VIRGINIA POLYTECHNIC INSTITUTE

CYRANO: A META MODEL FOR FEDERATED DATABASE SYSTEMS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DEPARTMENT OF COMPUTER SCIENCE
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

BY JOSEPH DZIKIEWICZ

ALEXANDRIA, VIRGINIA
MAY 1, 1996

APPROVED:

C. Egyhazy, Chairman

W. Frakes

S. Gupta

J. Roach

K. Triantis

CYRANO: A META MODEL FOR FEDERATED DATABASE SYSTEMS

by:

Joseph Dzikiewicz

Chairman: C. Egyhazy
Department: Computer Science

Abstract

The emergence of new data models requires further research into federated database systems. A federated database system (FDBS) provides uniform access to multiple heterogeneous databases. Most FDBS's provide access to only the older data models such as relational, hierarchical, and network models.

A federated system requires a meta data model. The meta model is a uniform data model through which users access data regardless of the data model of the data's native database.

This dissertation examines the question of meta models for use in an FDBS that provides access to relational, object oriented, and rule based databases. This dissertation proposes Cyrano, a hybrid of object oriented and rule based data models. The dissertation demonstrates that Cyrano is suitable as a meta model by showing that Cyrano satisfies the following three criteria:

1) Cyrano fully supports relational, object oriented, and rule based member data models.

2) Cyrano provides sufficient capabilities to support integration of heterogeneous databases.

3) Cyrano can be implemented as the meta model of an operational FDBS.

This dissertation describes four primary products of this research:

1) The dissertation presents Cyrano, a meta model designed as part of this research that supports both the older and the newer data models. Cyrano is an example of analytic object orientation. Analytic object orientation is a conceptual approach that combines elements of object oriented and rule based data models.

2) The dissertation describes Roxanne, a proof-of-concept FDBS that uses Cyrano as its meta model.

3) The dissertation proposes a set of criteria for the evaluation of meta models. The dissertation uses these criteria to demonstrate Cyrano's suitability as a meta model.

4) The dissertation presents an object oriented FDBS reference architecture suitable for use in describing and designing an FDBS.

Dedication

For Julie, as is all else.

Acknowledgements

Completing this work has been a lengthy and sometimes arduous process, and I never would have gotten this far without the guidance and patience of Csaba Egyhazy, my advisor. Thus first thanks must go to him.

Additional thanks go to Julie Dzikiewicz, for keeping me working on this dissertation when I should have, for Mike Airey, for letting me work on it when I shouldn't, and to Andy, Kate, and Diana Dzikiewicz, for keeping me from working on it when I didn't really want to.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

Federated database research examines ways of integrating multiple independent heterogeneous databases. Several existing systems integrate relational, hierarchical, network, and flat file database systems. These provide a uniform user access to data stored in any of the underlying databases.

Recently, the emergence of object oriented and rule based data models complicates this picture. Earlier systems, which use a relational or semantic data model as meta model (or integrating model), cannot be easily extended to allow access to the full range of functionality provided by the newer data models. As a result, there is a need to examine federated database issues in light of these new data models.

This dissertation examines the question of the meta model for a federated database system (FDBS) that supports relational, object oriented, and rule based databases as members of the federation. This dissertation presents Cyrano, a hybrid of object oriented and rule based data models designed for use in such an FDBS. In examining Cyrano, this research proposes the use of the following three criteria for the evaluation of meta models:

1) A meta model must fully support the translation from member database to gateway database.

2) A meta model must fully support the federation of gateway databases into a global database.

3) It must be possible to implement the meta model as part of an FDBS.

Based on these criteria, this dissertation concludes that Cyrano is a suitable meta model for an FDBS supporting relational, object oriented, and rule based member models.

1.1. Background

The background needed to address the central theme of this research is found in four related research areas:

1) *Data models.*

Data models include mechanisms for structuring data and operations for accessing the structured data.

2) The *database heterogeneity* problem.

Database heterogeneity encompasses the ways in which databases can be different. The database heterogeneity problem arises when users need access to data stored in multiple heterogeneous databases.

3) *Federated database systems.*

A federated database system (FDBS) provides a uniform interface to the data facilities of several independent member databases.

4) *Meta models for FDBS's.*

A meta model is the unifying data model of an FDBS. The user or application accesses the data in the FDBS via the meta model.

Chapter 2 gives a detailed survey of existing research into these areas, identifying the research gap filled by this dissertation. The remainder of this section provides a detailed summary of the information found in chapter 2.

1.1.1. Data Models

A data model is a collection of three elements [Codd 81]:

1) A set of data structure types used to construct complex data items. These structures can be relations, records, classes, or other structure types.

2) A set of operators which can be applied to these data structures. These include operators to define data templates from the structure types,

operators that store data, and query operators that retrieve data from stored data structures.

3) A collection of integrity rules which identify the acceptable database states. These include rules specifying valid values for database keys and rules for identifying data structures that can be stored.

Several data models have been defined. The earliest of these include the network and hierarchical models, which store data in linked records. The relational model stores data in tables or flat relations. More recently, the object oriented model stores data as complex data objects with their structure defined by classes. Rule based models store data as relations, but support operators that allow for the use of complex logical rules to retrieve data.

1.1.2. Database Heterogeneity

The heterogeneous database problem arises when an application needs to access data from multiple independent databases. It has become a focus of recent database research, and is the subject of special issues of "IEEE Computer" [1991] and "ACM Computing Surveys" [1990]. Examples of the problem abound:

- A corporation's business units have independently developed accounting systems, either because of decentralization or because newly acquired businesses have their own systems. In order to develop an overall business plan, the corporation needs access to all of the accounting data of its subsidiaries.

- An intelligence agency has over time acquired a number of legacy information systems, each of which tracks information about a different part of the world or subject area during a specific period of time. The agency wishes to examine all of this data with the intent of looking for patterns that cross geographical boundaries. To do this, analysts must be able to

examine all of the data using a unified system.

- An information retailer provides on-line access to a number of diverse data sources. Each of these sources is provided by a different information wholesaler, and each has its own access protocols and storage formats. The retailer wishes to provide its clients with a uniform interface to these diverse data sources.

Each of these applications requires access to data in diverse databases. The differences between these databases are called database heterogeneity. There are three primary dimensions of database heterogeneity relating to differences in data model, value, and structure.

1) Model differences relate to differences in the underlying data models of the databases. These include differences in the data structures and operators provided by different models. Model differences can be simple syntactic differences in access protocols or fundamental differences in database capabilities. Model differences have arisen as the result of many years of research into the design of data models. Ullman [1989] presents an overview of that research.

2) Data value differences arise when the data in a database is obsolete, incorrect, or incomplete. For example, a database may have an obsolete address for a person, while another contains only the person's first and last names and not the middle name. Value differences also include differences in units or data type, such as the representation of temperature in degrees Fahrenheit, Celsius, or Kelvin.

3) Structural differences relate to the ways in which different databases structure similar data. For example, one database maintains a table of people and their addresses, while another stores names and addresses in separate tables. Kim and Seo [1991] provide a detailed

examination of structural and value differences.

1.1.3. Federated Database Systems and Architectures

Ironically, the literature on heterogeneous databases is itself fairly heterogeneous in its use of certain terms. In particular, the terms "federated database system" and "multi-database" have different meanings when used by different authors. This dissertation adopts the terminology presented by Sheth and Larson [1990]. The following definitions are taken from that reference.

Multi-database systems are one solution to the heterogeneous database problem. A multi-database is a collection of databases that allows users to access data from any database in the collection. Member databases have varying degrees of heterogeneity and can be distributed geographically. Queries against the multi-database result in data being retrieved from one or more member databases.

A federated database system (FDBS) is a multi-database in which the individual member databases possess a large degree of autonomy. Autonomous databases have their own schemata and security restrictions and are generally controlled by some external authority. An FDBS is an appropriate solution when applications need to access multiple independent and heterogeneous databases.

A typical FDBS provides a single user interface for accessing data in autonomous and heterogeneous member databases. This is known as a 1:N architecture, in which one user interface provides access to N databases. Except where noted, this dissertation assumes that an FDBS is based on this 1:N architecture. (Hsiao and Kamel [1989] provides a thorough discussion of different heterogeneous database architectures.)

Much of the literature on FDBS's describes systems in terms of specific reference architectures [Sheth and Larson 1990; Deen et al. 1985]. A typical reference architecture describes an FDBS in terms of its schemes or data

structures and its data processing elements. This is similar to the traditional separation of software design into data versus function. This dissertation argues that this separation leads to a flawed view of the resolution of data heterogeneity, a view that emphasizes structure resolution and ignores differences in querying capabilities. In accord with this, this dissertation presents an object oriented reference architecture that emphasizes data manipulations over data structures.

1.1.4. Meta Models

A primary function of the FDBS is to present data from the member databases in terms of a common data model. The common or canonical data model of an FDBS is also called a meta model. Meta is a Greek prefix that translates roughly into "behind" or "after." Thus, a *meta model* is a model behind or after existing data models.

The FDBS maps data from the member databases into the common representation provided by the meta model. Thus, a meta model must be capable of expressing the data relationships represented by the underlying data models in the federation. This is analogous to the addition of fractions with different denominators, where the incompatibilities are handled by a common denominator. Once the fractions are interpreted in terms of the common denominator, one operates on them as if they were compatible fractions. In order to fully support its member models, a meta model must be able to capture the structural and behavioral characteristics of all the different databases participating in the federation.

There have been several FDBS's developed for both research and production use. The first wave of these generally support relational, hierarchical, and network DBMS's as members of the federation. These usually used a variation of the relational model as a meta model. Among these systems are ADDS [Breitbart et al. 1986; Thomas et al. 1990], Mermaid [Templeton et al. 1986;

Thomas et al. 1990], and Dataplex [Chung 1990; Thomas et al. 1990]. While they support early target members such as hierarchical, relational, and network models, they are incapable of supporting object oriented and rule based databases, the newer models.

The second wave of FDBS's use modern information processing strategies to integrate databases. For example, Pegasus uses an object oriented data model as the meta model [Ahmed et al. 1991]. Carnot uses the Cyc knowledge base as both meta model and unifying schema, using knowledge based technology to integrate member databases [Collet et al. 1991].

1.2. Cyrano, a Proposed Meta Model

Cyrano is a proposed meta model designed for use in this research. Cyrano is based on an analytic approach to object orientation that combines the strengths of object oriented and rule based data models. Chapter 3 describes Cyrano in detail.

Cyrano's analytic approach recognizes that people view the world differently depending on whether they are engaged in planning or analysis. During planning, people determine what classes of object are needed and proceed to find or create objects of that class. During analysis, people examine objects, determine their attributes, and classify the objects based on those attributes.

For example, when planning to hang a picture, one needs a hammer: having determined this, one looks in a toolbox or hardware store to get a hammer. But when analyzing an object, one might note that it has a wooden handle and a clawed metal head with a striking surface: only after this initial examination can one conclude that it belongs to the class of hammers.

Traditional programming is a planning task. Thus, traditional object oriented systems use classes as templates for the creation of new objects: if one needs a hammer object, one instantiates class hammer.

In contrast, retrieving data from heterogeneous databases is an analysis task, not a planning task. Therefore, traditional object oriented approaches are not suitable: classes should classify existing objects, not be merely templates for new objects.

As an analytic model, Cyrano classes group similar objects, classifying them according to a set of defining rules that describe the objects that are members of the class. All objects that satisfy the rules are members of the class.

This combines aspects of object oriented and rule based data models. Like object oriented models, Cyrano organizes data into complex objects that can themselves be composed from other complex objects. This allows the greatest range of data structuring available in existing data models. Like rule based models, Cyrano uses rules to indicate how some structures are deduced from other structures. This allows powerful representations of the dynamic relationships between data classes.

Cyrano allows three types of classes, based on how the class derives the existence of its member objects.

1) A Gateway Class is the translation of a member database structure (eg, relation or record type) into the Cyrano model. In a sense, a gateway class is the class of all objects for which representations are stored in the member database structure. Thus, the class serves as a gateway through which the contents of the member data structure is viewed.

2) A Derived Class contains rules that describe its members. Derived classes are the mechanisms by which Cyrano resolves database heterogeneity. For example, one database might treat first and last names as separate data items, while another combines them into a single data item. A Cyrano derived class can contain all names that appear in either format in either of the databases. Further, the Cyrano derivation rules can

normalize the format so that the derived object either joins names or not, whichever is desired by the user.

3) A Built-In Class is one that is built into the implementation. Examples of built-in classes are Boolean and Integer.

1.3. Criteria to Evaluate Cyrano's Suitability as a Meta Model

An examination of Cyrano's suitability as a meta model requires a set of criteria for evaluating meta models. Chapter 4 presents three such criteria proposed as part of this research:

1) A meta model should fully support its member models.

2) A meta model should support database integration.

3) It must be possible to implement the meta model as part of an FDBS.

The first of these states that the meta model must fully support its member models. This means that any information that can be retrieved directly from the member database can also be retrieved using the FDBS. This has two dimensions. First, the meta model must be capable of representing all database schemata that can be generated using the member model. If it does not, then there will be some member database that could not be accessed via the meta model. Second, the query language of the meta model must be as powerful as the query language of the member model. If it is not, then some queries against the member database will produce information that is not available through the meta model.

Previous research into FDBS's did not address the question of full query support. The need for such support becomes clear when studying the inclusion of rule based databases into the federation. Rule based databases are distinguished primarily by their query languages. This sets them apart from other

models, which are distinguished by the way that they organize data. The inclusion of rule based databases makes it clear that querying capabilities must be addressed.

The dissertation outlines a method for determining whether a meta model fully supports a member model. Chapter 5 uses this method to develop proofs that Cyrano fully supports relational, object oriented, and rule based data models as member models.

The second criteria states that a meta model must contain mechanisms for integrating member databases. The purpose of an FDBS is to provide the user with a uniform view of a set of databases that overcomes the heterogeneity inherent in those databases. This means that the FDBS must support the integration of the data in those databases as one of its data manipulations.

The purpose of a data model is to define the data manipulations provided by a DBMS. The meta model is the data model of the FDBS. Therefore, the meta model must support the integration of data, that being one of an FDBS's primary data manipulations.

Chapter 6 demonstrates that Cyrano provides sufficient features for integrating databases.

The third criteria states that it must be possible to implement a meta model as part of an FDBS. The best way of demonstrating this is through development of a proof-of-concept FDBS implementing the meta model.

Chapter 7 describes Roxanne, a proof-of-concept FDBS that uses Cyrano as its meta model.

Chapter 2: Existing Research.

Over the past few years, heterogeneous databases have become a thriving research area. Major computer science periodicals have devoted special issues to this area [ACM 1990; IEEE 1991]. Books discussing database heterogeneity for MIS managers have appeared [Hackathorn 1993]. Major software houses have shipped products to address the problem (e.g., Lotus Datalens, discussed in [Chorafas and Steinmann 1993]).

This chapter discusses some of the existing research into heterogeneous databases. This discussion identifies the areas of interest to this dissertation, identifying the research gaps filled by the present research. In particular, this chapter discusses in detail three areas of heterogeneous database research:

1) Database heterogeneity. Database heterogeneity is the source of the problems addressed by this research.

2) Heterogeneous database systems, with an emphasis on federated database systems. These are a family of solutions to database heterogeneity.

3) Meta models, or unifying data models of a federated database system. A meta model is a central element of many heterogeneous database systems.

2.1. Database Heterogeneity

Database heterogeneity includes all of the ways that databases represent similar information in different ways. As described in this section, database heterogeneity can be broken into three categories: model, value, and structural heterogeneity.

2.1.1. Model Heterogeneity

Data model heterogeneity includes those database differences that arise from the use of different data models. A data model is a method of structuring and accessing data. Each model has three elements:

1) Basic data types provide a way to represent atomic data items. A basic type includes a set of allowable values and operations that act on these values. Some example data types are string, integer, and boolean.

2) Structuring mechanisms allow the combination of simple data items into more complex structures. Some sample structure mechanisms are records, tables, and objects.

3) Query languages allow the retrieval of data from the database. Some query languages are the Structured Query Language (SQL) and the Datalog language.

Table 2.1 lists several common families of data models, all of which are discussed by Ullman [1989]. It describes how each family provides the elements of a data model. As seen in that table, data models differ in their approaches to all three elements.

Different data models support different basic data types. This may result from a different underlying hardware leading to type differences (as, for example, might cause a difference between a 16 bit integer and a 32 bit integer). It may result from different extensions supported by different databases (such as those models that support a base type for representing dates.)

Some type differences result from differences in data model approach to type. At one extreme lies the rule based Datalog model, which does not have atomic types but instead groups together amorphous data elements. At the other extreme lie object oriented databases, in which all simple and complex data items

Table 2.1: Data Models

Model	Basic Types	Structures	Query Language
Network and Hierarchical	As provided in native programming language.	Linked records. Network links form directed graph. Hierarchical links form directed trees.	Graph traversal functions integrated into programming language.
Relational	Typically include string, integer, character, date, etc.	Tables contain flat rows or tuples. No explicit links between rows.	Based on relational algebra or calculus. Queries always terminate, but have computational limitations.
Semantic	As with relational.	Relational tables expanded to allow unique identity for data elements. Elements can contain sets and complex structures as attributes. Explicit links between structures provided.	Typically extensions of SQL.
Object Oriented	Basic types provided. User can extend with new types.	Complex objects similar to semantic data items. Classes can inherit from other classes. Operations can be encapsulated within classes.	Either extended SQL or object oriented programming language.
Rule Based	No true basic types. Atomic data items are of amorphous type.	As with relational. Tables are derived from other tables based on rules.	Rules specify data that user wants. Rules are typically Horn clauses.

have a type (or class) and users can extend the model with their own complex types. Most data models lie somewhere between these extremes, supporting a basic set of atomic types that cannot be extended by the user.

Different structuring mechanisms result in radically different approaches to organizing data in different databases. Indeed, different data models are primarily distinguished by how they organize data.

The relational models provide the simplest approach to structuring data. In a relational database, atomic data items are organized into tables, with each column in the table having an assigned semantic and each row containing linked data items. (Tables are sometimes referred to as relations, with rows referred to as tuples.) There are no explicit links across tables: links between rows are represented by having shared key values. For example, we might know that two rows refer to the same person by the fact that they have the same SSN value.

