often use password protection to restrict access to the application. The database can also mark an application as in use, so that multiple users cannot try to modify the application simultaneously. If the user wishes, the editor can choose to periodically save the application in the database with version information so that the user can retrace the application development process and revert to a previously saved version of the application if necessary.

Each object that makes up part of the application is stored independently in the database and can be individually retrieved. This is perhaps the most important improvement in editor functionality that is achieved by including database storage. The user can selectively choose objects of particular types from the database and quickly build a full application.

Chapter 6

Discussion

This thesis has examined the advantages to be gained from incorporating database system capabilities into application authoring environments. One possible design for such a database component has been suggested and implemented. This design was then used as a base for the construction of a system for application object storage and retrieval. It is important to examine the innovative aspects of this work together with the unresolved design issues that remain.

6.1 Contributions

The market for multimedia application authoring tools is expanding rapidly as potential application authors and publishers search for easy, reliable methods of presenting their ideas to an audience. As yet, there is no clear leader in the commercial marketplace. Typically a tool is chosen based on the functionality required by a particular application and the platform on which the application is expected to run. When the work in this thesis was begun, none of the available authoring environments offered built-in database support. Recently however, several database vendors, most notably Oracle, have begun to develop authoring tools to be used in conjunction with their individual database systems. For obvious commercial reasons, these tools are unlikely to provide full multidatabase support despite the many advantages that unlimited database access can provide to the application user. The design and implementation of the multidatabase component suggested in this thesis represents an important innovation in the field of authoring tools by providing application designers with the ability to organize complex application data with great efficiency and flexibility. It also provides an effective method of presenting an audience with information that is stored in existing database systems. The design and implementation of the multidatabase, although originally carried out for the AthenaMuse® authoring environment, should be easy to port to other object-oriented systems.

The common user interface can save the application author significant time and effort as there is no need learn the programming interface for each individual database, or to rewrite the same application so that it can function with multiple databases. The multidatabase system presented concentrates on providing the functionality needed by an application user and designer. The system provides a set of common database types, including multimedia types, and a common query language that allows the user to retrieve and modify information in the database. The decision to use a common object model allows the greatest flexibility in integrating component database systems. The successful integration of many component databases representing file, relational, object-relational and object-oriented systems shows that the system can be a practical reality. Previous multidatabase prototypes that have been proposed have advocated a similar object-oriented solution, but have failed to implement their models for the complete range of database types. These efforts have often concentrated on one particular aspect of multidatabase technology, such as concurrency control, global schema maintenance or the resolution of data semantic heterogeneity. The multidatabase system proposed in this thesis provides for the integration of these emerging technologies. The majority of application designers demand flexibility in their choice of database system, but require simultaneous access to multiple systems much less frequently. The system also allows for application environments where the database system is accessed by a user in a remote location. By combining the multidatabase with the AthenaMuse® cross-platform development and runtime environment, component database systems are not restricted to run on one particular machine architecture.

The multidatabase interface was developed specifically with the end user in mind. Significant effort was made to anticipate the needs of the user in the design of the object interfaces that collectively make up the multidatabase. The query object allows easy manipulation of a query specification by breaking the query into lists and allowing each list to be altered individually. This facilitates the easy programming of database browsing tools and repetitive database query execution. A full set of schema identification and manipulation tools is included to allow the user to connect to an unfamiliar database and still extract useful information. By including multimedia database types among the fundamental database types, the user is able to take advantage of automatic translation between database information and application media types, as well as advanced media processing procedures that can be built into the multidatabase system but are

missing from most component systems. Traditionally, multimedia database technology and multidatabase technology have remained separate; this thesis represents an effort to show how the technologies can be effectively used in combination.

In order to present a relational database component system to the user of the multidatabase, a complex mapping between the relational schema components (i.e. tables, fields, keys) and the object schema components (i.e. classes, instances, attributes) is maintained. With minimal assistance from the component database administrator the multidatabase can translate the component database schema so that it accurately reflects multimedia data stored in the database. As is shown in the development of application object storage capabilities, the multidatabase is able to expand the component relational database functionality so that it can store a variety of complex data types, including sets and object references. An important contribution of this research into multidatabase systems is to show how simple database queries can be used to represent these complex data types, capitalizing on the major strength of the system to execute queries efficiently. This extended functionality does not require extensively tailored code to be written for each individual component system. Instead, it is built upon the foundation provided by the fundamental common data types and query language developed for the multidatabase system. In addition, the object-oriented nature of the multidatabase design ensures that component databases of similar type can benefit from utilizing the same translation and mapping routines.