Rule based databases use the same type of data structures as relational databases. However, they use two types of predicates to access relations, extensional and intensional predicates. Extensional predicates provide access to stored data, like a relational table. Intensional predicates are linked to rules that, when run, provide the data tuples that satisfy the predicate. Thus, intensional predicates are dynamic, based on running rules, instead of static, as are extensional predicates and relational tables.

Network and hierarchical models structure data into records. Records have an established type that specifies the data types and semantics of the record elements. Records can have explicit links to other records. In the network model, the graph of record links can be any directed graph (though some network models may require that no record have a link to itself). In the hierarchical model, the graph of links forms a set of trees, with each tree containing links from its root records through intermediate records to its leaves.

Object oriented models have the most sophisticated data structuring mechanisms. Object oriented models structure atomic data elements as the attributes of complex objects. Each object is a member of a class, which specifies the semantics and types of its attributes. A class specifies the operations that can be done on objects that are members of the class: the object, upon receiving the appropriate request, performs the operation specified in the class definition.

Objects can be linked in several ways. An object can have as an attribute another complex object. This can be an ownership relation in which the object owns its attribute object (as might happen if the attribute is part of the primary object) or it can be a link from one object to another in which there is no ownership (as might happen if the primary object only refers to the attribute object). Further, most object oriented systems allow attributes to be sets of objects or object links.

Classes themselves can be linked. A class can be a subclass of another class. When this occurs, the child class inherits the attributes and operations of its parent class. This allows object oriented databases to represent complex relationships between data types, representing varying degrees of generalization. Some object oriented systems allow multiple inheritance, in which a class can be a child to multiple unrelated parent classes. In this case, the child inherits from all of its parents.

Most data models use their structure mechanisms to group data objects. For example, a relational table specifies the structure of its rows as well as providing a central access point for them. In contrast, object oriented databases provide explicit container objects that group other objects. Multiple containers can hold different sets of objects of the same class.

Semantic data models are an evolutionary step between relational and object oriented models. Semantic models typically allow complex nested structures and unique identities for data items, as do object oriented models. However, they do not support inheritance, and they usually do not allow operations

to be associated with specific structures.

Different query languages result in differences in the information that a user can extract from a database.

In the oldest systems, network and hierarchical databases, there are no true query languages. Instead, these databases provide calls for integration with a programming language. These calls allow the application to navigate through the linked structures.

Relational databases typically support query languages derived from relational algebra, predicate calculus, or both. The most common of these is the standard Structured Query Language (SQL), which contains elements of both algebra and calculus. These languages have a solid mathematical foundation and guarantee that queries will safely terminate. However, they have limitations resulting from their lack of ability to loop or recurse through related tuples. For example, an SQL database cannot calculate the transitive closure of a relation (e.g., it could not find a list of all of a person's ancestors given a database listing parent-child relationships).

Most semantic databases support extensions of relational languages. This is most commonly SQL, because of its prevalence. The extensions provide access to the more complex data structures supported by semantic databases. They typically share both the strengths and weaknesses of the relational languages.

Object oriented databases fall into two categories. Like semantic databases, many object oriented databases support extensions to SQL. Many of these support looping or recursion, allowing for broader queries then those allowed by SQL. Other object oriented databases use a full programming language with support for simple queries. These typically evolved from object oriented programming systems to allow persistent storage of program objects.

Rule based databases are primarily distinguished by their query languages.

These allow users to specify queries as sets of rules. The data that satisfies the rules satisfies the queries. These languages may have provisions to make queries be safe (i.e., guaranteed to terminate). However, in an attempt to allow for the broadest range of queries, rule based databases may allow unsafe queries.

2.1.2. Data Heterogeneity

Data heterogeneity encompasses the ways in which data values can differ across databases. As derived from Kim and Seo [1991], table 2.2 lists the types of data heterogeneity. They are grouped into value and representational inconsistencies.

Value inconsistencies arise when databases contain different attribute values for the same data element. This can result from missing, obsolete, or incorrect data in one or more databases. For example, a person's address may not have been updated in one database, while in another it was incorrectly entered when updated. These inconsistencies can result from different databases having different default values for similar data elements.

Representational inconsistencies arise when databases contain values with the same semantics but different representation. There are several types of representational inconsistency, including differences in type, units, precision, and representation.

Data type differences arise when databases use different basic data types to represent data. For example, one database may use a 5 character field to hold zip codes, while another uses an integer. A subtle form of data type difference results when data items come from different databases that support similar but slightly different atomic types. For example, one database may use a 16 bit integer where another uses a 32 bit integer for a similar data value.

Table 2.2: Data Heterogeneity

Value Inconsistencies
Differences in data values between databases.

Missing Data
One database is missing data that another contains.

Obsolete Data
One database contains obsolete data.

Incorrect Data
One database contains incorrect data.

Representational Inconsistencies
Differences in representation of similar data.

Data Types Differences
Use of different data types to represent similar data.

Unit Differences
Use of different units for similar data.

Precision Differences
Different precision used for similar data.

Representation Differences
Different enumerated representations of similar data.

Differences in units arise when databases use different units for similar data elements. For example, one database may represent temperature in degrees fahrenheit, while another uses degrees kelvin.

Differences in precision arise when one database maintains a data value with more precision then another. For example, one database may represent irrational numbers such as pi in 100 digits, while another represents them in 1000 digits.

Differences in representation arise when databases use different enumerated values or measurement scales to represent the same data. For example, one database may store grades on an A-E scale, while another scores them on the 4.0-0.0 scale.

2.1.3. Structural Heterogeneity

Structural heterogeneity includes the ways in which different databases can use different schemata to represent similar information. While data heterogeneity contains the differences in representation and value of a single piece of data, structural heterogeneity consists of the possible differences in the organization of multiple pieces of data. As derived from Kim and Seo [1991], table 2.3 lists the types of structural heterogeneity.

Databases typically contain structures (relations, records, objects, etc.) that represent real-world entities. Structures contain attributes that describe some aspect of the structure. The database schema is the blueprint of the structures in the database.

Structural heterogeneity arises when different databases use different structures (and thus different schemata) to represent similar entities. This includes differences in naming, constraints, and data grouping. A fourth type, inheritance differences, applies only to object oriented databases.

Table 2.3: Structural Heterogeneity

Naming Differences
Different names used for similar data or same name used for different data.

Structure Name Differences
Differences in structure names.

Attribute Name Differences
Differences in attribute names.

Constraint Differences
Different constraints lead to different data domains.

Data Grouping Differences
Differences in grouping of data.

Missing Attribute Differences
A structure does not have all the attributes of a similar structure.

Multiple Structure Differences
Differences in the number of structures used to represent similar information.

Multiple Attribute Differences
Differences in the number of attributes used to represent similar information.

Structure vs Attribute Differences
A structure is used to represent information represented by attributes in another database.

Inheritance Differences
Differences in the class-subclass hierarchy of object oriented databases.

Naming differences arise when different names are used for similar entities or when the same names are used for different entities. Different databases can use different names for structures that represent the same entities (e.g., a relation called "CAR" versus a relation called "AUTOMOBILE"). Different databases can have different names for the same attributes of a structure (e.g., an attribute called "ADDRESS" versus an attribute called "HOME-ADDRESS"). Different databases can use the same name for structures or attributes that have different semantics (e.g., two attribute called "REVENUE", one that means gross revenue and one that means net revenue).

Constraint differences arise when different databases have different constraints apply to similar structures. Constraints limit the domain of a data element. When different databases have different constraints, their data values have different domains. As a result, one database's value domain can be a superset of another's, or domains can partially overlap. For example, one database may have a constraint that graduates must have no pending library fines while another database has no such restriction.

Data grouping differences arise when similar structures are organized in different ways. These include missing attributes, multiple structure differences, multiple attribute differences, and structure versus attribute differences.

A missing attribute difference arises when a structure in one database lacks an attribute found in a corresponding structure in another database. A missing attribute might have an implicit value in a particular database. For example, a database of army officers has no explicit service branch attribute: all personnel listed in the database are in the army.

A multiple structure difference arises when two databases use different arrangements of structures to represent the same data. Multiple structure

differences can be fairly complex, involving several incompatible structures used by different databases to represent the same information. For example, one database might maintain separate structures for clerical and production employees, while another uses structures for managers and workers. Managers from the second database would be stored as either clerical or production employees in the first database depending on whom they manage.

Multiple attribute differences arise when two databases use different numbers of attributes for the same information. For example, one database may use separate attributes for first, last, and middle names, while another database uses a single attribute containing the entire name.

Structure versus attribute differences arise when one database uses attributes to represent data that another database represents as separate structures. For example, one database might maintain a person's employer as an attribute while another has a separate structure for employers.

The complexity of the object oriented model results in inheritance differences. Inheritance differences arise when one database uses a different class-subclass hierarchy than another. This may produce other types of structural heterogeneity. For example, one database might have a manager class which is a direct subclass to a person class, while another might have the manager class be a subclass to an employee class which is itself a subclass of the person class. If the employee class has some constraint that does not apply directly to the manager and person classes, then inheritance of the constraint results in a constraint difference between the two manager classes.

2.2. Federated Database Systems

This section describes heterogeneous and federated database systems as solutions to database heterogeneity. A heterogeneous database system is any

system that integrates heterogeneous databases. A federated database system (FDBS) is a heterogeneous database system in which the individual databases are autonomous. This section presents two ways of classifying heterogeneous database systems, introduces a reference architecture for use in discussing FDBS's, and discusses existing FDBS's and other research into FDBS issues.

2.2.1. Classifications of Heterogeneous Database Systems

To understand the differences between heterogeneous database systems, one must first identify the classes of these solutions. This section describes two ways of classifying heterogeneous database systems. The first classifies them by their system architectures. The second classifies them by the ways in which they handle diverse heterogeneous databases.

There are several architectures in use to address the heterogeneous database problem. Hsiao and Kamel [1989] provide a categorization of these architectures. This categorization distinguishes between source systems, which generate transactions, and target systems, which process them. In a typical application, a user at a source system queries for data in a target system, requiring the translation of the query transaction from source to target formats. Hsiao and Kamel distinguish architectures by how many source and target systems they support. Figure 2.1 shows these architectures.

A 1:1 system supports one source system and one target system. This may be static, done to initialize the source system with data from the target system. 1:1 systems are appropriate when adding a single database as an external data store for an existing source database. Static translation is appropriate when translating existing data to a new database management system. Research using this approach is described by Katz and Wong [1982] and the SDDTG CODASYL committee [1977].

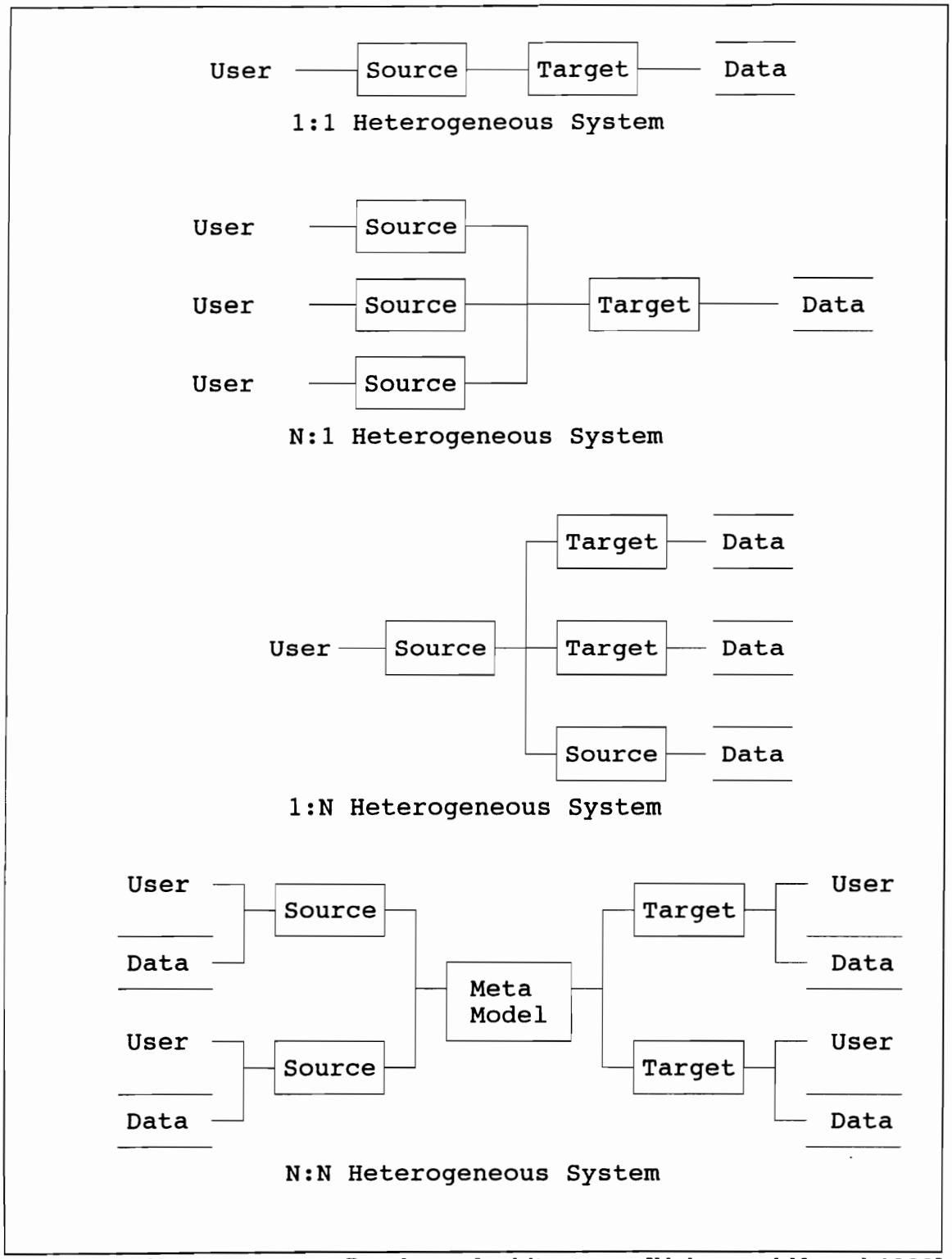

Figure 2.1: Heterogeneous Database Architectures [Hsiao and Kamel 1989]

In an N:1 system, a single target system processes transactions from several sources. This is analogous to the way that a single assembly language underlies several high level languages (and like an assembly language, the target data model tends to be at a lower level then the source models). This is useful when different data models are appropriate for applications sharing a data store. The target system provides data storage and transaction management, allowing implementers to concentrate on the source model. The Multilingual Database System (MLDS) is an example of such a system [Demurjian and Hsiao 1988].

In a 1:N system, a source system provides uniform access to data stored in several target systems. This is appropriate when a user or application group needs access to multiple databases. This approach requires a meta model. All user interaction with the system uses the meta model. This dissertation adopts this architecture to address the heterogeneous database problem. Example 1:N systems are described in section 2.2.3.

N:N systems allow users of different DBMS's to access all data stored in all databases. This approach requires a meta model. Transactions between databases require translation from the source model to the meta model to the target model. This is appropriate when pre-existing databases and applications need to share data. The HD-DBMS is a sample N:N system [Cardenas 1987; Cardenas and Pirahesh 1980].

1:N and N:N systems are often called multi-databases. When the target systems have a high degree of autonomy, a multi-database is called a federated database system (FDBS).

Multi-databases can be classified according to how they resolve database heterogeneity. Sheth and Larson [1990] provide such a classification, shown in figure 2.2.

The Sheth and Larson taxonomy first classifies multi-database systems

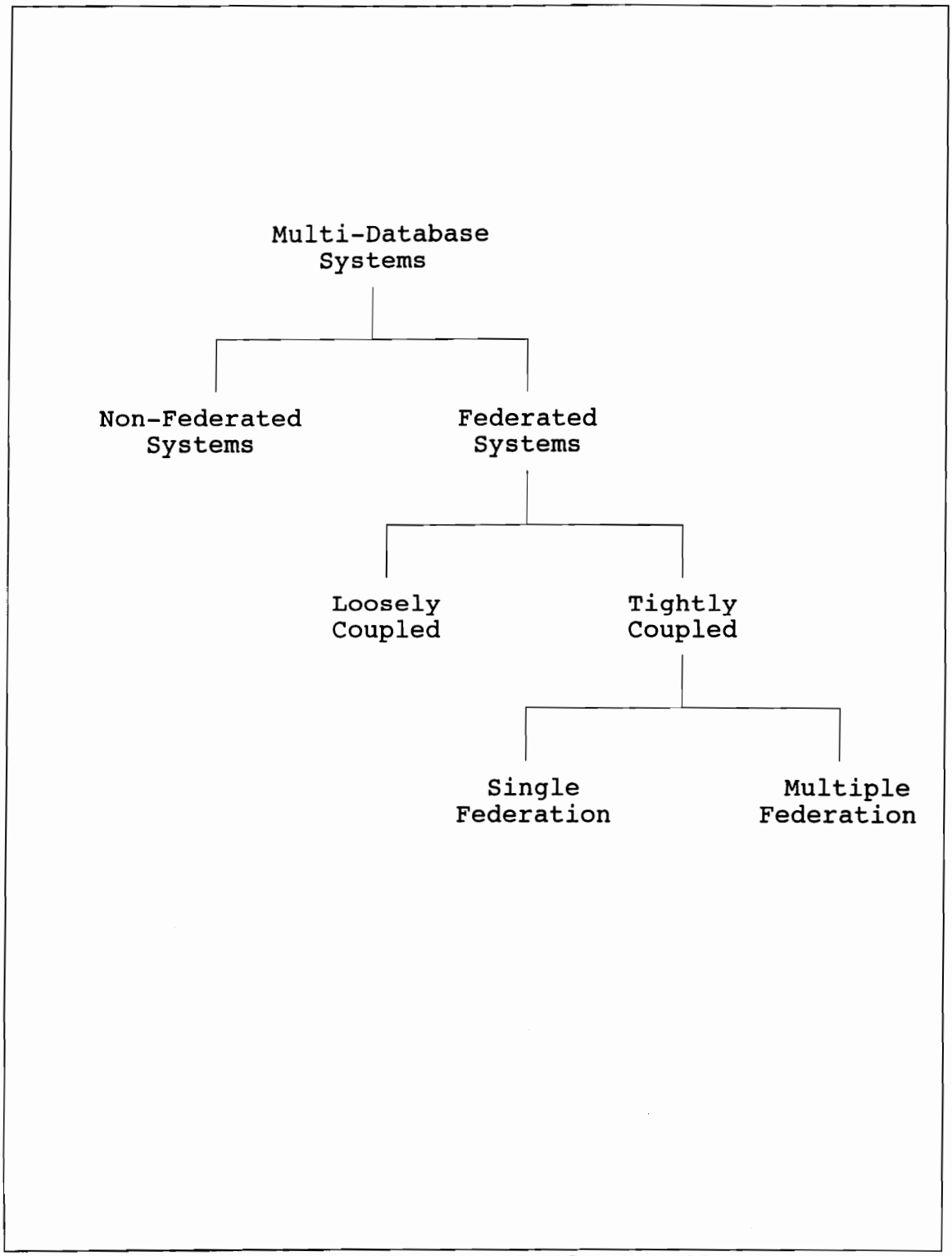

Figure 2.2: Multi-Database Taxonomy [Sheth and Larson 1990]

according to the level of autonomy of the member databases. In a non-federated database system, the central system maintains complete control of member databases, including control over data security and user authentication. In a federated database system (FDBS), member databases are autonomous and maintain independent control over data access. Because the FDBS user does not need to have administrative control over the member database, an FDBS is appropriate when member databases already exist with their own access procedures.

FDBS's are further classified according to whether the user or the system integrates the data from the various databases. In a loosely coupled system, the user directly accesses the translated member databases and performs his own data integration. In a tightly coupled system the FDBS administrator uses system tools to create and maintain a global database that integrates data from member databases.

Tightly coupled systems are further classified according to how many global databases they allow. In a single federation system, there is a single global database for all users. In a multiple federation system, several global databases support different users. Both types of system can have user views that further refine the global database to support specific user groups.

2.2.2. A Reference Architecture for Federated Database Systems

A reference architecture provides a framework for the design and study of federated database systems. Several authors have presented such reference architectures. However, these suffer from their separation of database operations into data structure and data processing. To address the shortcomings inherent to that approach, this section presents an object oriented reference architecture that combines data structure and processing. An advantage of such a reference architecture is that it can highlight the important issues in resolving database

heterogeneity.

Several authors have presented reference architectures for use in discussion of FDBS's [Sheth and Larson 1990; Deen and Amin 1985; Litwin et al. 1990; Dwyer and Larson 1987]. These typically include a schema architecture, which describes the schema layers necessary to support the translations done by the FDBS, and a process architecture, which describes the types of translations performed on the data to support the various transactions. This follows the traditional software design approach of separating data from function (or, in this case, schema from process).

For example, Sheth and Larson's reference architecture contains a schema architecture and building blocks for forming a processor architecture [Sheth and Larson 1990]. The schema architecture shows the schema layers, giving a blueprint of how data is organized across elements of the federation. The processor architecture identifies the processing elements that translate between the schema layers.

The problem with this approach is that it isolates the data structures from the data processing. In broad terms, the processing of data by different databases tends to be similar: most provide mechanisms for data querying, data update, and schema modification. Data models differ in these areas only at the detailed level, such as in the specific capabilities of their query languages. In contrast, data models have dramatic differences in the ways that they structure data, such as the differences between flat relations and complex objects. Thus, when using traditional FDBS architectures to study the effect of database heterogeneity, the tendency is to concentrate on the schema architecture over the process architecture.

This can lead one to overlook the differences in the transactions supported by the different databases. But these differences, while often subtle, are important: users are ultimately more concerned with what they can do with data

then how it is structured. Therefore, what is needed is an architecture that emphasizes differences in transactions over differences in structure.

Figure 2.3 shows a proposed FDBS reference architecture that provides such an emphasis.

The building blocks of the architecture are databases. Each database is an object in the object oriented design sense: it maintains an internal state (the contents of the database and/or its schema), may alter its state (to update the schema or stored data), and returns specific results in response to certain messages (its supported transactions). The schema is internal to the database and, together with the data model, determines the domain of acceptable transactions.

The reference architecture identifies the layers of databases that together make up an FDBS. In order to process its transactions, a database may issue transactions to databases from a lower layer. The types of databases at the different layers are member, gateway, global, and user databases.

A *member database* is independent of the FDBS. It may be built under any data model supported by the FDBS. It accepts transactions in the language of its native data model and returns results in its native format.

A *gateway database* is the translation of a member database into the meta model of the FDBS. Thus, it serves as a gateway through which higher FDBS layers can access the member database. Each gateway database corresponds to a single member database. The gateway database accepts transactions in the meta model's native form, translates them into the language of the member database, receives the results, translates them into meta model format, and returns them to the originator of the transaction.

A *global database* combines one or more gateway databases. It accepts transactions in the meta model's native form, breaks them into sub-transactions

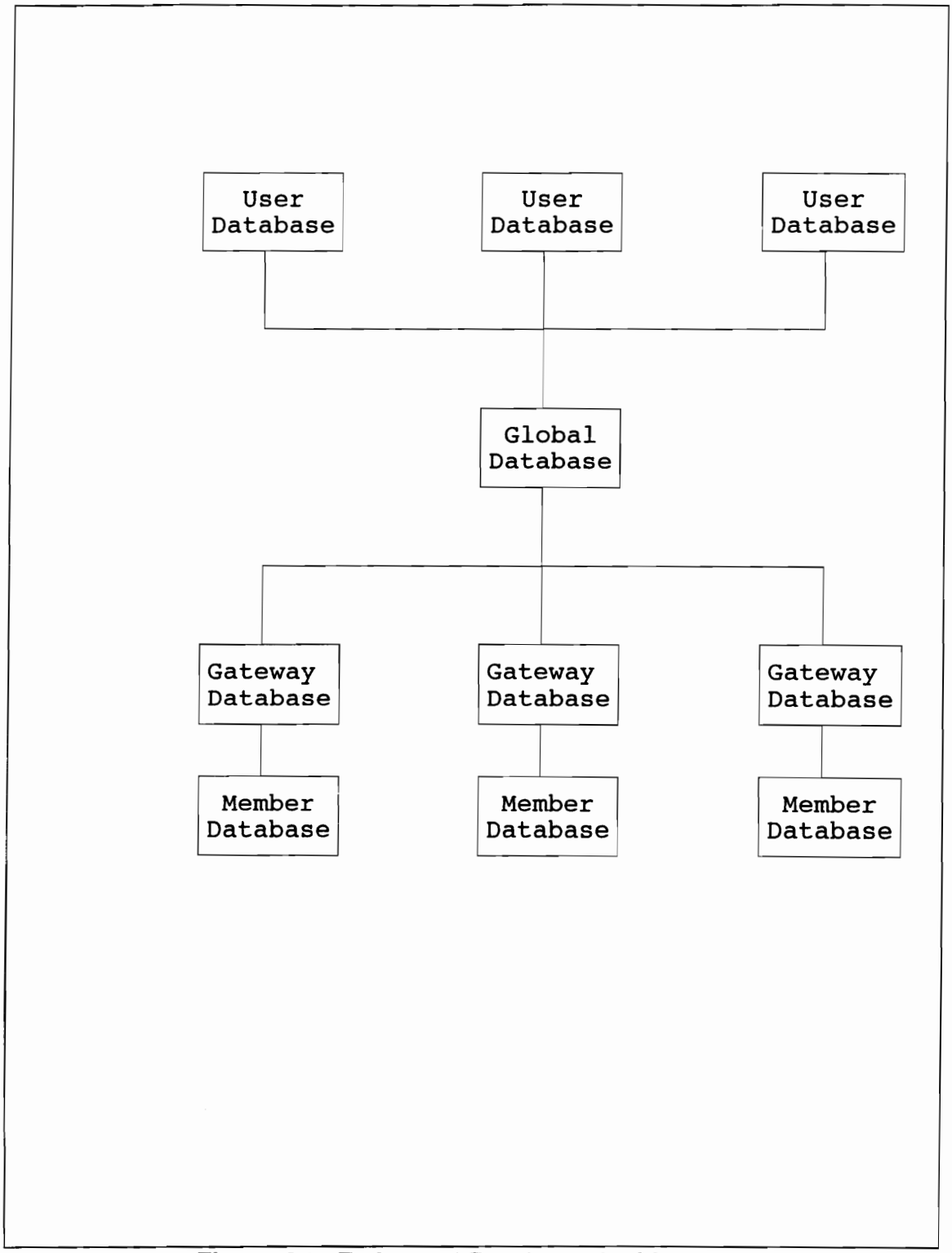

Figure 2.3: Federated Database Architecture

each of which runs against a single gateway database, submits the sub-transactions to the gateway databases, and merges the results to form the result of the original transaction. These actions can be guided by rules generated either by a database administrator or automatically.

A *user database* is a user view of the global database. It provides a translation of the global database, either to limit a user's access to a subset of the global database or to present the data in a manner more appropriate to the user's applications and needs. It is similar to a view in a relational database.

A given FDBS may use some or all of these architectural layers. In particular, several FDBS's do not provide the capability of integrating gateway databases into a global database. Such FDBS's do not provide global or user databases. Instead, they provide the user with a collection of gateway databases to access.

By emphasizing the translation of transactions, this reference architecture concentrates more on the transaction processing capabilities of different databases then on the way data is organized. In contrast, by concentrating on schemas, earlier reference architectures emphasize data structuring differences over transaction processing. Because queries are database transactions, the proposed architecture is more appropriate then previous architectures in addressing the differing query capabilities of different databases.

The reference architecture highlights the critical issues in federated database design. The central problem for a federated database is the translation of data between different representations. In the federated architecture, this is represented by the transitions between database layers. These transitions represent the processing done by the FDBS to process different types of heterogeneity.

In the translation from member to gateway database, the FDBS overcomes data model heterogeneity. Data model heterogeneity arises when the member databases are built on different data models. In the translation to the gateway database, the FDBS normalizes all data on the FDBS's meta model, thus overcoming data model heterogeneity. Therefore, this is the first major translation performed by the FDBS.

Once translated into the meta model, value and structural heterogeneity still remain. The translation from gateway to global database overcomes this heterogeneity. The global database is the integration of the gateway databases, providing access to the data of the gateway databases with value and structural heterogeneity removed. Therefore, this is the second major translation performed by the FDBS.

The third transition, from global to user database, is similar to the translation from conceptual schema to user view in a relational database and is not unique to FDBS's. Because of this, this dissertation does not emphasize this translation.

2.2.3. Existing Research into Federated System Development

There has been much research into issues related to federated databases. Early work examined heterogeneous database design issues. Later, first generation FDBS's integrate relational, hierarchical, and network databases. Second generation systems add object oriented and rule based systems to the mix. Throughout this time, research appeared on specific issues in heterogeneous database system design.

In 1990, ACM Computing Surveys devoted a special issue to heterogeneous database systems. It includes surveys on federated system design and implementations [Sheth and Larson 1990; Litwin et al. 1990] and a survey of several existing federated systems [Thomas et al. 1990]. The issue provides a good starting point for research into federated systems.

Early FDBS work developed functional designs for federated systems [Adiba and Portal 1978]. This work recognized the fundamental issues of the FDBS approach, the normalization of member databases using the meta model and the integration of the gateway databases into a single global database. The work also includes a limitation common to later research: it concentrates on the design of federated system software and pays little attention to data modeling issues.

As shown in table 2.4, first generation FDBS's share several characteristics in common. Most support relational, hierarchical, and network databases as member databases. Most use a relational or semantic meta model. And most are tightly coupled.

Most of these systems support some combination of relational, network, hierarchical, and flat file databases. This reflects the databases that were common when these systems were designed. Of these systems, only the relational supports a true query language: thus these systems have query languages capable of representing relational queries.

Also reflecting the then-popular data models, most of these use a relational meta model. The exceptions use semantic or similar functional meta models, an evolutionary step beyond the relational models.

Most of these systems are tightly coupled, with either a single or multiple federation. The exception, Buneman's system, is a loosely coupled system that presents users with the collection of all schemes of the gateway databases. Users generate combined queries that explicitly reference the different gateway databases. The queries are in the language of the meta model; the system provides data model normalization but not integration of databases. This approach is useful when there are so many member databases that developing the schema for the global database is a prohibitively large job. This approach is discussed in [Buneman et al. 1990].

PRECI* is a hybrid of the tightly and loosely coupled approaches [Deen et

Table 2.4: First Generation Federated Database Systems

System	Status	Classification	Meta Model	Supported External Models	Reference
ADDS	In Production	Multiple Federation	Extended Relational	Relational, Hierarchical, Network	Breitbart et al. 1986
California Intelligent Database Assistant (CALIDA)	In Production	Loosely Coupled	Relational	Relational, Hierarchical, Flat File	Chorafas and Steinmann 1993
Dataplex	Prototype	Multiple Federation	Relational	Relational, Hierarchical	Chung 1990
DAVID	Prototype	Single Federation	Cluster (Extended Rel.)	Relational, Hierarchical, Network	Jacobs 1985
IMDAS	In Production	Single Federation	Semantic Assoc. Model	Relational, GBASE (object oriented), Flat File, Shared Memory	Thomas et al. 1990
Ingres-Star	In Production	Multiple Federation	Relational	Relational, Hierarchical, Flat File	Thomas et al. 1990
Mermaid	Under Development	Multiple Federation	Relational	Relational, Network, Flat File	Templeton et al. 1986
Buneman's System	Prototype	Loosely Coupled	Relational	Relational	Buneman et al. 1990
PRECI*	Prototype	Tightly/Loosely Coupled Hybrid	Relational	Relational	Deen et al. 1985
Multibase	Prototype	Multiple Federation	Daplex (a functional model)	Relational, Hierarchical, Network, Flat File	Chan and Ries 1982 Shipman 1981

al. 1985]. PRECI* allows a core group of member databases to be included in the global database, while other databases are accessed directly via their gateway databases. This provides data model normalization and partial integration. It also allows the easy addition of new member databases to the federation without requiring the effort of altering the global database to support them. This approach is ideal for an environment like the Internet, in which users may wish integrated access to a core set of databases without sacrificing ad hoc access to other diverse databases.

Table 2.5 shows second generation FDBS's. These fall into two categories: research oriented and commercial systems.

The research systems (Carnot, Pegasus, ViewSystem, CIS, FBASE, and UniSQL/M) differ from first generation systems in that they support object oriented member databases and do not support the now-aging network and hierarchical databases. They also use newer models as meta models. However, like their predecessors they tend to be tightly coupled federations.

The commercial systems (IBM's Distributed Relational Data Architecture (DRDA), Lotus DataLens, and the California Intelligent Database Assistant (CALIDA, the first implementation of GTE's Intelligent Database Assistant)) are more similar to the first generation FDBS's. Thus, they are the emergence of first generation technologies into the marketplace. The primary difference is that the commercial systems tend to be loosely coupled, thus avoiding the many open research issues in resolving value and structural heterogeneity.

In addition to papers on specific federated systems, there is a growing body of literature on specific issues common to several systems. These include heterogeneous update and database integration.

A heterogeneous update transaction is an update transaction submitted to

Table 2.5: Second Generation Federated Database Systems

System	Status	Classification	Meta Model	Supported External Models	Reference
Carnot	Prototype	Single Federation	Cyc (Rule Based)	Entity Relationship, Relational, Object Oriented	Collet et al. 1991
Pegasus	In Development	Multiple Federation	IRIS (Object Oriented)	Object Oriented, Relational	Ahmed et al. 1991
UniSQL/M	In Production	Single Federation	Object Oriented	Object Oriented, Relational	Kim and Seo 1991
ViewSystem	Prototype	Multiple Federation	VODAK (Object Oriented)	Relational, Object Oriented	Pitoura et al. 1995
CIS	Production	Multiple Federation	Object Oriented	Relational, Graphical	Pitoura et al. 1995
FBASE	In Development	Loosely Coupled	FSQL	Relational, Object Oriented	Pitoura et al. 1995
DRDA	Commercial	Single Federation	Relational	Relational	Chorafas and Steinmann 1993
Lotus DataLens	Commercial	Loosely Coupled	Relational	Relational, Hierarchical	Chorafas and Steinmann 1993
CALIDA	Commercial	Loosely Coupled	Relational	Relational, Hierarchical, Flat File	Chorafas and Steinmann 1993

the global or user database that propagates down and causes data to be added to the member databases. This requires the translation of updates down through the system. It also requires the synchronization of the updates across all member databases in order to ensure that updates are consistent. Because of the complexity of this task, most FDBS's do not support update in the initial development. Heterogeneous update is outside the scope of this dissertation. Some papers on heterogeneous update are [Breitbart et al. 1986; Dayal and Bernstein 1982; Gligor and Popescu-Zeletin 1986; Pu 1988; Soparkar et al. 1991].

Database integration is the integration of gateway databases to create a global database. The types of database heterogeneity described in section 2.1 complicate this task. While database integration is currently a hot research topic, it is beyond the scope of this work. Different integration approaches are described by Batini et al. [1986], Dayal and Hwang [1984], Effelsberg and Mannino [1984], and Larson et al. [1989]. More papers can be found in a special issue of ACM's SIGMOD RECORD devoted to semantic heterogeneity [1991].

There have been several approaches proposed for providing system tools for dealing with heterogeneity. Buneman et al. [1990] present an approach in which users resolve heterogeneity in their queries in a loosely coupled federated system. Motro [1987] presents an algebra for the construction of "superviews," or integrated schemas. Deen et al. [1987] presents an approach used by PRECI* in which algebraic tools allow the integration of translated gateway database schema elements.

2.3. Meta Models

Although several meta models have been used in FDBS's, none has emerged as the definitive choice for integration of relational, object oriented, and rule based databases. This section identifies several features of a meta model that can make it capable of integrating the three target models. Based on those

criteria, the chapter examines meta models used in existing FDBS's. Table 2.6 lists these models and identifies which of the features they support.