The second major contribution of this thesis is to overcome the boundaries placed between application behavior and application data by providing the ability to store both in a component database system. There are many advantages to be gained from storing application classes and objects in a database. The application components are made available for reuse, and components can be subjected to complex versioning and security control. Previously, the only method of storing application objects was to write the application in an object-oriented language with persistent object extensions. This is unrealistic for most application designers who do not wish to learn the complexities of a programming language such as C++, and would prefer to take advantage of an application programming language that provides higher level constructs for the advanced layout and control components typically needed by any application. The user can choose between storing a complete application object, or merely capturing the object behavior in

the database by storing only the object class specification. Important differences among the basic types of application objects were identified along with their needs for special treatment during the storage and retrieval process. Most importantly, it is shown that it is not necessary to use an object component database in order to achieve this functionality.

6.2 Future Directions

The multidatabase system proposed and implemented in this thesis provides a solid foundation for continuing research in this area. The storage of application objects is an example of one research project which would have been impossible to complete without the base functionality provided by the multidatabase. There are still many improvements that could be made to the multidatabase system, the most obvious of which is to continue to expand the number and type of component databases supported.

There has been some debate [Ram 1991] over the role that database standards can play in the development of multidatabase systems. Certainly, standards are not sufficient by themselves to solve all multidatabase issues. However, published research work on multidatabase systems has chosen to completely ignore the issue of standards. There is much to be gained by integrating a standard database interface as one type of component database. It immediately opens up participation in the multidatabase system to any database that conforms to the standards. Although the multidatabase is necessarily constrained by the functionality provided by the standard interface, it is still possible to create a separate component implementation for any database system that requires additional functionality that can only be implemented by interfacing with the system directly. The research into current and emerging standards also served to point out the shortcomings in these standards that will affect all application users that attempt to incorporate database access. The ODMG standard requires more attention to be paid to the issue of providing runtime schema identification and manipulation abilities. SQL3 could benefit from expanding the object model it uses to at least include object methods. It would also be extremely beneficial to all users if a common standard query language could be established for both standards. Separately or together, the SQL3 and ODMG standards will continue to evolve, and it will be important for the multidatabase system to both provide an interface to databases systems that support these standards and to comply with the standards where appropriate.

During the development of any common interface protocol, there must be trade-offs between the aim of achieving a universal set of interface functions and providing access to individual database functionality. One example of a solution to this compromise is the query interface in the multidatabase system, which allows for queries specified in the native query language in addition to the standard query language. Some database systems provide facilities for defining stored procedures or triggers. Currently these are inaccessible via the multidatabase system. However, future research might reveal a solution whereby the user could be alerted to additional functionality available in a particular component database and provided with a means to access it.

The multidatabase system would also benefit greatly from further research into the development of its multimedia capabilities. There is a wide range of media processing routines that could be added as methods to the media class definitions. These methods can then be referenced in a query specification, and sophisticated intelligent searches carried out on databases that may not possess any multimedia capabilities themselves. External, user specified routines could be used to process media in these situations so that immediate advantage could be taken of advances in processing technology. The user should not have to rely on, or wait for, updates to the multidatabase system.

There are several important research topics in multidatabase technology: global query processing; global schema maintenance; the semantic heterogeneity of data; and concurrency control; that require a multidatabase system for the testing and evaluation of possible solutions to these data management problems. The multidatabase system presented in this thesis is an excellent vehicle for this type of testing and evaluation. The modular design allows new functionality to be added easily and its ties to an application authoring tool allow prototype applications to be developed with ease. At the same time, the multidatabase system can be truly evaluated for the ease with which new functionality can be inserted. Perhaps the user will need to be given a choice as to which global integration algorithm to use when connecting to a set of component databases. Many algorithms require user input to accurately resolve integration issues. The application authoring environment would enable various user interfaces to be developed and tested to facilitate this decision making process.