Several meta model features help to integrate relational, object oriented, and rule based databases. The first of these shows whether a translation exists from the meta model to the relational model. The next two relate to support for object oriented data models, specifying whether the meta model can represent complex objects (i.e., object aggregation) and inheritance. The following two relate to features of rule-based databases, indicating support for deductive structures (i.e., structures with values derived from values of other structures) and recursive queries. The final aspect is the support of database integration as a feature of the meta model's language.

Of these criteria, all are required to fully support relational, object oriented, and rule based member databases with the exception of support for deductive structures. However, deductive structures allow a more graceful representation of rule based database elements.

Most FDBS's use an established data model as the meta model. Frequently, the FDBS designers modify the data model to provide additional support for data structures that are not normally provided by the meta model. For example, a modified relational model typically allows nested relations that can represent the record links of a network model.

When the first wave of FDBS's were designed, the relational model was the state-of-the-art data model. Therefore, most first generation FDBS's use a modified relational model as the meta model (e.g., ADDS, Dataplex, Ingres-Star, Mermaid [Thomas et al. 1990]). A modified relational model is adequate for supporting relational, hierarchical, and network models as member models. However, it does not support such complex data structuring mechanisms of object

Table 2.6: Meta Model Support of FDBS Features

	Extended Relational	Semantic	Database Logic	Extended Database Logic	Rybinski	Pegasus	Carnot	Cyrano
Relational Support	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Complex Objects	No	Yes	No	Yes	No	Yes	Yes	Yes
Inheritance	No	No	No	No	No	Yes	Yes	Yes
Deductive Structures	No	No	No	Yes	No	No	Yes	Yes
Recursive Queries	No	No	No	No	No	Yes	Yes	Yes
Language Based Database Integration	No	No	Yes	Yes	No	Yes	Yes	Yes

oriented databases as inheritance and complex objects. Neither does it provide support for recursive queries as required to support rule based databases. Thus, modified relational models do not provide full support for the newer data models.

Some first generation FDBS's use semantic or functional models as meta models (eg, IMDAS, Multibase [Thomas et al. 1990]). These provide some support for object oriented constructs but fall short of full support for inheritance and recursive queries.

Several first generation systems use meta models designed specifically for federated use. The first of these, Database Logic [Jacobs 1982; Jacobs 1985], serves as a logical foundation for heterogeneous databases in the way that first order logic serves as a foundation for relational databases. Like a relational database, the schema and constraints of a Database Logic database serve as a logical language and a theory in the language. The contents of the database form a model of the theory.

Database Logic differs from the relational model in that it is based on second instead of first order logic. Instead of relations, Database Logic uses clusters, or second order predicates, to structure data. Being second order, clusters can contain other clusters, violating the relational first normal form and allowing the representation of the linked records of hierarchical and network databases.

Database Logic provides no support for object oriented constructs such as inheritance. Further, with its query languages based on relational languages, Database Logic does not support recursion and thus is less powerful than rule based systems. Therefore, Database Logic gateway databases do not fully support object oriented or rule based member models.

Extended Database Logic [Grant and Sellis 1990] extends Database Logic with support of some object oriented and rule based constructs. Extended data structuring allows complex data elements to be treated as atomic data items, thus

supporting data aggregation and complex objects. Clusters can be derived based on sets of rules in the style of intensional predicates of rule based systems, allowing some capabilities of deductive databases. However, Extended Database Logic supports neither inheritance nor recursive queries and thus falls short of fully supporting object oriented and rule based databases. It does, however, provide some support for data integration through the use of derived clusters.

Responding to perceived weaknesses in the cluster model, Rybinski [1987] proposed a semantic meta model capable of supporting relational, network, and hierarchical databases. Rybinski's model is based on first order logic, thus ensuring that compactness and completeness hold for the model. (Database Logic is based on second order logic in which compactness and completeness do not hold.) However, like Database Logic, Rybinski does not provide mechanisms to support either object oriented data modeling or recursive rule based queries.

The Multimodel Multidatabase System (M(DM)) is a meta model based on second order logic [Gangopadhyay and Barsalou 1991]. It provides mechanisms for defining data element types found in local databases. However, it does not support inheritance and lacks a query language; it is therefore both incomplete and incapable of supporting object oriented databases.

Designed when object oriented and rule based models were achieving prominence, the second generation FDBS's generally use these types of meta models. For example, Pegasus uses an object oriented meta model [Ahmed et al. 1991], UniSQL/M uses a hybrid object-oriented/relational model [Kim and Seo 1991], and Carnot uses a knowledge base as both meta model and global database [Collet et al. 1991]. Pitoura et al. [1995] present a good survey of these and other systems that use an object oriented meta model.

An object oriented model is suitable as a meta model because of the breadth of its data modelling capabilities. Of the models studied, object oriented

models have the richest sets of data modelling features. These make it relatively easy to represent various member structures as objects without needing to reduce the complexity of the representation. For example, a relation can be viewed as an object with atomic attributes and no class-subclass relationships. In contrast, when representing a complex object in a relational meta model, the object must be reduced to several interlocking flat relations, thus sacrificing the unity of the original representation.

Pegasus provides a good example of an object oriented meta model [Ahmed et al. 1991]. It is based on the Iris object oriented data model and contains the standard data modelling elements of an object oriented model: a class hierarchy, complex objects with unique object identities, and encapsulated functions. Pegasus uses the Heterogeneous Object Structured Query Language (HOSQL) for its data and query language. HOSQL is a version of SQL modified for object oriented use in a heterogeneous database environment.

Pegasus satisfies the requirement of full support for its member databases by providing a rich set of features for object oriented data modeling and a data and query language that is a superset of SQL. Pegasus's target member data models are the relational and Iris object oriented models. The object oriented model is sufficient for representing relations, and HOSQL, an extension of OSQL (itself an extension of SQL) subsumes SQL and thus provides full support for relational queries. Pegasus's meta model is an extension of Iris (and OSQL, the basis of HOSQL, is the query language for Iris): therefore, Pegasus clearly provides full support for Iris.

Pegasus satisfies the requirement of support of database integration though several mechanisms provided by HOSQL. A Pegasus administrator can create abstract superclasses as superclasses of existing classes. The abstract superclass federates its subclasses, which can be from different gateway databases. HOSQL also supports name aliases for resolving naming differences

and derived functions that resolve differences between different gateway classes and functions.

A rule based meta model has the advantage of supporting structures with values derived from other structures. Such a mechanism is particularly appropriate for integrating heterogeneous databases. Intensional predicates can have multiple derivation rules, each of which accesses the data from one gateway data structure. These predicates integrate the gateway structures in a manner orthogonal to normal rule based data manipulations.

Carnot uses Cyc, a knowledge base, as both meta model and global database [Collet et al. 1991]. Cyc is a knowledge base containing 50,000 entities and relationships. It provides a full set of knowledge representation and manipulation features. Carnot integrates member data into the Cyc knowledge base through use of articulation axioms.

Carnot supports relational, entity-relationship, and object oriented data models as member models. It provides full support for these by application of its rich set of knowledge base structures. Collet et al. [1991] discusses both those structures (which have an object oriented flavor) and how they represent the member data structures.

Carnot's articulation axioms provides support for database integration. Carnot first translates the member databases into Cyc representations and then uses articulation axioms to guide the integration of the member data into Cyc. Articulation axioms are logical rules that indicate how to extract knowledge from the gateway databases and integrate it into the global knowledge base.

Although the second generation meta models support most of the features required to integrate the target member models, there is no clear winner among them. This dissertation systematically examines the features required for a meta

model to support the three target models. Chapter 4 presents the results of this examination, with a further analysis of the second generation meta models identified in this chapter.

Chapter 3: Cyrano: A Meta Model

Cyrano is a hybrid of object oriented and rule based data models suitable for use as a meta model. Cyrano was designed as part of this research to be capable of federating relational, object oriented, and rule based databases.

This chapter describes Cyrano in some detail. The chapter begins with a description of an analytic approach to object orientation that serves as the conceptual basis for Cyrano. It moves to a detailed view of Cyrano, complete with a sample Cyrano application. The chapter concludes with a discussion of how Cyrano relates to other efforts of combining object oriented and rule based models.

3.1. Analytic Object Orientation

This dissertation introduces analytic object orientation as the basis of the Cyrano meta model. Analytic object orientation is a style of object orientation that is particularly appropriate for use in meta models. Analytic object orientation recognizes that there is a difference between the world views suitable to planning and analysis. While traditional object orientation is appropriate for planning, querying heterogeneous databases requires analysis, suggesting that a meta model should differ from traditional object oriented models. Analytic object orientation contains such differences, incorporating the strengths of rule based data models into an object oriented model. Because of this, analytic object orientation has several practical differences from traditional object oriented models.

People view the world differently depending on whether they are engaged in planning or analysis. During planning, people determine what types of object are needed and find or create objects of those types. During analysis, people examine objects, identify their attributes, and classify the objects based on those attributes.

For example, when planning to hang a picture, one needs a hammer. Having determined this, one looks in a toolbox or hardware store to get a hammer. But when analyzing an object, one first notes that it has a wooden handle and a clawed metal head with a striking surface. Only after this initial examination can one conclude that it is a hammer. During planning, specification of the type precedes identification of the object. During analysis, examination of the object precedes identification of its type.

In traditional programming, the programmer is concerned with developing algorithms or plans for solving problems. Thus, traditional programming is a planning task. Because of this, traditional object oriented systems use classes as templates for the creation of new objects: if one needs a hammer object, one instantiates class hammer.

In contrast, retrieving data from heterogeneous databases requires the examination of existing data from diverse data sources. Thus, it is an analysis task, not a planning task, and traditional object oriented approaches are not suitable: classes should classify existing objects, not be templates for new objects.

Building on this observation, this research embraces analytic object orientation, an alternate approach to object orientation. An analytic object oriented system presupposes a universe of objects which it classifies into one or more classes or types. Each class contains a list of defining rules: objects belong to the class if they satisfy the rules. This is an analytic approach in which the existence of objects precedes their classification.

Analytic object orientation derives from both object oriented and rule based systems. From traditional object orientation, analytic object orientation borrows the concepts of object and class: all data items are objects, objects are members of classes, and objects receive messages and process them using methods which are defined at the class level and encapsulate processing within the object. From rule based models, analytic object orientation borrows the use of rules to define

membership. (And in fact, a rule based system, which uses declarative rules to define data items of interest, is an analytic system.) However, instead of defining satisfaction of a predicate, the rules of an analytic object database define membership in a class.

There are four practical differences between analytic object oriented systems and most traditional object oriented systems.

1) In a traditional object oriented database, all objects reside in data storage and queries identify objects that match search criteria. In an analytic object database, the universe of objects consists of both stored objects and objects that may be derived from the stored objects. For example, given HAMMER and NAIL objects, we can derive the existence of HAMMER-NAIL objects containing one hammer and one nail. Finding members of the class HAMMER-NAIL creates data representations of conceptual objects in much the same way that running a relational query generates a new table containing new rows of data.

2) In an analytic object oriented system a given object can satisfy the defining rules of several classes. Therefore, objects can be members of multiple classes. (Although some object oriented databases allow objects to be members of multiple unrelated classes, they are the exceptions.)

This is similar to the situation that arises in formal mathematical systems in which a data item can be a member of multiple types. For example, the number 2 is a member of the type of positive integers as well as the type of even integers.

3) The analytic object oriented approach to class-subclass relationships differs from that of traditional object oriented models. In traditional models, a subclass identifies its parent classes and inherits the attributes of those parents. In an analytic object oriented system, a class is a subclass to another if and only if the rules of the subclass imply

membership in the parent class. This is explicit if the defining rules of a class require membership in the parent. It is implicit if the rules of the subclass imply membership in the parent. Because this class-subclass relationship is fairly loose, analytic object oriented systems require multiple inheritance.

4) Analytic object oriented queries are class definitions. A database query returns data that satisfies specific rules. In terms of analytic object orientation, a query identifies the class of objects that are of interest. Thus, an analytic object oriented query is an ad hoc class definition and writing a query is indistinguishable from creating a new class.

In sum, analytic object orientation combines the strengths of rule based models (which tend to be more analytic in nature) with those of object oriented systems. As such, analytic object orientation serves as a good conceptual basis for a meta model combining the strengths of these two model types.

3.2. Cyrano: An Analytic Object Oriented Meta Model

This section describes Cyrano, a meta model based on analytic object orientation. Cyrano was designed as part of this research to serve as a meta model for an FDBS supporting relational, object oriented, and rule based member models. As such, Cyrano is representative of meta models based on both rule based and object oriented models. Cyrano was first presented by Dzikiewicz and Egyhazy [1994].

Section 3.2.1 contains an introduction to Cyrano, laying out Cyrano's basic approach to organizing data. To more fully demonstrate Cyrano, section 3.2.2 contains a Cyrano solution of a sample problem. Section 3.2.3 describes how Cyrano incorporates the features of object oriented and rule based models. Finally, section 3.2.4 describes how Cyrano could be used as a meta model by

discussing Cyrano's relation to the FDBS architecture presented in section 2.2.2.

3.2.1. An Introduction to Cyrano

A Cyrano database consists of a set of class specifications. Each class specification contains the class's defining rules and methods. All objects, either existing or potential, that satisfy any of a class's defining rules are members of the class. Upon being sent a message, an object invokes the corresponding method specified by its class specification.

Cyrano supports three types of classes, which are distinguished by the nature of their defining rules and methods.

1) Built-in classes represent atomic data types. Their defining rules and methods (operations) are implicit to their implementation. Built-in classes do not have Cyrano specifications. For example, **BOOLEAN** is a built-in class that supports the operations **AND**, **OR**, and **NOT**. Several built-in classes have methods for atomic type conversion. For example, the **INTEGER** class includes methods for conversion of integers to the **STRING** class.

2) A gateway class is the translation of a member database structure into the Cyrano model. The defining rule of a gateway class is implicit in this translation: a gateway class contains all Cyrano objects that correspond to data items stored as members of the class's corresponding structure. Thus, the class is a gateway through which the contents of the member data structure are viewed. A gateway object handles a message by returning the value of the corresponding member database attribute or function. Depending on the implementation, this can involve either returning an attribute of the object or translating the message into the format of the corresponding member database and passing it to the database.

3) Derived classes extract and reorganize data from other classes.

Each derived class contains a set of derivations, each of which has a logical rule that serves as a defining rule. All objects that satisfy a derivation rule are members of the class. Each derivation contains rules for handling messages: to handle a message, a derived object invokes the corresponding rules in the derivation used to identify it.

By providing a rule-based integration of other classes, derived classes handle database heterogeneity. By extracting desired data from other classes, derived classes act as queries.

This use of the term "derived class" differs from its use in programming languages such as C++. In C++, a derived class is one that inherits from another class.

3.2.2. A Sample Cyrano Application

This section describes a sample Cyrano application. In the course of this description, several features of Cyrano are clearly demonstrated. Section 3.2.2.1 describes the sample problem. Section 3.2.2.2 describes the Cyrano solution to the problem.

3.2.2.1. A Sample Federated Database Problem

Several news providers deliver news stories via real-time data feeds. Many of these feeds are geared towards keeping financial services providers informed about events that might affect the prices of securities. Although each provider has a different format, most include the following information with each story:

1) The text of the story.

2) The headline of the story.

3) The date of the story.

4) A list of keywords describing the contents and/or search categories of the story.

5) A list of stocks symbols of companies that are referenced in the story.

Unfortunately, there is neither a standard set of keywords or stock symbols nor a common convention for formatting keywords and symbols. For example, the Reuters North American Equities newswire identifies each stock with embedded text in the format "<STOCK.EX>", where "STOCK" is the name of the stock and "EX" is the code of the exchange on which it is traded (e.g., "IBM.N" is the symbol for IBM traded on the New York Stock Exchange) [Reuters 1992]. In its DowVision newswire, on the other hand, Dow Jones includes with each story a list of unique stock symbols for companies referenced in the story [Dow-Jones 1992]. These symbols do not necessarily include the exchange. To complicate the problem, different feeds use different symbols for the same companies. For example, one feed might use "COG" for Cognitive Engineering Inc, while another uses "COG" for Cogsworth's Cogs. These differences in symbology complicate the task of developing an information system that provides access to stories from multiple news providers.

A company maintains two databases of news stories. A relational database contains stories received from Reuters and has tables for stories, their keywords, and their stock symbols. An object oriented database contains DowVision stories and has classes for stories, symbols, and subfeeds of the primary feed (DowVision contains several subfeeds with different subject areas that are available to different subscribers). Subclasses of the symbols class contain stocks and keywords. In addition, there is a rule-based Symbology database that contains rules for normalizing Dow Jones and Reuters keywords and stock symbols into a standard set.

The company wants a federated database system that provides uniform access to Reuters and DowVision stories categorized by the normalized symbols.

This requires the integration of the DowVision and Reuters databases into a common format. It further requires that the symbols used at the global database layer are the normalized symbols indicated by the symbology database.

3.2.2.2. A Cyrano Solution

In accord with the reference architecture of section 2.2.2, the Cyrano solution contains several databases representing the different levels of database federation. The problem statement identifies three member databases, the Dow Jones, Reuters, and Symbology databases. Three Cyrano gateway databases provides gateways to these member databases. The Cyrano global database federates these gateway databases to provide a uniform view of the data. Finally, a user database presents a view for a user with specific requirements for his use of the database.

The problem statement contains three member databases, the relational Reuters, object oriented DowVision, and rule-based symbology databases. Each requires a gateway database to allow access by the federated system.

A Cyrano gateway database is a set of gateway classes. Each gateway class maps to a single member database structure. The implementation of the gateway class translates data requests from Cyrano format into the format of the member database.

Figure 3.1 shows the member schema for the Reuters database and its corresponding Cyrano gateway classes. The relational schema is shown as the SQL statements that create its tables. The gateway classes are simple syntactic translations of the relations. Tables become classes, rows become objects, and columns become supported messages. Gateway classes representing relations support one type of message, a request for a specific attribute.