There is still significant work remaining to refine the application object and storage procedures developed in this thesis. The application user would benefit from a greater freedom to

specify how particular application classes are interpreted and stored in a database. For example, the user may be aware that certain class member values need not be stored. These members can be ignored during the storage procedure in order to increase the speed performance and decrease the storage space required without compromising the validity of the stored object. The user may wish to design a special set of classes that have special object creation restrictions, similar to widgets, and will need to be able to convey this information to the object retrieval system. There is also room for improvement in the storage of values in relational databases. The entire multidatabase system would benefit from further performance analysis, testing and tuning.

The combination authoring tool and multidatabase environment is also eminently suitable for the development and testing of intelligent database agents. One obvious place for agent technology to be of use is in the resolution of database integration issues. The agent can base its decisions on those of previous users. Alternatively, such agents might be developed to record a user's database access patterns across many database systems and use this knowledge to predict future access patterns. By building a set of relationships between user queries and data retrieval patterns the agent can build a web of connections between keywords and the data in each database. This information can be used to prompt a user with suggestions as to where else he or she might look for information related to his or her current query. A simple agent might simply be responsible for maintaining a list of databases, together with appropriate connection information, for the user to browse and choose from.

A parallel research direction which is deserving of significant effort is the evaluation of the added benefit that applications can derive by exploiting the database facilities offered by the multidatabase system. In educational settings, the database can provide a student with the opportunity to search through large image and text repositories that would otherwise remain unknown to them. It can provide a teacher with a means of following and recording a student's progress through an application and of tailoring an application's response accordingly. A database can be used to store a student's own research results and notes. Significant potential may exist for co-operative learning environments where databases of information are built up through a group effort and knowledge can easily be shared remotely by providing access to a central database repository.

Conclusion

This thesis has examined the database system requirements of an object-oriented application authoring environment and developed a multidatabase component suitable for inclusion in this environment. Application designers, whether they are creating an electronic textbook or a simple media viewer, can benefit greatly from using a database to organize application data or keep track of user preferences. There is an entire class of applications, those that provide graphical user interfaces to existing database systems, that can only be authored in an environment that provides database support.

The most important aspect of the multidatabase is that it provides an application designer with a standard interface to all supported component databases. A common object data model and query language, together with a set of standard base data types, are developed for the multidatabase and all other component databases are translated to and from this model. The application designer is concerned only with the interface to the common model and can remain ignorant of individual database protocols. The implementation of the multidatabase includes examples of hierarchical, relational and object component systems. The multidatabase is designed to provide all the functionality anticipated to be needed by the designer, including the ability to build queries incrementally and access database schema information. The common model greatly simplifies the process of transferring data between databases and constructing queries across databases.

The presence of a multidatabase component in an authoring environment enables further innovative research into the part played by a database within an application. Traditionally, applications have been divided into two parts, the application data and the application behavior. Significant improvement in application functionality can be achieved by extending the multidatabase to accommodate the storage and retrieval of application objects in addition to the data. Application objects can then be made available for reuse, and object version and security control becomes straightforward.

Appendix A

This appendix details the application programming interface to the multidatabase module of AthenaMuse®. Where appropriate, short pieces of ADL are included to indicate probable uses of a class method. The methods of a class are given in alphabetical order for convenience.

Standard Data Types

The following simple types are available in the multidatabase:

- string
- integer
- real

The following complex data types are also available and are implemented as object classes. The class interface for each type is described later in this document:

- date
- time
- timestamp
- set
- monetary
- binary
- media
- image

These names should be used when a class method requires a type to be specified. In addition, when appropriate, any valid class name is accepted as a type.

DBdatabase

This is the principle class and represents a connection to a component database. Methods are available on the class to perform schema identification and modification, to control query execution, transaction management and to store application objects in the component database.

AddAttribute (string *className*, string *attrName*, string *attrType*) : boolean

Adds an attribute *attrName* of type *attrType* to the class *className*. In a relational database this is equivalent to adding a field to a table. Returns TRUE if the attribute is added to the class, otherwise FALSE.

```
// assume dbase is an instance of DBdatabase in all code examples
// and that a connection to a database has been successfully
// established by calling the Open method.
boolean success;
success = {'AddAttribute, 'person, 'sibling, 'person} => dbase;
```

Commit ()

Causes the current transaction to be committed and automatically begins a new transaction. Only applies to those database systems that support transactions.

CreateAppObj (string *className*, list *arguments*) : handle

This method returns a handle to an application object of type *className* constructed on the heap. The method is used to create instances of classes retrieved from the database. The number and type of the values in *arguments* determines which constructor is used. If no *arguments* is an empty list, the default constructor is used.

CreateClass (string *className*, list *attributes*) : boolean

Causes the creation of a new class called *className* in the database schema. The list *attributes* contains a list of lists. Each of these sublists is made up of two strings, the first specifies the attribute name and the second the attribute type. Returns TRUE if the class is created, FALSE if creation fails.

```
list attrList = {{'name, 'string}, {'age, 'integer}};
success = {'CreateClass, 'person, attrList} => dbase;
```

CreateObj (string *className*, list *attributes*, list *data*) : handle

Creates an instance of the class *className* in the database and returns a handle to the newly created database object. The returned handle will be a handle to an object of class **DBobject**. The list *attributes* contains a list of attribute names for which initial data values will be specified. The list *data* contains the corresponding data values in the order specified in the list of attribute names.

```
handle hPerson;
hPerson = {'CreateObj, {'name, 'age}, {'John, 14} } => dbase;
```

CreateSubClass (string *className*, list *superClasses*, list *attributes*) : boolean

Similar to **CreateClass**, but allows the specification of a list of class names from which the newly created class should inherit. The list may contain no class names.

```
success = {'CreateSubClass, 'student, {'person}, {'school, 'string}} => dbase;
```

DeleteClass (string *className*) : boolean

Removes the definition of the class named *className* from the database schema. Returns TRUE if deletion is successful, otherwise FALSE. Also necessarily removes all objects of this class.

Execute (handle *hQuery*) : handle

Executes the query specified by *hQuery*, a handle to an object of type **DBquery**. The query specification is pre-processed appropriately for the component database type before the query is executed. A handle to a **DBcursor** object is returned from which the results may be retrieved.

```
handle hResult;
DBquery query;
{'SetText, "select age from person where name = 'John'"} => query;
hResult = {'Execute, &query} => dbase;
```

ExecuteStr (string *query*) : handle

The string *query* is send directly to the database for execution. No pre-processing of the query is carried out. A handle to a **DBcursor** object is returned.

FindClass (string *name*) : handle

A handle to a **DBclass** object, representing the class *name* that is defined in the database schema, is returned from which the class properties may be retrieved.

```
handle hClass;
list attributes;
hClass = {'FindClass, "person"} => dbase;
attributes = 'GetAttributes => hClass;
```

GetAllClasses () : list

A list of the names of all the classes defined in the database schema is returned.

GetBaseClasses () : list

Returns a list of the names of all the base classes (classes that have no superclasses) that are defined in the database schema.

GetPersistentRoot (string *rootName*) : handle

Returns a handle to the database object pointed to by the persistent root named *rootName*, or NULL if the root does not exist in the database. The handle is to an object of class **DBobject**.

GetPersistenRoots () : list

Returns a list of the names of all the persistent roots defined in the database. The list will be empty if no roots are defined or if the database does not support the notion of persistent roots.

Login (string *userName*, string *password*) : boolean

Enables the user to specify a username and password when connecting to the database. This method is usually called before a connection to the database is opened. It can also be called later to change the username under which database operations are carried out. Returns TRUE if the username and password are accepted successfully.

Open (string *databaseName*, string *server*, string *location*, string *type*) : boolean

Opens a connection to the database named *databaseName* that is located on the machine *server* and can be located on that machine by means of *location*. If server is an empty string then the database is presumed to reside on the local machine. Similarly *location* is an empty string when it is not required to specify the database. *type* is a code representing the type of database system. Currently supported types are ODBC, O2, PG95, UNISQL, MSQL and FILE. Each type is described more completely after the description of the class interface. The method returns TRUE if a connection is established, FALSE otherwise. Typical usage would be:

```
DBdatabase dbase;
boolean success;
success = {"Open", "ProjectDB", "ceci.mit.edu", "/mit/kate/databases", "UNISQL"} => dbase;
```

Store (handle *hObject*) : boolean

Stores an ADL object instance, referenced by *hObject*, to be stored in the database so that it can be retrieved later. The class information for the object is stored first if this is the first instance of that class to be stored. Returns TRUE if the object is stored successfully.