The classes in figure 3.1 show the format of a Cyrano gateway class. The

```
create table Rt_Story (
        Headline char(80),
        LeadParagraph char(255),
        StoryDate date,
        ID char(16));

create table Rt_Keywords (
        Keyword char(20),
        StoryDate date,
        ID char(16));

create table Rt_Stocks (
        Stock char(20),
        StoryDate date,
        ID char(16));
```

3.1.a: Relational Schema.

```
CLASS Rt_Story IS GATEWAY WITH
        STRING Headline;
        STRING LeadParagraph;
        DATE StoryDate;
        STRING ID;
END CLASS.

CLASS Rt_Keywords IS GATEWAY WITH
        STRING Keyword;
        DATE StoryDate;
        STRING ID;
END CLASS.

CLASS Rt_Stocks IS GATEWAY WITH
        STRING Stock;
        DATE StoryDate;
        STRING ID;
END CLASS.
```

3.1.b: Cyrano Classes.

Figure 3.1: Reuters Database

header section names the class and identifies it as a gateway class. The message section identifies the messages that the class supports. A message specification includes the type returned by the message, the message name, and its arity (which defaults to 0).

An object oriented gateway class is somewhat more complicated. Figure 3.2 shows the schema for the DowVision database and its corresponding Cyrano gateway classes. The DowVision database is designed under the proposed standard object data model of the Object Database Management Group [Cattell et al. 1994]. Each gateway class corresponds to a DowVision member class. The gateway classes support a message for each attribute or operation of the member class.

As shown by the stock and keyword classes, a gateway class can be a subclass of another gateway class. When this occurs, the subclass handles all messages handled by its parent. A gateway class should only be made a subclass if the associated member database supports inheritance and if the subclass's corresponding member data structure is a subclass of the parent class's corresponding member structure.

As shown by the **Categories** message of the **DJ_Story** class, Cyrano does not directly support set attributes. Instead, Cyrano uses a predicate to indicate set membership. The Cyrano predicate returns true if its parameter is a member of the set and false otherwise.

Figure 3.3 shows the schema for the Symbology database and its corresponding Cyrano gateway classes. The Symbology database is a Datalog database. Each class corresponding to a predicate supports one message for each predicate parameter. The predicate parameters do not have names: the first class message returns the first predicate parameter, the second returns the second, and so on. There is no difference between gateways to intensional or extensional predicates: from Cyrano's perspective, the implementation of a

```
interface DJ_Story() : persistent
{
        attribute String Headline;
        attribute String Lead;
        attribute Date Story_date;
        relationship DJ_Feed Feed;
        relationship Set<Category> Categories;
};

interface DJ_Feed() : persistent
{
        attribute String Name;
};

interface DJ_Category() : persistent
{
        attribute String Label;
};

interface DJ_Stock : DJ_Category() : persistent
{};

interface DJ_Keyword : DJ_Category() : persistent
{};
```

3.2.a. Object Oriented Schema.

```
CLASS DJ_Story IS GATEWAY WITH
        STRING Headline;
        STRING Lead;
        DATE Story_date;
        DJ_Feed Feed;
        BOOLEAN Categories(1);
END CLASS.

CLASS DJ_Feed IS GATEWAY WITH
        STRING Name;
END CLASS.

CLASS DJ_Category IS GATEWAY WITH
        STRING Label;
END CLASS.

CLASS DJ_Stock IS GATEWAY IS DJ_Category WITH
END CLASS.

CLASS DJ_Keyword IS GATEWAY IS DJ_Category WITH
END CLASS.
```

3.2.b: Cyrano Classes.

Figure 3.2: DowVision Database

```
Predicates:
        S_Stock(3);
        S_Keyword(3);

Examples:
        S_Stock("RT", "IBM.N", "IBM");
        S_Stock("DJ", "IBM", "IBM");
        S_Stock("BT", base, normal) :- S_Stock("DJ", base, normal);
        S_Keyword("RT", "JP", "YEN");
        S_Keyword("DJ", "N/ERN", "EARNINGS");
```

3.3.a. Datalog Schema.

```
CLASS S_Stock IS GATEWAY WITH
        STRING Feed;
        STRING BaseStock;
        STRING NormalizedStock;
END CLASS.

CLASS S_Keyword IS GATEWAY WITH
        STRING Feed;
        STRING BaseCategory;
        STRING NormalizedCategory;
END CLASS.
```

3.3.b. Cyrano Classes.

Figure 3.3: Symbology Database

predicate as extensional or intensional is encapsulated within the rule-based database.

A Cyrano global database consists of a collection of derived classes. These derived classes import data from the classes in the gateway databases. Figure 3.4 shows the Cyrano classes for a global database for the sample problem.

Derived classes are the mechanisms by which Cyrano resolves database heterogeneity. A derivation can normalize the values of its base data through the processing of its methods. By having different derivations perform different normalizations, a derived class can normalize data from different sources in different formats. For example, the class **Stories** normalizes the formats of **RT_Story** and **DJ_Stories** into a single uniform format.

Derived classes can also normalize the types of attributes from different gateway databases. When normalizing complex types (i.e., gateway and derived classes), the derived class extracts atomic data attributes from the source and reconstructs them as required. When normalizing simple types (i.e., built-in classes), the derived class uses type conversion methods provided by the built-in classes to normalize on a uniform type.

A derived class definition consists of one or more sets of derivation rules. Each derivation rule starts with a list of the base variables and their classes. The classes are the base classes of the derived class. Base classes can be any defined class, including the current class: a derived class can be recursive. In the example, the base variable for the first derivation of **Stories** is **s** of type **Rt_Story**.

The second part of the derivation rule is the defining rule or guard. The guard is a boolean equation in which the base variables appear as the only free variables. In order to find members of the derived class, Cyrano finds all values of the base variables that make the guard be true. Each such set of values serves as the base objects of a corresponding derived object. In the example, the guard

```
CLASS Stories IS DERIVED WITH
        Rt_Story s : TRUE
        WITH
                STRING Feed IS "RT";
                STRING Subfeed IS "";
                DATE Date is s.StoryDate;
                STRING ID IS s.ID;
                STRING Headline IS s.Headline;
                STRING Lead is s.LeadParagraph;
        END;

        DJ_Story s : TRUE
        WITH
                STRING Feed IS "DJ";
                STRING Subfeed IS s.Feed.Name;
                DATE Date IS s.StoryDate;
                DJ_Story ID is s;
                STRING Headline IS s.Headline;
                STRING Lead IS s.Lead;
        END;
END CLASS.

CLASS Stock IS DERIVED WITH
        Stories s, RT_Stocks r, S_Stock st :
                r.StoryDate = s.Date AND
                s.Feed = "RT" AND
                r.ID = s.ID AND
                r.Stock = st.BaseStock AND
                st.Feed = "RT"
        WITH
                STRING Symbol IS st.NormalizedStock;
                Stories Story IS s;
        END;

        DJ_Stock dj_s, S_Stock st, Stories s :
                s.Feed = "DJ" AND
                s.ID.Categories(dj_s) AND
                dj_s.label = st.BaseStock AND
                st.Feed = s.Subfeed
        WITH
                STRING Symbol IS st.NormalizedStock;
                Stories Story IS s;
        END;
END CLASS.

CLASS Category IS DERIVED WITH
        Stories s, RT_Keywords r, S_Keyword k :
                r.StoryDate = s.Date AND
                s.Feed = "RT" AND
                r.ID = s.ID AND
                r.Keyword = k.BaseCategory AND
                k.Feed = "RT"
        WITH
                STRING Symbol IS k.NormalizedCategory;
                Stories Story IS s;
        END;

        DJ_Keyword dj_k, S_Keyword k, Stories s :
                s.Feed = "DJ" AND
                s.ID.Categories(dj_k) AND
                dj_k.label = k.BaseCategory AND
                k.Feed = s.Subfeed
        WITH
                STRING Symbol IS k.NormalizedCategory;
                Stories Story IS s;
        END;
END CLASS.
```

Figure 3.4: A Global Database

for the first derivation of **Stock** is a complex rule that ensures that stock **r** refers to story **s** and that **r** and **st** refer to the same stock.

The third part of the derivation rule is the method section. This identifies the methods used to handle messages to members of the class. Each method specification consists of the name of the message followed by a function on the base objects and message parameters. The result of this function is the value returned on receipt of the message. In the example, the method for **Symbol** in both derivations of **Stock** is **st.NormalizedStock**.

Taken together, the classes of the global database provide a uniform view of Reuters and DowVision messages and symbols. The **Stories** class unifies the messages. The **Stock** class links normalized stock symbols to stories that they describe. The **Category** class does the same for keywords.

A Cyrano user database consists of a set of derived class definitions. These classes can retrieve data from derived classes in the global database. Figure 3.5 gives Cyrano class definitions for a user database that presents a simplified view of news stories for a user with an interest in computers. The user database demonstrates several features of derived classes.

The three classes in the user database demonstrate three uses for derived classes. **U_Category** shows how a derived class can combine the objects from two other derived classes: **U_Category** is the union of **Stock** and **Keyword**. **U_Headline** shows how a derived class can combine objects into composite objects: **U_Headline** is the join of **U_Category** and **Story**, projecting on the **Symbol** and **Headline** attributes. **U_Computer_Headline** shows how a derived class can be a subclass of another class similar to the result of the relational SELECT operator: **U_Computer_Headline** selects all **U_Headline** objects that refer to computer-oriented symbols.

U_Computer_Headline also demonstrates derived class inheritance. A

```
CLASS U_Category IS DERIVED WITH
        Stock s : TRUE
        WITH
                STRING Symbol IS s.Symbol;
                STRING Type IS "Stock";
                Stories Story IS s.Story;
        END;

        Keyword k : TRUE
        WITH
                STRING Type IS "Keyword";
                STRING Symbol IS k.Symbol;
                Stories Story IS k.Story;
        END;
END CLASS.

CLASS U_Headline IS DERIVED WITH
        U_Category c : TRUE
        WITH
                STRING Symbol IS c.Symbol;
                STRING Headline IS c.Story.Headline;
        END;
END CLASS.

CLASS U_Computer_Headline IS DERIVED IS U_Headline WITH
        :
                THIS.Symbol = "IBM" OR
                THIS.Symbol = "HP" OR
                THIS.Symbol = "APPLE" OR
                THIS.Symbol = "MSOFT" OR
                THIS.Symbol = "SUN"
        WITH
        END;
END CLASS.
```

Figure 3.5: A User Database

derived class can be a subclass of another derived or gateway class. When this occurs, an object of the subclass must satisfy the guard of a derivation of each parent class. This results in the derived object having a set of base objects for both subclass and parent class derivations. A derived class inherits the methods of its parents. (Note that, for derived classes, methods are tied to derivations and not to the class as a whole. Therefore, two objects of the same subclass may not inherit support for the same parent class messages.) Where both subclass and parent have a method for the same message, the subclass's method takes precedence. When two parent classes support the same method, the subclass inherits the method of the first parent listed in its specification. In the example, **U_Computer_Headline** is a subclass to **U_Headline**. **U_Computer_Headline** inherits all **U_Headline**'s methods and has none of its own methods or base objects.

A derivation guard and methods of a subclass may invoke methods of the parent class by using the keyword "THIS" to send messages to itself. For example, **U_Computer_Headline**'s guard uses its parent's **Symbol** method.

3.2.3. Cyrano and Object Oriented and Rule Based Data Models

This section describes Cyrano in relationship to object oriented and rule based data models. Section 3.2.3.1 describes Cyrano as an object oriented model. Section 3.2.3.2 describes those features of rule based models incorporated into Cyrano.

3.2.3.1. Cyrano as an Object Oriented Data Model

Cyrano supports the primary data structuring features of object oriented data models: object identity, aggregation, and generalization. In addition, Cyrano supports the encapsulation of processing within an object.

In an object oriented database, each object has a unique identity. This means that data objects have unique existence independent of the values of data that they include. This differs from a relational database, in which a row has no identity beyond being a collection of its data elements: change one data element, and it is a different row.

Cyrano supports object identity by maintaining separate objects for each way that an object can be found to be a member of a class. Object identity is ultimately tied to the identity of the underlying member objects. This relationship is direct for gateway objects, each of which represents one such member object. It is indirect for derived objects, which are tied to the gateway (and thus member) objects from which they are ultimately derived.

Object aggregation means that objects can contain other complex objects. This differs from relational databases, in which rows contain only atomic data items.

Because Cyrano takes a strict object oriented approach, object attributes are not directly visible outside of the object (and in fact are not identified in the class specification). However, Cyrano provides attributes by its use of member functions: a member function with no parameters returning a value of class A can be said to be an attribute of type A. Because methods can return any class, a Cyrano object can have attributes of any class. Cyrano methods can represent complex ways of combining data objects (e.g., lists and sets): the sample application demonstrates how a method can represent a set attribute.

Generalization is the inheritance relationship between classes. Cyrano directly supports inheritance.

Finally, object oriented systems encapsulate processing of object methods within the object. Users of an object do not need to know how the object processes its messages. In keeping with this, the implementation of a Cyrano object's processing is encapsulated within the object.

3.2.3.2. Cyrano and Rule Based Data Models

Cyrano borrows from rule based models both in the types of classes that it contains and in the types of rules that it uses.

A rule based data model uses predicates to represent complex data structures. A rule based system supports two types of predicates: extensional, which serve as a gateway to stored data, and intensional, for which the system processes rules to find the satisfying data items. Cyrano's class types correspond to these two predicate types.

A Cyrano gateway class corresponds to an extensional predicate. Like an extensional predicate, finding the members of a gateway class requires retrieving them from storage. While most rule based systems retrieve extensional data from a local data store, Cyrano satisfies its gateway classes by extracting data from remote heterogeneous databases.

A Cyrano derived class corresponds to an intensional predicate. Like an intensional predicate, finding the members of a derived class requires processing rules. In an intensional predicate, these rules describe the data that satisfies the predicate. In a derived class, the guard rule of a derivation describes those objects that are members of the class. As the rule based system uses all of a predicate's rules to find data that satisfies the predicate, so a Cyrano class contains the objects that satisfy any guard of any derivation of the class.

In a rule based database, querying the database requires writing rules for a new intensional predicate. Similarly, querying the Cyrano database requires specifying a new derived class. Thus, intensional predicates and derived classes represent queries in their respective data models.

Rule based databases represent rules with a variant of predicate calculus. Typically, as in the case with Datalog, a rule based system uses Horn clauses to represent logical rules. Cyrano uses the basic predicate calculus as the basis for its rules, allowing boolean operators and universal and existential quantifiers.

3.3. Cyrano and Other Object Oriented/Rule Based Hybrid Systems

The literature discusses several approaches to uniting object oriented and rule based systems.

Some of the early work on combining these approaches involves integrating object oriented and rule based programming systems [Chen and Warren 1988]. Typically, this involves having rules manipulate objects, thus adding sophisticated data typing to a rule based programming system. In these cases, rules are external to the objects that they act upon and not encapsulated within them, as in traditional object oriented systems. Early efforts at integrating object oriented and rule based databases followed this same approach [Greco and Rullo 1989].

The Telos database language, as used as part of the ConceptBase database, follows a different approach [Staudt et al. 1994; Jarke et al. 1995]. Telos provides three views of a database: rule-based, concept based, and object oriented. The rule-based approach is similar to the earlier method: external rules act on objects. However, the object oriented view comes closer to Cyrano.

In the object oriented view, Telos uses typical object oriented class definitions specifying the attributes, superclasses, and methods of classes. Unlike typical object oriented class definitions, in which methods map to functions programmed in a procedural language, Telos method definitions consist of sets of rules for generating their method results.

Like Cyrano, Telos uses ad hoc query class definitions as queries. Each Telos query class is a subclass to an existing class and has rules specifying which members of the parent class are also members of the query class. The query class may be a subclass to multiple parent classes. For example, the query class **HIGH-PAID-WOMEN** is a subclass to **WORKERS** and **WOMEN** with the added constraint that the **SALARY** attribute of **WORKERS** exceeds 50,000.

Telos differs from Cyrano in its approach to objects. Telos takes a traditional object oriented approach to objects: objects are specific data values

stored as extensional data in a data store. Even the members of query classes are objects gotten from parent classes. This contrasts with Cyrano objects, which can be generated as part of class instantiation by combining data from other objects.

In summary, Telos is similar to Cyrano in that both integrate rules into an object oriented framework. Further, both Cyrano and Telos queries are class definitions containing rules that define class membership. Telos differs from Cyrano in that it follows the traditional object oriented view in which class precedes object. In the terms presented in section 3.1, Telos is primarily a planning object oriented system as opposed to an analytical system.

Telos is meant as a traditional data model, but [Staudt et al. 1994] discusses the possibility of using it as a meta model.

Chapter 4: Criteria for Evaluating Meta Models

This chapter addresses the question of how to evaluate Cyrano as a meta model. As defined in section 1.1.4, a meta model is the common or unifying data model of an FDBS. In this context, a data model is a collection of three elements [Codd 81]:

1) A set of data structure types used to construct complex data items;

2) A set of operators which can be applied to these data structures;

3) A collection of integrity rules which identify the acceptable database states.

Therefore, a meta model is a collection of data structures, operators, and integrity rules that serve to normalize and integrate multiple heterogeneous databases.

This chapter identifies three criteria for the evaluation of a meta model. The chapter expresses these criteria in terms of the reference architecture presented in section 2.2.2. However, the concerns addressed by these criteria are applicable to most federated database systems. For systems based on other architectures, the terminology used in the criteria may need adjustment.

The three criteria for meta model evaluation are:

1) A meta model must fully support the translation from member database to gateway database.

2) A meta model must fully support the federation of gateway databases into a global database.

3) It must be possible to implement the meta model as part of an FDBS.

In discussing these criteria, this chapter uses the symbols $=^{i}$, $\subset^{i}$, and $\subseteq^{i}$ to indicate relationships between sets of query results. In particular, ($\mathbf{A} =^{i} \mathbf{B}$) is used to indicate that query results **A** contains the same information as query results **B**,

($\mathbf{A} \subset^{i} \mathbf{B}$) is used to indicate that query results A contains less information than query results B, and ($\mathbf{A} \subseteq^{i} \mathbf{B}$) is used to indicate that either ($\mathbf{A} =^{i} \mathbf{B}$) or ($\mathbf{A} \subset^{i} \mathbf{B}$).

The subsequent three chapters use these criteria to establish that Cyrano is a suitable meta model.

4.1. Support of Member Data Models

An FDBS exists to allow users access to the data facilities of its member databases. To do so, the meta model must support full access to all databases that can be built using the member models. When a meta model provides such access, it fully supports the member data models.