StoreAs (handle *hObject*, string *objName*) : boolean

Similar to **Store**, but stores the object together with a persistent name *objName* for the object with which it can be identified for later retrieval.

StoreClassOnly (handle *hObject*) : boolean

Stores only the ADL class definition for the class of *hObject* and does not store the individual instance information. Returns TRUE if the class is stored successfully.

RefreshStructure ()

Reread the database schema definition from the database. This is only necessary if updates are made outside of the multidatabase environment. Changes made within the environment are automatically reflected in the schema.

Retrieve (handle *hObject*, list *theList*) : handle

Retrieves an ADL object instance from the database which is referenced by the database object *hObject*. The list is used to hold special constructor arguments, such as a handle to a parent widget when a widget object is being retrieved.

RetrieveByName (string *className*, string *objName*, list *theList*) : handle

Retrieves an ADL object instance from the database which is of class *className* and which is identified by *objName*. If the object cannot be found in the database a NULL handle is returned.

RetrieveClassOnly (string *className*, handle *parentClass*) : handle

Retrieves the definition of the application class *className* from the database and loads it into the environment so that instances of that class can be created. The class is created as a nested class of *parentClass*. A handle to the class is returned. This handle will be NULL if the class is not successfully created.

DBclass

This class represents a class definition in a database. A class has superclass, attribute, method, key and extent properties associated with it. Methods are provided to access these properties.

GetAttributes () : list

A list of the attributes defined for the class is returned. The list is composed of two sublists. The first sublist is a list of attribute names. The second is a list of the data types of these attributes.

```
list attrList = 'GetAttributes => hClass;
// attrList would be {{'name, 'age, 'sibling},{'string, 'integer, 'Person}}
```

GetMethods () : list

Returns a list of all the methods defined for the class. The returned list contains a sublist for each method. This sublist itself contains three elements. The first element is the name of the method. The second is a string specifying the return type of the method. The third element is a list of strings specifying, in order, the parameter types for the method.

GetSubClasses () : list

Returns a list of the names of all the subclasses of the class.

GetSuperClasses () : list

Returns a list of the names of all the superclasses of the class.

DBcursor

This class is used to access the results of a query execution. The class also provides information as to the format of the query result set. Each entry in the result set is represented as a list. The cursor class can be thought of as a pointer that moves forward and backwards in this list. As an example, consider the following code:

```
handle hCursor;
hCursor = {'ExecuteStr, "select name, age from person where name = 'John'"} => dbase;
// assuming that are three instances of person in the database that match this query, the result
// set would be
// { {'John, 14} , {'John, 9} , {'John, 12} }
```

ColumnCount () : integer

This method returns the number of elements in an entry of the result set.

```
integer count;
count = 'ColumnCount => hCursor
// count would now equal 2
```

IsLast () : boolean

Returns TRUE if the cursor is currently positioned at the last result in the set. A subsequent execution of the method **Next** would result in an error.

Next () : list

Moves the cursor forward one entry in the result set and returns the value of that entry.

```
while('IsLast => hCursor != TRUE) do {
        result = 'Next => hCursor;
        echo(at(1, result)); }              // always prints "John" in this example
```

OpenAt (integer *position*) : list

Moves the cursor to the result entry at *position* and returns that entry.

Position () : integer

Returns the position to which the cursor currently points.

Prev () : list

Moves the cursor to the previous entry in the result set and returns that entry.

RowCount () : integer
Returns the number of entries in the result set.

Success () : boolean
Returns TRUE if the query executed successfully, otherwise FALSE. This value should be checked before attempting to retrieve the results.