4.1.1. Definition of Full Support

This section defines when a meta model fully supports a member model. As a first step in defining support, the section identifies when a gateway database fully supports a member database. With this definition, the section defines full support of a member model as the ability to define a gateway database for each possible member database.

As with the FDBS reference architecture in section 2.2.2, these definitions take a black box view of databases, emphasizing the information that can be retrieved from the database. The retrievable information of the database is a function of all of the elements of its data model: the data structures and constraints determine the data that can be stored in the database, while the operators determine the ways in which queries can extract information from the stored data. By emphasizing retrievable information, this dissertation focuses on query operators: other operators are left to future research.

A gateway database fully supports a member database if a user can use the gateway database to answer any question that he could answer using the member database. In other words, for any query against the member database,

there is a query against the gateway database that returns equivalent results. This leads to definition 4.1:

Definition 4.1: Given:

MEMBER is a member database;

GATEWAY is a gateway database serving as a gateway to the data in MEMBER;

Q(2) is the query function: for any query q and database d, Q(q, d) returns the set of data generated by running q against d;

Then:

GATEWAY fully supports MEMBER if and only if:

$$\forall(q_1)\exists(q_2)(Q(q_2, \text{GATEWAY}) =^{i} Q(q_1, \text{MEMBER}))$$

Definition 4.1 takes a black box view of database support: gateway database A supports member database B if A can produce all of the outputs of B. To some extent, this is a measure of the power of the database's query languages: if B's query language is more powerful, then A will be unable to generate all of the outputs that B can generate.

This is not to say that every query against the gateway database has an equivalent query against the member database. The meta query language can be more powerful then the member query language. This is necessary if the FDBS is to fully support different data models with different query capabilities. In this case, the meta query language must provide a common superset of the query languages of its members.

Definition 4.2 builds on definition 4.1 to define when a meta model fully supports a member data model.

Definition 4.2: Given:

MEMBER is a data model;

META is a meta model;

{member} is the set of all databases that can be represented using the data structures and satisfying the constraints of MEMBER;

{meta} is the set of all databases that can be represented using the data structures and satisfying the constraints of META;

SUPPORTS(2) is a boolean function that applies definition 4.1 to 2 databases: SUPPORTS(D_1, D_2) returns true if and only if gateway database D_1 fully supports member database D_2 according to definition 4.1;

Then:

META fully supports MEMBER if and only if:

$$\forall(d_1|d_1 \in \{member\})\exists(d_2|d_2 \in \{meta\})(SUPPORTS(d_2, d_1))$$

Informally, definition 4.2 states that a meta model fully supports a member data model when every possible member database is fully supported by some gateway database.

If a member model is more capable of organizing data then the meta model, then there will exist member databases for which there are no corresponding gateway databases. Thus, to some extent the satisfaction of definition 4.2 is a measure of the data structuring capabilities of the meta model compared to its member models.

Taken together, definitions 4.1 and 4.2 identify when a meta model fully supports a member model. Under these definitions, full support consists of the ability of the meta model (and thus of the FDBS) to answer any question that could be answered by a database built under the member model. If this holds, then a

user does not sacrifice any data retrieval capability by using the FDBS instead of accessing the member DBMS directly.

4.1.2. Necessity of Full Support

The necessity for a meta model to provide full support to its member models arises from the purpose of an FDBS. An FDBS exists to provide uniform access to independent databases. The requirement to provide access results in a need to provide full support to the member databases. The requirement to support independent databases results in the need to provide full support for all possible member databases, and thus to provide full support to the member model.

When a user queries a database, he issues questions to a black box. That block box takes his questions in the form of queries and returns answers in the form of query results. For the user, these answers are the end product of his access.

If a user, by using an FDBS, is limited in the answers that he can receive, then he is limited in the ways that he can access the database. This might be acceptable for an application in which the user's data needs are constrained in some fashion. However, in the general case, a user should not sacrifice data access as a result of using the FDBS. Therefore, by definition 4.2.1, the meta model should provide full support to the member databases of its federations.

This might suggest that a meta model could constrain the member databases to some supported subset of the possible databases. However, the independence of member databases argues against this.

An FDBS must provide access to pre-existing and independent member databases. If the meta model required some constraints on supported databases, then it would be unable to support access to some possible members. This happens because the independence characteristic of members means that they are designed without consideration for their place in the federation.

Further, even if a given database is a fully supported member, the independence characteristic means that the database might be changed in some unpredictable manner. If the meta model does not fully support all possible member databases built under the member model, then such changes could lead to a lack of support for a previously supported member database. This would be unfortunate for those users dependent on the member.

To ensure that the meta model can support any desired database, and to ensure that it continues to support those databases that it already supports, the meta model must be able to support all possible databases built using the member model. By the definitions of section 4.2.1, this means that the meta model must fully support the member model.

4.1.3. Demonstration of Full Support

Given definitions 4.1 and 4.2, we can prove that a meta model fully supports a member data model by providing:

1) A translation function T() that translates results from the format of the member data model into the format of the meta model. (T() overcomes any purely syntactic differences in the query results format of the member model and the meta model.)

2) An algorithm A1 that, given a member database in the member data model, generates a gateway database in the meta model.

3) An algorithm A2(Q) that, given a member database, a gateway database generated by A1, and a query Q against the member database in the member query language, generates an equivalent query against the gateway database in the meta query language.

4) Proof that the result of applying T() to the result of any query Q against the member database is equal to the result of running the query

generated by A2(Q) against the gateway database generated by A1.

Chapter 5 uses this method to prove that Cyrano, the meta model introduced in chapter 3, fully supports the relational, object oriented, and rule-based data models.

4.2. Support of Database Integration

The primary job of the global database is to integrate the data of the gateway databases. This section discusses the requirement that a meta model needs to include mechanisms for this integration. Section 4.2.1 defines what is meant by support for database integration. Section 4.2.2 argues that a meta model must provide such support. Section 4.2.3 provides criteria for judging whether a meta model provides sufficient integration support.

4.2.1. Definition of Integration Support

Meta model support for database integration involves providing features that allow the complete and frugal federation of gateway databases into a global database. A complete integration is one that preserves all data from the gateway databases in the global database. A frugal integration is one that produces no data beyond that found in the gateway databases. A meta model supports database integration when it provides model-specific features that integrate multiple gateway databases into a single global database.

In integrating data, a global database is incomplete when some information can be retrieved from a gateway database that cannot be retrieved from a global database. Ideally, a global database will be a complete integration of its member databases. Definition 4.3 defines a complete integration.

Definition 4.3: Given:

Q(2) is the query function: for any query q and database d, Q(q, d) returns the set of data generated by running q against d;

GLOBAL is a global database;

$\{g_1, ..., g_n\}$ is a set of gateway databases;

Then:

GLOBAL provides a complete integration of $\{g_1, ..., g_n\}$ if and only if:

$$\forall(m:1\leq m\leq n)\forall(q_1)\exists(q_2)(Q(q_1, g_m) \subseteq^{i} Q(q_2, GLOBAL))$$

In other words, a complete integration occurs only when every query against any gateway database has an equivalent query against the global database that returns some superset of the results of the gateway query.

While complete integration is desirable in many cases, sometimes it is not necessary. If the users of a global database do not need (or should not have) access to some piece of information from the gateway database, then the global database need not preserve that information. This will often be the case in multiple federation systems where there are several global databases supporting different user communities.

A frugal integration is one that produces no extraneous data from outside of the gateway databases. Definition 4.4 specifies when a frugal integration occurs:

Definition 4.4: Given:

Q(2) is the query function: for any query q and database d, Q(q, d) returns the set of data generated by running q against d;

GLOBAL is a global database;

$\{g_1, ..., g_n\}$ is a set of gateway databases;

Then:

GLOBAL provides a frugal integration of $\{g_1, ..., g_n\}$ if and only if:

$$\forall(q_0)\exists(q_1, ..., q_m)\exists(X_1, ..., X_m : 1 \le X_i \le n)$$
$$Q(q_0, \text{GATEWAY}) \subseteq^{i} (Q(q_1, g_{X1}), ..., Q(q_m, g_{Xm}))$$

In other words, an integration is frugal when the result of every query against the global database is a function of the results of some set of queries against the gateway databases.

For a meta model to support database integration, it must provide features to integrate gateway databases. Gateway database differences can include any of the types of database heterogeneity identified in section 2.1. A meta model's integration facilities must be capable of resolving these types of heterogeneity to integrate the gateways into a single global database.

For example, a Cyrano derived class can integrate data in gateway classes from different gateway databases. The Cyrano database administrator can write derivations for the class that specify how heterogeneity is resolved. These normalize the gateway classes into one unified format.

As identified in section 2.3, different meta models provide different kinds of features for database integration. These include the articulation axioms of Carnot [Collet et al. 1991], the abstract classes, aliases, and derived functions of Pegasus [Ahmed et al. 1991], and the derived clusters of Extended Database Logic [Grant and Sellis 1990]. All of these allow different global database structures to be integrated to provide a uniform view of the data.

4.2.2. Necessity of Integration Support

As specified in the definition of data models, a data model includes operators for manipulating data. In a traditional database, the data manipulations include querying, database update, and modifications to the database structures.

The operators for these manipulations are embodied in query and data languages for the data model.

A tightly coupled FDBS provides an additional data manipulation: the integration of gateway databases to form a global database. The meta model, as the data model of the FDBS, should provide operators for this additional data manipulation. Therefore, a meta model needs specific mechanisms to allow the integration of its gateway databases.

Several FDBS's provide data integration outside of the meta model. For example, Mermaid uses the relational model as a meta model and provides integration outside of the meta model [Templeton et al. 1986], requiring administrators and users to use additional mechanisms outside of the model in order to perform data manipulation. This approach emphasizes heterogeneous data structures while ignoring the data operators required to overcome this heterogeneity. This is similar to the view of a data model as only a collection of data structure types, a view described by Codd [1981] as misguided.

4.2.3. Demonstration of Integration Support

One can show that a meta model supports database integration by showing that it can resolve the various types of database heterogeneity. As discussed in section 2.1, there are many types of database heterogeneity. If a meta model can resolve all of these, then it includes sufficient features to integrate databases.

The approach to such a demonstration will vary based on the meta model and the type of heterogeneity. In general, such a demonstration should show that the resolution can handle any occurrence of the heterogeneity type. It should also show that the resolution is both complete and frugal.

Chapter 6 contains demonstrations that Cyrano can resolve the types of heterogeneity identified in section 2.1.

4.3. Demonstration of the Implementation of a Meta Model

The heterogeneous database problem is a practical problem. Users accessing data from different data sources encounter this problem daily. Further, much of the research on heterogeneous databases has been done in an industrial setting: most of the federated database systems (FDBS) described in section 2.2.3 were developed by commercial researchers seeking solutions to corporate data management problems.

As the problem is practical, so must be the solution. As part of such a solution, a meta model must be practical. For this to be the case, it must be possible to implement the meta model as part of an operational FDBS. This requirement serves as the third criterion for judging meta models.

The primary way of demonstrating that a meta model can be implemented is to implement it. Therefore, a proof-of-concept FDBS using the meta model is the best demonstration of the practicality of that meta model.

4.4. The Sufficiency of the Three Criteria

Sections 4.1 through 4.3 established that it is necessary that a meta model satisfy the following three criteria:

1) A meta model must fully support the translation from member database to gateway database.

2) A meta model must fully support the federation of gateway databases into a global database.

3) It must be possible to implement the meta model as part of an FDBS.

This section demonstrates that these three criteria are sufficient for evaluating a meta model. This section derives this sufficiency from the requirements of a meta

model.

A meta model is the data model of an FDBS. Therefore, it should be judged based on its ability to support the data manipulations unique to an FDBS. Section 2.2.2 identifies the unique data manipulations of FDBS's to be the translation of data from member to gateway database and the integration of gateway databases into a global database.

The first criteria addresses the first of these data manipulations. The first criteria requires that the meta model allow a complete translation of data and transactions between the meta model and any of its member models. Therefore, the first criteria is sufficient to measure the data translation of a meta model.

The second criteria addresses the second of these data manipulations. The second criteria requires that the meta model support the integration of gateway databases. Therefore, it measures the meta model's capability of supporting the second data manipulation.

The third criteria addresses the relationship of a meta model to an FDBS. The third criteria states that the meta model be able to be implemented as part of an FDBS. Therefore, it measures the ability of the meta model to serve as part of an FDBS.

Taken together, these criteria evaluate the meta model's suitability to fulfill its role as part of an FDBS. Since fulfilling this role is the central purpose of a meta model, these criteria are sufficient to evaluate the meta model.

Chapter 5: Cyrano Support for Existing Data Models

The first of the three criteria for evaluating a meta model is its support for member data models. This chapter presents proofs that Cyrano fully supports relational, object oriented, and rule based data models. These proofs take the form specified in section 4.1.3, consisting of the following:

1) A translation function T() that translates results from the format of the member data model into the format of the meta model.

2) An algorithm A1 that, given a member database in the member data model, generates a gateway database in the meta model.

3) An algorithm A2 that, given a member database, a gateway database generated by A1, and a query against the member database in the member query language, generates an equivalent query against the gateway database in the meta query language.

4) Proof that the result of applying T() to the result of any query Q against the member database is equal to the result of running the query generated by A2 given Q against the gateway database generated by A1.

This chapter contains proofs in this form that Cyrano supports the three target member models: relational, rule-based, and object oriented.

This chapter contains several algorithms for generating Cyrano derived classes that are equivalent to member database queries. These algorithms are designed for use in these proofs. They do not necessarily produce the optimum equivalent classes: they merely produce classes that are easy to prove equivalent to the original queries. Thus, these algorithms should not be used as guides for constructing Cyrano classes.

5.1. Relational Models

This section proves that Cyrano fully supports relational data models as member models. It describes a translation function T() that translates relational query results into Cyrano format, an algorithm A1 that translates a relational schema into a Cyrano gateway database, and an algorithm A2 that translates a relational query into an equivalent Cyrano class specification. It concludes with a proof that T(), A1, and A2 are all valid and complete, and thus Cyrano supports relational member models.

This proof uses domain predicate calculus as a sample relational model. Domain predicate calculus is one of the mathematical formalisms on which the relational model is founded, the others being relational algebra and tuple calculus. (A proof of the equivalence of relational algebra, tuple calculus, and domain calculus is presented by Ullman [1982].) In domain predicate calculus, a table with N columns is represented by an N-ary predicate. Each row in the table is an N-tuple that satisfies the predicate. Queries are well-formed formulae in which the free variables form a row of the table of query results.

Domain predicate calculus does not assign types to atomic data items. To represent untyped atomic objects in Cyrano, the following discussion uses a built-in class called OBJECT. OBJECT supports all operations of all built-in classes.

5.1.1. A Translation Function T() for the Relational Model

A relational query returns an MxN table in which each of M rows is an N-tuple that satisfies the query. Cyrano represents each row by an object with N attributes, each of which has the value of the corresponding element of the tuple. The set of M Cyrano objects corresponding to the M tuples in the table forms a Cyrano class that is equivalent to the table.

(The statement, "Cyrano object X has attribute Y with value Z," means that X, upon being sent message Y, will return value Z. In this chapter, this is

represented by the statement, "X.Y = Z".)

Figure 5.1 contains $T_{Rel}()$, the translation function for the relational model.

5.1.2. An Algorithm A1 for the Relational Model

The schema of a relational database is a set of predicates, each of which provides access to a stored table. In Cyrano, an N-ary predicate has an equivalent class with N attributes. The attribute values of each member of the equivalent class correspond to an N-tuple retrieved from the relational database. Figure 5.2 contains the algorithm to generate the Cyrano gateway classes for a set of predicates.

5.1.3. An Algorithm A2 for the Relational Model

A query in domain calculus returning an N-tuple is a well-formed formula in which there are N free variables, one for each place in the N-tuple. The values of the variables that satisfy the query form the N-tuple. Figure 5.3 contains algorithm $A2_{Rel,}$ which forms the Cyrano derived class that is equivalent to the results of the query.

5.1.4. Proof of Correctness for T(), A1, A2

Before proving the correctness of $T_{Rel}()$, $A1_{Rel}$, and $A2_{Rel}$, we prove a lemma.

Lemma 5.1:

For any:

1) Relational schema R;
2) Predicate P(N) chosen from R;

and given:

1) Table T containing all tuples that satisfy P(N);
2) The set G of Cyrano gateway classes generated by $A1_{Rel}$

$T_{Rel}()$:

Input:

TABLE[M,N], an MxN table of values

Output:

M objects, each with values for attributes $\{A_1, A_2, ..., A_N\}$

Processing:

```
DO for I := 1 to M
        Create object O_I
        DO for J := 1 to N
                Set O_I.A_J = TABLE[I, J]
        ENDDO
ENDDO
```

Example:

Where TABLE[M,N] is:

1 2 3
4 5 6

Create the following set of objects:

O_1: $O_1.A_1 = 1$, $O_1.A_2 = 2$, $O_1.A_3 = 3$
O_2: $O_2.A_1 = 4$, $O_2.A_2 = 5$, $O_2.A_3 = 6$

Figure 5.1: A Translation Function for Relational Databases

$A1_{Rel}$:

Input:

A domain calculus schema consisting of definitions for a set of predicates

Output:

A corresponding Cyrano gateway database

Processing:

```
DO for all predicates P(N) in the input schema
        Create gateway class P
        DO for I := 1 to N
                Add support for message A_I
        ENDDO
ENDDO
```

Example:

For a domain calculus schema with definitions for:

PERSON(5)

Create the following Cyrano gateway database:

```
CLASS PERSON IS GATEWAY WITH
        OBJECT A1;
        OBJECT A2;
        OBJECT A3;
        OBJECT A4;
        OBJECT A5;
END CLASS.
```

Figure 5.2: Algorithm A1 for the Relational Model

$A2_{Rel}$:

Input:

A domain calculus query Q with free variables X_1, ..., X_N returning N-tuples

Output:

A Cyrano derived class C

Processing:

Create Cyrano class C
Create a derivation D of class C
DO for i = 1 to N
Create base variable X_i of class OBJECT for D
ENDDO
DO for all predicates $P_i(N_i)$ referenced in Q
Create a base variable O_i of class P_i for C
Replace all references to $P_i(V_1, ..., V_{Ni})$ in Q with:
(O_1=V_1 AND ... AND O_{Ni}=V_{Ni})
ENDDO
Use the resulting Q as the guard for C
Create the following methods section for C:
OBJECT A_1 IS X_1; ...; OBJECT A_N IS X_N

Example:

Given the query:
(X, Y, Z: PARENT(X, Y) AND PARENT(Y, Z))
Create the derived class:

```
CLASS C IS DERIVED WITH
        OBJECT X; OBJECT Y; OBJECT Z;
        PARENT O1; PARENT O2 :
        ((O1.A1=X AND O1.A2=Y) AND
                (O2.A1=Y AND O2.A2=Z))
        WITH
                OBJECT A1 IS X;
                OBJECT A2 IS Y;
                OBJECT A3 IS Z;
        END;
END CLASS.
```

Figure 5.3: Algorithm A2 for the Relational Model

applied to R;

3) The Cyrano class C from G that corresponds to P(N);

4) The set O of objects that are members of C;

Then:

$O = T_{Rel}(T)$

In other words, $A1_{Rel}$ generates a set of gateway classes that are equivalent to the input schema.

The Cyrano implementation of a relational gateway class creates a member object for each tuple in the associated relation. Therefore, lemma 5.1 is true according to Cyrano's implementation of the relational gateway class.

With this in hand, we turn to Theorem 5.1:

Theorem 5.1:

For any:

1) Relational schema R;

2) Query Q against R with N free variables;

and given:

1) The table T that satisfies Q;

2) The set G of Cyrano gateway classes generated by $A1_{Rel}$ applied to R;

3) The Cyrano class C generated by $A2_{Rel}$ applied to Q;

4) The set O of all objects that are members of C;

Then:

$O = T_{Rel}(T)$

In other words, $A2_{Rel}$ generates an equivalent class for any relational query and thus Cyrano fully supports the relational model.

We can rephrase theorem 5.1 to state that a tuple $<V_1, ..., V_N>$ satisfies Q if and only if there exists an object O in class C such that ($O.A_1=V_1$ AND ... AND $O.A_N=V_N$). This is equivalent to saying that the values $<V_1, ..., V_N>$, when substituted for $<X_1, ..., X_N>$ in the guard of C, make the guard true. This can be proven by induction on the number of logical operators in Q.

In the base case, with zero operators, Q is a call to some predicate P(M) in the form $P(V_1, ..., V_M)$. The derivation rule will substitute for this ($P.A_1=V_1$ AND ... AND $P.A_M=V_M$). By lemma 5.1, the substitution is true if and only if $<V_1, ..., V_M>$ is a member of P's table. As this is the desired semantic, theorem 5.1 is true for the base case.

The inductive step is easily accomplished by noting that the logical operators available in domain calculus have the same meaning as the equivalent boolean operators available in Cyrano. Thus, given that the logical atoms are correct (as proven in the base case), the logical formulae will be true.

Thus, theorem 5.1 is true, and Cyrano fully supports the relational model.

5.2. Rule-Based Models

This section proves that Cyrano fully supports rule based data models as member models. It includes a translation function T() that translates rule based query results into Cyrano format, an algorithm A1 that translates a rule based schema into a Cyrano gateway database, and an algorithm A2 that translates a rule based query into an equivalent Cyrano class specification. It concludes with a proof that T(), A1, and A2 are all valid and complete, and thus Cyrano supports rule based member models.

This proof uses Datalog [Ullman 1989] as a sample rule-based data model. Datalog provides a query language that is similar to the programming language Prolog. It uses Horn clauses to represent logical relations and supports two types of predicates. Extensional predicates are the interfaces to stored relations: finding

the data that satisfies the predicate requires accessing the data in storage. Intensional predicates have rules associated with them: finding the data that satisfies the predicate requires processing the rules. These predicates are similar to the classes of Cyrano, with extensional predicates similar to gateway classes and intensional predicates similar to derived classes.

A Datalog schema is a collection of predicates and their defining rules. The extensional predicates map stored data into Datalog. Intensional predicates invoke queries which the database solves by performing the corresponding rules. To query the database, the user writes the rules for a new intensional predicate. In doing so, he extends the schema by the new predicate.

For use with Cyrano, this proof assumes that the schema of a rule-based database consists of a set of intensional and extensional predicates that are known to be part of the database. For each such predicate, there is a corresponding Cyrano gateway class. The Datalog query language is held to be the language in which new intensional predicates are specified.

Sections 5.2.1 through 5.2.4 prove that, for any Datalog predicate with rules that contain only recursive references or references to predicates from a Datalog schema, there exists a corresponding Cyrano class. This in itself is not sufficient proof, for a user may create a Datalog program in which rules reference other predicates that he has defined and which therefore have no corresponding gateway class. For example, a user may generate defining rules for predicate **WORKER(3)**. The user may subsequently generate rules for **PROFESSIONAL(2)** that refer to **WORKER(3)**.

To handle this case, section 5.2.5 demonstrates that any collection of Datalog predicates can be reduced to a single predicate. In combination with the proof in this section, this shows that Cyrano fully supports Datalog as a member model.

5.2.1. A Translation Function T() for Rule-Based Models

A Datalog predicate P with arity N is structurally equivalent to a predicate in the relational model. Thus, $T_{Datalog}()$ is equal to $T_{Rel}()$ as described in section 5.1.1.

5.2.2. An Algorithm A1 for Rule-Based Models

Each Datalog predicate corresponds to a class, with each tuple that satisfies the predicate corresponding to an object in the class. As this is identical to the relational case, $A1_{Datalog}$ is equal to $A1_{Rel}$ as described in section 5.1.2.

5.2.3. An Algorithm A2 for Rule-Based Models

Given a Datalog predicate, $A2_{Datalog}$ creates an equivalent derived class containing one derivation corresponding to each rule for the predicate. It calls the function CREATE_DERIVATION() to create each derivation. Figure 5.4 contains $A2_{Datalog}$. Figure 5.5 contains CREATE_DERIVATION().

5.2.4. Proof of Correctness for T(), A1, A2

To run a Datalog query is to invoke a predicate and get the results. This section will prove that taking those results and converting them using $T_{Datalog}()$ results in the set of objects that are members of the predicate's class as created using $A2_{Datalog}$. This section first proves a lemma.

Lemma 5.2.1:

For any non-recursive Datalog rule R of the form:

$$P(V_1, ..., V_n) :- \quad P_1(V_{1,1}, ..., V_{1,n1}),$$
$$...,$$
$$P_m(V_{m,1}, ..., V_{m,nm})$$

And given:

$A2_{Datalog}$:

Input:

A predicate P and its defining rules

Output:

A Cyrano class definition C

Processing:

Create class C
DO for all rules R in P
Call CREATE_DERIVATION() to create a derivation corresponding to R
Add the derivation to C
ENDDO

Example:

Taking as input:

ANCESTOR(X, Y) :- PARENT(X, Y)
ANCESTOR(X,Y) :- ANCESTOR(X,Z), PARENT(Z, Y)

The output is:

CLASS ANCESTOR IS DERIVED WITH
OBJECT X; OBJECT Y; PARENT O_1:
($O_1.A_1$ = X AND $O_1.A_2$ = Y)
WITH
OBJECT A_1 IS X;
OBJECT A_2 IS Y;
END;
OBJECT X; OBJECT Y; OBJECT Z;
ANCESTOR O_1, PARENT O_2:
($O_1.A_1$ = X AND $O_1.A_2$ = Z AND
$O_2.A_1$ = Z AND $O_2.A_2$ = Y)
WITH
OBJECT A_1 IS X;
OBJECT A_2 IS Y;
END;
END CLASS.

Figure 5.4: Algorithm A2 for Datalog

CREATE_DERIVATION():

Input:
A Datalog rule R

Output:
A Cyrano derivation D

Processing:
Create empty derivation D

DO for all predicate references P_i in R
Add variable O_i of class P_i as a base variable to D
ENDDO

DO for all variables V_i appearing in R
Add variable V_i of class OBJECT as a base variable to D
ENDDO

Initialize C to TRUE
DO for all predicate references $P_i(X_1, ..., X_n)$ in D
Set C = C AND $P_i.A_1=X_1$ AND ... AND $P_i.A_n=X_n$
ENDDO
Set the guard of the derivation equal to C

Where the left-hand side of R is $P(X_1, ..., X_n)$, add the following to D:
OBJECT A_1 IS X_1; ..., OBJECT A_n IS X_n;

Example:
Taking as input:
P(X, Y) :- Parent(X, Y)
The output is:
OBJECT X; OBJECT Y; PARENT O_1:
($O_1.A_1$ = X AND $O_1.A_2$ = Y)
WITH
OBJECT A_1 IS X;
OBJECT A_2 IS Y;
END;

Figure 5.5: CREATE_DERIVATION() Algorithm

1) D, the Cyrano derivation resulting from applying CREATE_DERIVATION() to R,

2) O, the set of Cyrano objects derived by D;

3) V, the set of all n-tuples that satisfy P using rule R;

4) For each predicate $P_i(N_i)$, an equivalent Cyrano class C_i;

Then:

$$O = T_{Datalog}(V)$$

Or, in other words, CREATE_DERIVATION() creates a Cyrano derivation that is equivalent to the Datalog rule that it receives as input.

Lemma 5.2.1 can be proven by induction on the number of predicates referenced in the rule.

In the base case, where there are no predicates, the effect of a Datalog rule of the form:

$$P(X_1, ..., X_n).$$

is to tie the method values to the various X_i's. In this case, CREATE_DERIVATION() produces a clause of the form:

OBJECT V_1; ...; OBJECT V_m: TRUE
WITH
    OBJECT $A_1 = X_1$; ...; $A_n = X_n$;
END;

where each V_i is a value referenced in the set $\{X_i\}$. ($\{X_i\}$ may include constant and variable values.) In this case, the Cyrano derivation guard is always true, as is the Datalog rule. The attributes that are constant in the Datalog rule are held constant

in the derivation, and the attributes that are variables are allowed to be any value in the derivation. Therefore, the Cyrano derivation is equivalent to the Datalog rule.

For the inductive step, each additional invocation of a predicate in rule R is of the form:

$$P_i(X_1, ..., X_{ni})$$

For this, CREATE_DERIVATION() adds a base object O_i of class P_i to the Cyrano derivation rule. It also adds the following in conjunction to the guard:

$$O_i.A_1=X_1 \text{ AND } ... \text{ AND } O_i.A_{ni}=X_{ni}$$

This looks for an object O_i of class P_i with attribute values $<X_1, ..., X_{ni}>$. By the fourth given of lemma 5.2.1, class P_i is equivalent to predicate $P_i(n_i)$, and therefore the addition to the guard is equivalent to matching the predicate. Therefore the inductive step holds, and lemma 5.2.1 is true.

We now turn to Theorem 5.2:

Theorem 5.2:

For any:

1) Datalog schema D;

2) Datalog predicate P with rules that reference predicates in D or recursively reference P;

and given:

1) The set V of all N-tuples that satisfy P;

2) The set S of Cyrano gateway classes generated by $A1_{Datalog}$ applied to D;

3) C is the Cyrano class gotten by applying $A2_{Datalog}$ to P(N);

4) The set O of Cyrano objects that are members of C;

Then:

$$O = T_{Datalog}(V).$$

In other words, for a given predicate P(n), $A2_{Datalog}$ creates an equivalent Cyrano class.

When the Datalog predicate is not recursive, the proof of theorem 5.2 is fairly simple. First, note that a predicate is defined by the disjunction of its rules. Second, note that a Cyrano object is derived by a disjunction of its derivations. $A2_{Datalog}$ creates the derivation rules of a class by calling CREATE_RULE() for each rule and creating the Cyrano class that is the disjunction of the resulting derivations. Lemma 5.2.1 states that CREATE_RULE() creates valid Cyrano derivations for all non-recursive rules. Therefore, theorem 5.2 is true when P(n) is not recursive.

To prove the recursive case, we first prove a lemma. The lemma is based on the observation that any tuple that satisfies predicate P(N) satisfies it with a finite number of recursive calls to P(N). (If there are an infinite number of calls, P(N) does not terminate and therefore never recognizes the tuple.)

Lemma 5.2.2: Given:

1) The assumptions of Theorem 5.2 hold;

2) O is the set of objects discovered to be members of C within at most i recursive references of C;

3) V_i is the set of N-tuples discovered to have satisfied P(N) within at most i recursive calls to P(N);

Then:

$$\forall(i)(O_i = T(V_i))$$

This can be proven by induction on i.

For the base case, S_0 and V_0 are built without using the recursive rules or

derivations. The first part of the proof of theorem 5.2 shows that the non-recursive rules are equivalent to the non-recursive derivations of class C, and therefore that $S_0 = T(V_0)$.

For the inductive step, note that V_i is generated using some combination of non-recursive clauses and V_{i-1}. By the second given of theorem 5.2, there are equivalent Cyrano classes for the predicates referenced in the non-recursive clauses. By the inductive hypothesis, the known set S_{i-1} contains those Cyrano objects that are equivalent to V_{i-1}. As shown in the proof for lemma 5.2.1, when equivalent objects are known for predicates referenced in a Datalog rule, then the derivation created by CREATE_DERIVATION() for the rule generates equivalent objects to the tuples that satisfy the predicate. Therefore, the inductive step holds and lemma 5.2.2 is true.

Given lemma 5.2.2, and noting that, as i approaches infinity, $S_i = S$, $V_i = V$, and $S_i = T(V_i)$, we prove that $S = T(V)$. Thus, theorem 5.2 is true, and Cyrano fully supports any single Datalog predicate.

5.2.5. Reduction of a Datalog Program to a Single Predicate

Any set of Datalog predicate definitions can be reduced to definitions for a single predicate by the REDUCE_DATALOG procedure contained in figure 5.6.

In the example, P(N) is equivalent to all rules $P_i(N_i)$ in that $P(i, X_1, ..., X_{Ni}, 0, ..., 0)$ is true if and only if $P_i(X_1, ..., X_{Ni})$ is true. Therefore, any complete Datalog program can be reduced to rules for a single predicate, and the proof that there is an equivalent Cyrano class for any predicate also proves that there are equivalent classes for any Datalog program.

5.3. Object Oriented Data Models

The data models previously discussed in this chapter have standard

REDUCE_DATALOG:

Input:

R, a set of rules defining Datalog predicates $\{P_1(N_1), ..., P_m(N_m)\}$

Output:

A set of rules defining predicate P(N)

Processing:

Let $N = MAX(N_1, ..., N_m) + 1$

DO for all predicates P_i

Replace each reference to $P_i(X_1, ..., X_{Ni})$ in R

with a reference to $P(i, X_1, ..., X_{Ni}, 0, ..., 0)$

where (0,...,0) contains $N - (N_i + 1)$ zeroes

ENDDO

Example:

For the following rules:

PARENT(X, Y) :- MOTHER(X, Y)

PARENT(X, Y) :- FATHER(X, Y)

GRANDFATHER(X, Y) :- FATHER(X, Z), PARENT(Z,Y)

GRANDMOTHER(X, Y) :- MOTHER(X, Z), PARENT(Z,Y)

FAMILY(X, Y, Z) :- FATHER(Y, X), MOTHER(Z, X)

create the following rules:

P(1, X, Y, 0) :- MOTHER(X, Y)

P(1, X, Y, 0) :- FATHER(X, Y)

P(2, X, Y, 0) :- FATHER(X, Z), P(1, Z, Y, 0)

P(3, X, Y, 0) :- MOTHER(X, Z), P(1, Z, Y, 0)

P(4, X, Y, Z) :- FATHER(Y, X), MOTHER(Z, X)

and note that:

P(1, ...) is equivalent to PARENT(...)

P(2, ...) is equivalent to GRANDFATHER(...)

P(3, ...) is equivalent to GRANDMOTHER(...)

P(4, ...) is equivalent to FAMILY(...)

Figure 5.6: REDUCE_DATALOG Algorithm

representations that can be compared to Cyrano. This is not true of object oriented data models. There is little consensus on what is an object oriented data model: while there is a proposed standard [Cattell et al. 1994], it is not uniformly applied in existing object database systems and research.

This presents a problem when attempting to prove that Cyrano supports object oriented data models. One approach is to attempt to develop an inclusive definition of object oriented models, a task that is beyond the scope of this research. An alternate approach, and that taken herein, is to choose a sample object oriented query language and prove that Cyrano supports it.

The chosen model is based on one proposed by Kim for use in the Orion database [Kim 1990b]. Because it is meant for use by end-users, Kim's query language provides several syntactic shortcuts to achieve results that can be achieved in other ways. While these make the language easier to use, they do not extend its query range. In order to simplify the proof, this section proves support of a simplified version of Kim's query language that does not include these shortcuts. Section 5.3.5 describes and justifies these simplifications.

Kim's data model is a standard object oriented model. An object has attributes and methods. An attribute can have an atomic value or be a reference to another object. A method can have parameters and returns a value. The class of the object specifies the names and types of the attributes and methods of the object. A class can be a subclass to one or more other classes and inherits the attributes and methods from its parents.

In the simplified approach, a query takes the following form:

SELECT *variable* FROM *variable_list* WHERE *qualification*

where:

variable is one of the variables listed in the variable list.

variable_list lists the free variables that appear in the qualification section and their types.

qualification is a logical expression on the listed variables.

The query determines the values of the variables named in the variable list that make the qualification be true. The set of values for the variable named in the SELECT section is the result of the query.

The qualification contains an expression in which individual clauses can be joined by boolean operators. Each clause contains a simple expression comparing two terms. The terms are either values gotten from an object variable from the variable list or constants. Where an attribute or method points to another object, a term can indirectly access attributes and methods of the indicated object.

Note that the above query form is identical to the query form of an SQL database. The query language of Kim's model, like that of most object oriented databases, is a modified version of SQL. The primary difference in query form lies in the tests that can be performed as part of the qualification. In SQL, all data variables have either atomic data items or flat relations for their values. In Kim's model, a variable can refer to a complex object, and therefore the qualifier can refer to attributes of the variable's attributes. Further, the qualifier can invoke a method of one of its variables, which is not possible in an SQL database.