TypeList () : list
Returns the data types of the elements of a result entry.

```
list types;
types = 'TypeList => hCursor;
// in this example, types would now equal {'string, 'integer}
```

DBquery

This class represents a database query. The multidatabase uses a standard query format that is based on SQL-92 with extended functionality to accommodate object database systems. A typical SQL query has four parts, each part represented by one line in the sample query below. The query finds the name, age and salary of all people called John who are working on project 3345 and orders the result set by salary amount. The keyword in each line is given in bold face.

select name, age, salary
from person, project
where name = 'John' and projectID = 3345
order by 3

The query class enables the query to be specified as a string or built up from its component parts. The query is subject to pre-processing by the multidatabase system. See the description of the supported database system types at the end of this document for more information on the type of processing carried out,.

Bind (integer *n*, any *value*)
A parameter in a query can be specified by a question mark (?) when the query is constructed. This mark must be replaced by an appropriate value before the query is executed. This method replaces the parameter marked by the *n*th question mark with *value*.

```
handle hCursor;
DBquery query;
{'SetText, "select name from person where name = ?"} => query;
{'Bind, 1, 'John} => query;
hCursor = {'Execute, &q} => dbase;
{'Bind, 1, 'Robert} => query;
hCursor = {'Execute, &q} => dbase;
```

BindAll (list *values*)

This function replaces all the question marks in the query with the values from the supplied list in order.

SetQryFrom (list *fromList*)
Sets the list of class names from which the query result will be generated.

```
{'SetQryFrom, {'person, 'project}} => query;
```

SetQryOrderBy (list *orderList*)
Sets the order in which the results will appear.

```
{'SetQryOrderBy, {2}} => query;
```

SetQrySelect (list *selectList*)
Sets the list of attributes from which the query result will be generated.

```
{'SetQrySelect, {'name, 'age, 'salary}} => query;
```

SetQryText (string *queryText*)
Sets the text of the entire query. The query will automatically be broken down into its component parts.

SetQryWhere (list *whereList*)
Sets the list of conditions which govern result generation.

```
{'SetQryWhere, {"name = 'John'" , "projectID = 3345}} => query;
```

GetQryFrom () : list
Returns the list of classes from which the query result will be generated.

GetQryOrderBy () : list
Returns the order in which the results will be listed.

GetQrySelect () : list
Returns the list of attributes from which the query result will be generated.

GetQryText () : string
Returns the complete text of the query. This will be automatically generated from the component parts if necessary.

GetQryWhere () : list
Returns the list of conditions which governs result generation.

DBbinary

This represents the binary data type. It is assumed that the data is either stored directly as bytes or as a reference to an external file that holds the bytes. If the data is stored in an external file, then the file location can be retrieved. The location has three components, a name to identify the machine on which the file resides, a path to locate the file on that machine, and the name of the file itself. Currently, since there is no means of returning a pointer to a binary stream of data to the ADL program directly, the only option with binary data is to store it in a temporary file.

Create (string *pathName*, string *fileName*, string *hostName*) : handle

Returns a handle to a DBbinary object where the binary information is stored in the file location specified by the method parameters.

```
handle hBinary;
hBinary = new {'Create, "/mit/kate/data", "mydata.bin", "ceci.mit.edu"} => DBbinary;
```

CreateAsBinary (string *pathName*, string *fileName*, string *hostName*) : handle
Creates a DBbinary object where the binary information is read from the file location specified by the method parameters and stored internally to the object.

GetFile () : string
If the binary data is stored as a reference to an external file, returns the external filename.

GetHost () : string
If the binary data is stored as a reference to an external file, returns machine name where the file is stored.

GetPath () : string
If the binary data is stored as a reference to an external file, returns the path to the external file.

IsBinary () : boolean
Returns TRUE if the binary data is stored internally to the object. Returns FALSE if the binary data is stored as a reference to an external file.

DBmedia inherits from DBbinary

This class represents media data that is stored either internally in binary form, or as a reference to an external file.

Create (*string* pathName, *string* fileName, *string* hostName) : *handle*
Returns a handle to a DBmedia object.

DBimage inherits from DBmedia

This class represents image data specifically.

Create (*string* pathName, *string* fileName, *string* hostName) : *handle*
Returns a handle to a DBimage object.

GetImage () : *handle*
Returns a handle to an MMimage object that has been constructed from the information contained in the DBimage object.

DBdate

This class is used to represent the date data type. The data type is composed of three components, an integer from 1 to 12 representing the month, an integer from 1 to 31 representing the day, and a four digit number representing the year.
Create (integer *month*, integer *day*, integer *year*) : handle

Returns a handle to a DBdate object.

GetDateList () : list
Returns the date in the form of a list with three elements, month, day and year in that order.

GetDateString () : string
Returns a string representation of the date in the form dd/mm/yyyy.

SetDate (integer *month*, integer *day*, integer *year*)
Sets the date represented by the DBdate object.

DBmonetary

This class is used to represent the monetary data type. Currently the only information available is the amount of money represented by an object of this class. Currency information is not yet incorporated into the class.

Create (double *amount*) : handle
Returns a handle to a new DBmonetary object representing *amount*.

GetAmount () : double real
Returns a real representing the monetary amount.

SetAmount (double *amount*)
Sets the monetary amount to *amount*.

DBobject

This class represents a database object. The object will be an instance of a particular database class, which can be found from querying the object. The object has methods which allow manipulation of its attributes and methods.

Call (string *methodName*, list *parameters*) : any
Call the method *methodName* on the object with the parameters given in the list *parameters*. The method returns the result of the method invocation.

```
// assume hObj is a handle to a valid database object of type person
real wage;
wage = {'Call, 'GetMonthlyWage, {'January}} => hObj;
```

Drop () : boolean
Delete the object from the database. Returns TRUE if the deletion is successful.

```
boolean success;
success = 'Drop => hObj;
```

GetAttribute (string attrName) : any

Returns the value of the attribute named *attrName* for the object.

```
integer age;
age = {'GetAttribute, 'age} => hObj;
```

Name () : string

Returns the name of the object class to which the object belongs.

SetAttribute (string attrName, any attrValue) : boolean

Sets the value of the attribute named *attrName* to *attrValue*. Returns TRUE if the attribute is successfully set, otherwise FALSE.

```
{'SetAttribute, 'age, 15} => hObj;
```

DBset

This class represents the set data type. A set is used to hold a collection of data types. Duplicate data types are allowed within the set. To the ADL programmer, a set can be viewed as a list of database objects or values.

AddElement (any *element*)

Adds a data value to the set. The type of *element* must be a valid database type, either a simple type e.g. integer or a handle to a complex type e.g. **DBobject**.

Create (list *setElements*) : handle

Returns a pointer to a new **DBset** object whose contents are initialised to the elements contained in *setElements*. Again, the type of the elements of this list must be a valid database type.

```
handle hSet;
list setOfNumbers = {1, 4, 3, 5};
hSet = new {'Create, setOfNumbers} => DBset;
{'AddElement, 9} => hSet;
```

GetList () : list

Returns the list of elements in the set.

SetList (list setElements)

Sets the list of elements in the set to *setElements*.

DBtime

This class represents the time data type. The time data types has three components, an integer between 0 and 23 representing the hour, an integer between 0 and 59 representing the minute and an integer between 0 and 59 representing the seconds.

Create (integer *hour*, integer *minute*, integer *second*) : handle

Returns a handle to a new **DBtime** object.

GetTimeList () : list

Returns a list composed of the three elements that make up the data type.

GetTimeStr () : string
Returns a string in the form "hh:mm:ss".

SetTime (integer *hour*, integer *minute*, integer *second*)
Sets the time of the data type.

DBtimestamp

This class represents the timestamp data type which consist of a time and a date together represented by a data value of type **DBtime** and a data value to type **DBdate**.

Create (handle *date*, handle *time*) : handle
Returns a handle to a new **DBtimestamp** object.

GetTstampList () : list
Returns a list composed of two elements, a handle to a **DBdate** object and a handle to a **DBtime** object.

GetTstampStr () : string
Returns a string of the form "mm/dd/yyyy hh:mm:ss".

SetTstamp (handle date, handle time)
Sets the values for the timestamp data type.

Database System Types

A short description of the way in which the supported database systems are treated in the multidatabase environment and the features of the multidatabase that are available to each system.

File System

The file system is one type of hierarchical database. In the multidatabase system it is represented as a database with two object classes, file and directory. The class definition for each is:

```
class file {
        string name;                    // the filename
        string root;                    // the directory path to the file
        boolean isReadOnly
        boolean isLink

        handle GetFile()                // return handle to Osfile
        handle ParentDir()              // return handle to directory object
        string FullPath()               // the full name specification
};


class directory : file {
```

```
        setof(file) files;                  // the files contained in the directory
        setof(directory) subdirectories     // the sub-directories
        setof(string) fileNames
        setof(string) subDirNames;
};
```

A connection to a directory system is opened by specifying the path name of the desired root directory of the filesystem as the database name. The system automatically generates one persistent root for the system, named 'root', which is a handle to the root directory object.

Relational Database System

Each table of a relational database system is imported as a class in the multidatabase environment. Each field in the table becomes a class attribute and each row becomes an object of that class. When constructing queries using the **DBquery** class, queries such as 'select person, person.name from person where person.age = 15' become allowable. Each result entry has two elements, a handle to a **DBobject** of type person and a string.

In addition, the database administrator or user can set up a table in the database called 'am2types'. The table has five fields, all of type string, that contain information that enables the multidatabase to identify multimedia embedded within the database.

am2types

type	table	newAttrName	pathAttr	fileAttr
image	person	photo		photo
image	photos	filename		filename
binary	files	data	pathname	filename

For example, the photo field in the table named person would be interpreted as an image of datatype DBimage in the multidatabase, the file location of which is stored in the photo attribute. The data attribute of the table named 'files' would be interpreted to be of type DBbinary, where the binary data is found in the external file referenced by the pathname and filename attributes.

Object Database Systems

The class hierarchy created in the object database system is available directly in the multidatabase system. The extended query language available in the multidatabase system allows method invocations and attribute paths to be included in the query specification.

Bibliography

Ahmed R. et al., "The Pegasus Heterogeneous Multidatabase System," *IEEE Computer*, December 1991.

Albert J. et al., "Automatic Importation of Relational Schemas in Pegasus," *Research Issues in Data Engineering: Interoperatibility in Multidatabase Systems*, April 19-20, 1993.

American National Standards Institute, *America National Standard for Information Systems: Database Language - SQL: ANSI X3.135-1992*, American National Standards Institute, 1992.

Barbara D. and Clifton C., "Information Brokers: Sharing Knowledge in a Heterogeneous Distributed System," *Database and Expert Systems Applications 4th International Conference*, September 1993.

Bertino E., Negri M., Pelaggati G. and Sbatella L., "Integration of Heterogeneous Database Applications through an Object-Oriented Interface," *Information Systems*, Vol.14, No.5, 1989.

Bertino E. and Martino L., *Object-Oriented Database Systems: Concepts and Architectures*, Addison-Wesley, 1993

Bancilhon F., Delobel C. and Kanellakis P., *Building An Object-Oriented Database System: The Story of O_2*, Morgan Kaufmann, 1992.

Booch G., *Object-Oriented Analysis and Design*, Benjamin/Cummings, 1994.

Borkin S.A., *Data Models: A Semantic Approach for Database Systems*, The MIT Press, 1980.

Bright M.W., Hurson A.R. and Pakzad S.H., "A Taxonomy and Current Issues in Multidatabase Systems," *IEEE Computer*, March 1992.

Bright M.W., Hurson A.R. and Pakzad S.H., "Automated Resolution of Semantic Heterogeniety in Multidatabases," *ACM Transactions on Database Systems*, Vol.19, No.2, June 1994.

Browne S., "Communication and Synchronization Issues in Distributed Multimedia Database Systems," *Lecture Notes in Computer Science*, No. 759, 1993.

Bucci G., Detti R., Pasqui V. and Nativi S., "Sharing Multimedia Data Over a Client-Server Network," *IEEE Multimedia*, Fall 1994.

Buddhikot M.M., Parulkar G.M. and Cox J.R., "Design of a large scale multimedia storage server," *Computer Networks and ISDN Systems*, Vol.27, 1994.

Bukhres O.A., Chen J., Du W., Elmagarmid A.K. and Pezzoli R., "InterBase: An Execution Environment for Heterogeneous Software Systems," *Computer*, Vol.26, No.8, August 1993.