The use of a complex object is illustrated in the following sample query, which finds all companies with home offices in Virginia:

```
SELECT c
FROM c: Company
WHERE c.home-office.state = "Virginia"
```

The following finds all employees that live in the home state of their companies.

```
SELECT e
FROM e: Employee, c: Company
WHERE e.company = c
      AND c.home-office.state = e.address.state
```

5.3.1. A Translation Function T() for Object Oriented Models

A query in Kim's query language returns a collection of objects. Because these are objects that are stored in the database, each has a corresponding Cyrano gateway object. Each object in Kim's data model has a set of attributes and methods. The corresponding Cyrano gateway object handles a message for each attribute or message of the Kim object by accessing the Kim object.

Because of this, the translation function $T_{Kim}(O)$, when O is a Kim object, returns the corresponding Cyrano gateway object. In a sense, then, $T_{Kim}()$ is provided by the implementation of Cyrano's gateway to the Kim database.

In contrast, relational and rule based queries return relations that do not exist in the database. These relations must be translated into new Cyrano objects, requiring new translation functions. However, in the object oriented case the objects exist: all that is needed is to select the objects that satisfy specific criteria.

5.3.2. An Algorithm A1 for Kim's Model

A schema under Kim's model is a set of class definitions, each of which lists the superclasses of the class, its attributes, and its methods. This is equivalent to a Cyrano class definition. Therefore, $A1_{Kim}$ only has to adjust the class definition to match Cyrano's syntax. Figure 5.7 contains $A1_{Kim}$.

5.3.3. An Algorithm A2 for Kim's Model

As noted, a query in Kim's query language consists of three parts: a target variable, a variable list, and a qualification. The target variable indicates the values that serve as the result of the query. The variable list corresponds to the Cyrano base object list. The qualification corresponds to the Cyrano guard. Figure 5.8 contains $A2_{Kim}$, which uses these facts to generate the Cyrano class for a query.

$A1_{Kim}$:

Input:

A schema S, consisting of a set of classes specified using Kim's object oriented data model

Output:

A corresponding Cyrano gateway database

Processing:

DO for all classes C_{Kim} in schema S
Create a Cyrano class C_{Cyr}
DO for all parent classes P_{Kim} of C_{Kim}
Make P_{Cyr} a parent to C_{Cyr}
ENDDO
DO for all value attributes A of C_{Kim} where A is of class D_{Kim}
Add method A with arity 0 and type D_{Cyr} to C_{Cyr}
ENDDO
DO for all methods M of C_{Kim} with parameters $P_1...P_n$
and returning a value of class E_{Kim}
Add method M with arity n and type E_{Cyr} to C_{Cyr}
ENDDO
ENDDO

Example:

For a schema with definitions for:

Person
has Name: String;
Parent is a Person
has IsChild(P) : boolean;

Create the following Cyrano schema:

```
CLASS PERSON IS GATEWAY WITH
        STRING NAME;
END CLASS.

CLASS PARENT IS GATEWAY (IS PERSON) WITH
        BOOLEAN IsChild(1);
END CLASS.
```

Figure 5.7: Algorithm A1 for Kim's Object Oriented Data Model

$A2_{Kim}$:

Input:

A query with target variable T, variable list {T : C_0, V_1 : C_1, ..., V_n : C_n}, and qualification Q

Output:

A Cyrano derived class

Processing:

```
Create Cyrano class C
Create a derivation of class C
Add (C0 T) to the base variable list of the derivation
DO for i := 1 to n
        Add (Ci Vi) to the base variable list of the derivation
ENDDO
Use Q as the guard for the derivation
Add method (C0 RESULT IS T) to the derivation
```

Example:

Given the query:

```
SELECT e
FROM e: Employee, c: Company
WHERE e.company = c
        AND c.home-office.state = e.address.state
```

Produce the following Cyrano class definition:

```
CLASS C IS DERIVED WITH
        Employee e, Company c:
                e.company = c AND
                c.home-office.state = e.address.state
        WITH
                Employee RESULT IS e;
        END;
END CLASS.
```

Figure 5.8: Algorithm A2 for Kim's Object Oriented Data Model

5.3.4. Proof of Correctness for T(), A1, A2

Before proving support for Kim's model, we prove a lemma.

Lemma 5.3.

For any:

1) Set S of classes in Kim's model;

2) Class C chosen from S;

and given:

1) The Cyrano class D generated by $A1_{Kim}$ applied to S;

2) The set O of objects that are members of S;

3) The set P of objects that are members of D;

Then:

$$\forall(o \mid o \in O)\ \exists(p \mid p \in P)\ (p = T_{Kim}(o))$$

and:

$$\forall(p \mid p \in P)\ \exists(o \mid o \in O)\ (p = T_{Kim}(o))$$

In other words, the gateway database generated by $A1_{Kim}$ contains classes equivalent to the classes in its input database.

The truth of Lemma 5.3 is clear from the way that A1 constructs the database. Cyrano's implementation of the object oriented gateway class ensures that D_i contains a corresponding object for each object of class C_i.

Theorem 5.3.

For any:

1) Set S of classes in Kim's model;

2) Query Q against S;

and given:

1) The Cyrano database D constructed by applying $A1_{Kim}$ to

S;

2) The derived class C constructed by $A2_{Kim}$ applied to Q;

3) The set K of objects returned by Q;

4) The set O of objects that are members of C;

Then:

$$\forall(k \mid k \in K)\ \exists(o \mid o \in O)\ o = T_{Kim}(k)$$

and:

$$\forall(o \mid o \in O)\ \exists(k \mid k \in K)\ o = T_{Kim}(k)$$

In other words, the members of class C are equivalent to the response of query Q, and therefore $A2_{Kim}$ is correct.

Assume that Q has N variables in its variable list. Thus, the possible objects that can instantiate the variables form a set T of N-tuples. By Lemma 5.3, there is a set U of N-tuples of Cyrano objects equivalent to T. Because the semantics of Cyrano's query language are the same as those of Kim's language, the guard of D will be true for that subset V of U that corresponds to the subset W of T that causes Q to be true. Q returns the projection of W on some specific variable. By the construction of $A2_{Kim}$, D returns the projection of V on the equivalent variable. Thus, D contains the equivalent objects to those returned by Q. Theorem 5.3 is true, and Cyrano fully supports Kim's object oriented data model.

5.3.5. Other Features of Kim's Model

This section describes the simplifications made to Kim's object oriented data model as used in this discussion. The descriptions of the simplifications include brief explanations of how the features can be supported in Cyrano. The following is a list of these simplifications.

1) Kim allows the results of two or more queries to be joined using

set operators. For example, a query could consist of:

QUERY1 ∩ QUERY2

Union and set difference are also supported.

The processing of these set operations can be performed by a query construct of the form QUALIFIER1 OPERATOR QUALIFIER2, where OPERATOR is AND for intersection, OR for union, and AND NOT for set difference. Queries that return heterogeneous sets of objects can be simulated in Cyrano through heterogeneous classes that join classes representing the base queries. For example, "QUERY1 ∩ QUERY2" can be represented by a Cyrano derived class that returns those objects in both the corresponding QUERY1 and QUERY2 derived classes.

2) Kim allows the values of variables to be partially qualified in the variable list. This is equivalent to placing those rules in the qualification.

3) Kim allows queries to return some subset of an object's attributes. Cyrano can do this by only specifying some subset of methods for the derived class.

4) Kim allows a variable to represent an object from some combination of classes. Cyrano can do this by representing the class combination as a separate class formed by combining the base classes.

5) Kim allows a class to be automatically expanded into all of its super- or sub-classes. This can be done explicitly in Cyrano by listing all super- and sub-classes.

6) Kim allows attributes to be sets of objects and provides operators to test whether any or all members of the set match certain criteria. Set attributes can be replaced by predicates with one parameter. The tests can be replaced by formulae using existential and universal quantifiers. This provides a relational view of the sets, an equivalent representation.

For example, a query phrase:

"ALL company employees salary > 10000"

which tests that all employees of the company have salary greater than 10000, can be replaced with the phrase:

"FORALL (EMPLOYEE E) (NOT Company.Employees(E)) OR (E.Salary > 10000)"

7) Kim supports a RECURSE keyword in a query. The RECURSE keyword causes a recursive expansion of an attribute list. For example, a query that finds all ancestors of "Andy" can be:

SELECT P FROM P: PERSON, ANDY: PERSON WHERE

ANDY name = "Andy" AND P = ANDY RECURSE(parent)

The phrase "ANDY RECURSE(parent)" expands to "ANDY parent", "ANDY parent parent", "ANDY parent parent parent", ...

Cyrano can represent a recursive phrase by a separate recursive derived class. The query's derived class can reference the recursive derived class.

Chapter 6: Cyrano's Support for Database Integration

This chapter demonstrates that Cyrano provides sufficient features to federate heterogeneous databases. By demonstrating that Cyrano provides such support, this chapter shows that Cyrano satisfies the second of the three requirements for meta models, the ability to combine gateway databases into a global database.

This demonstration is broken into demonstrations that Cyrano can overcome the types of heterogeneity that arise between gateway databases as described in sections 2.1.2 and 2.1.3.

6.1. Cyrano and Data Heterogeneity

As described in section 2.1.2, data heterogeneity includes the ways in which data values can differ across databases. Table 6.1 duplicates table 2.2, which describes the different types of data heterogeneity. Section 2.1.2 describes these classifications in greater detail.

This section shows that Cyrano can support the database designer in resolving the types of data heterogeneity. Section 6.1.1 describes how Cyrano supports the resolution of value inconsistencies by allowing the operator to establish rules for determining which data to use. Section 6.1.2 describes how Cyrano supports the resolution of representational inconsistencies by allowing the specification of rules for normalizing data into a single representation.

Taken together, these sections show that Cyrano provides sufficient functionality to resolve data heterogeneity.

6.1.1. Cyrano and Value Inconsistencies

Value inconsistencies arise when different databases contain different values for similar data. Figure 6.1 gives a simple example in which this occurs.

Table 6.1: Data Heterogeneity

Value Inconsistencies
Differences in data values between databases.

Missing Data
One database is missing data that another contains.

Obsolete Data
One database contains obsolete data.

Incorrect Data
One database contains incorrect data.

Representational Inconsistencies
Differences in representation of similar data.

Data Types Differences
Use of different data types to represent similar data.

Unit Differences
Use of different units for similar data.

Precision Differences
Different precision used for similar data.

Representation Differences
Different enumerated representations of similar data.

```
CLASS A_Salary IS GATEWAY WITH
        STRING Name;
        STRING DateUpdated;
        INT Salary;
END CLASS.

Data values:
        {"Andy", "95-09-01", 520},
        {"Kate", "95-09-01", 364},
        {"Diana", "95-09-01", 260}
        {"Julie", "95-09-01", 7280}

CLASS B_Salary IS GATEWAY WITH
        STRING Name;
        STRING DateUpdated;
        INT Salary;
END CLASS.

Data values:
        {"Andy", "95-09-01", 520},
        {"Kate", "94-09-01", 312},
        {"Diana", "95-09-01", 822}
        {"Joe", "95-09-01", 5200}


CLASS Salary IS DERIVED WITH
        A_Salary A:
                        FORALL(B_Salary B)
                                A.Name <> B.Name OR A.DateUpdated >= B.DateUpdated
                STRING Name IS A.Name;
                STRING DateUpdated IS A.DateUpdated;
                INT Salary IS A.Salary;
        END;
        B_Salary B:
                        FORALL(A_Salary A)
                                A.Name <> B.Name
                                OR B.DateUpdated > A.DateUpdated
                                OR (B.DateUpdated = A.DateUpdated
                                        AND B.Salary <> A.Salary)
                STRING Name IS B.Name;
                STRING DateUpdated IS B.DateUpdated;
                INT Salary IS B.Salary;
        END;
END CLASS.

Data values:
        {"Andy", "95-09-01", 520},
        {"Kate", "95-09-01", 364},
        {"Diana", "95-09-01", 260}
        {"Julie", "95-09-01", 7280}
        {"Diana", "95-09-01", 822}
        {"Joe", "95-09-01", 5200}
```

Figure 6.1: Cyrano and Value Inconsistencies

Figure 6.1 also contains derived classes that resolve the value inconsistencies.

In the example, two classes present different views of the salaries of different workers. These classes also specify the date when the values were last updated. Sample data values for the classes show the various types of value differences. These include:

1) Missing data. **A_Salary** is missing a value for Joe's salary. **B_Salary** is missing a value for Julie's salary.

2) Obsolete data. **B_Salary** has an obsolete value for Kate's salary.

3) Incorrect data. Either **A_Salary** or **B_Salary** has an incorrect value for Diana's salary.

In the case of Diana's salary, the example shows that it is not necessarily clear which of two databases contains correct data. Although the two databases contain different values for Diana's salary, it is not clear which is correct.

Figure 6.1 gives a derived class **Salary** that resolves the value inconsistencies between the two classes. In particular, **Salary** resolves the different types of value inconsistencies by:

1) Resolving missing data differences by presenting values for all persons identified in either class. Therefore, both Joe and Julie show up in **Salary**'s data values.

2) Resolving obsolete data differences by presenting only the most recently updated data for any person. Therefore, only Kate's most recent salary appears in **Salary**'s data values.

3) Resolving incorrect data values by presenting all mismatched data values. Therefore, both of Diana's salaries appear in the **Salary** data. In this case, it is left to the user to determine which value is correct.

Although this example is fairly simple, it shows that Cyrano provides tools

to overcome value inconsistencies. To some extent, however, value inconsistencies are difficult to overcome: because they result from errors in the use of databases, they do not follow any consistent rules. This example demonstrates that Cyrano provides the database administrator with sufficient tools to implement various strategies for resolving these differences.

6.1.2. Cyrano and Representational Inconsistencies

Representational inconsistencies arise when different databases use different representations for similar data elements. Figure 6.2 contains a simple example where this occurs. Figure 6.2 also contains a derived class that overcomes these inconsistencies.

In the examples, two classes represent weather measurements taken at different locations. **National_Weather** describes measurements taken at National Airport. **Dulles_Weather** contains measurements taken at Dulles. These classes include the following types of representational inconsistencies:

1) Data type differences. **National_Weather** uses an integer to represent humidity, while **Dulles_Weather** uses a real number.

2) Unit differences. **National_Weather** uses Fahrenheit to represent temperature, while **Dulles_Weather** uses degrees Celsius.

3) Precision differences. **National_Weather** contains humidity rounded to the nearest five percent, while **Dulles_Weather** contains the unrounded data.

4) Representation differences. **National_Weather** uses a 0-10 scale for air quality, while **Dulles_Weather** uses an A-F scale.

Figure 6.2 contains a derived class **TotalWeather** that resolves these inconsistencies. In particular, **TotalWeather** presents all measurements in a normalized format. **TotalWeather** normalizes the data by:

```
CLASS National_Weather        /* weather at National Airport */
IS GATEWAY WITH
        INT Date;                       /* days since 1/1/90 */
        INT Temperature;                /* degrees Fahrenheit */
        INT Humidity;           /* percentage to nearest 5% */
        INT AirQuality;         /* 0 - 10 */
END CLASS.

CLASS Dulles_Weather                    /* weather measured at Dulles */
IS GATEWAY WITH
        INT Date;                       /* days since 1/1/90 */
        INT Temperature;                /* degrees Celsius */
        REAL Humidity;                  /* percentage, no rounding */
        CHAR AirQuality;                /* A - F */
END CLASS.


CLASS TotalWeather            /* weather in both locations */
IS DERIVED WITH
        National_Weather W : TRUE
                STRING Location IS "National";
                INT Date IS W.Date;
                INT Temperature IS (W.Temperature - 32) * 5 / 9;
                                        /* normalize to Celsius */
                REAL Humidity IS W.Humidity.Real;
                                        /* convert type to REAL */
                REAL HumidityPrecision IS 1.0;
                INT AirQuality IS W.AirQuality;
        END;
        Dulles_Weather M : TRUE
                STRING Location IS "Dulles";
                INT Date IS M.Date;
                INT Temperature IS M.Temperature;
                REAL Humidity IS M.Humidity;
                REAL HumidityPrecision IS 5.0;
                INT AirQuality IS IF(M.AirQuality = "A" THEN 10;
                                                M.AirQuality = "B" THEN 7;
                                                M.AirQuality = "C" THEN 5;
                                                M.AirQuality = "D" THEN 3;
                                                M.AirQuality = "F" THEN 0);
        END;
END CLASS.
```

Figure 6.2: Cyrano and Representational Inconsistencies

1) Resolving type differences by converting the integer humidity into a real number, using the Cyrano built-in type translation methods.

2) Resolving unit differences by converting the Fahrenheit temperature into Celsius.

3) Resolving the precision difference by explicitly representing the precision of each humidity measurement. This makes the precision available to the user.

4) Resolving representation differences by translating the air quality measurements to the 0-10 scale. This requires some interpretation of how the A-F scale translates.

This example shows that Cyrano provides sufficient tools to overcome representational inconsistencies. Therefore, Cyrano can provide the tools to overcome both types of data heterogeneity.

6.2. Cyrano and Structural Heterogeneity

As described in section 2.1.3, structural heterogeneity includes the different ways in which different databases represent similar information. Table 6.2 duplicates table 2.3, which shows the different types of structural heterogeneity. Section 2.1.3 describes these types of heterogeneity in greater detail.

This section shows that Cyrano can support the database designer in resolving the types of data heterogeneity. Section 6.2.1 describes how Cyrano supports the resolution of naming differences. Section 6.2.2 describes how Cyrano supports the resolution of Constraint differences. Section 6.2.3 describes how Cyrano supports the resolution of data grouping differences. Section 6.2.4 describes how Cyrano supports the resolution of inheritance differences.

Taken together, these sections show that Cyrano provides sufficient capabilities for resolving structural heterogeneity.

Table 6.2: Structural Heterogeneity

Naming Differences
Different names used for similar data or same name used for different data.

Structure Name Differences
Differences in structure names.

Attribute Name Differences
Differences in attribute names.

Constraint Differences
Different constraints lead to different data domains.

Data Grouping Differences
Differences in grouping of data.

Missing Attribute Differences
A structure does not have all the attributes of a similar structure.

Multiple Structure Differences
Differences in the number of structures used to represent similar information.

Multiple Attribute Differences
Differences in the number of attributes used to represent similar information.

Structure vs Attribute Differences
A structure is used to represent information represented by attributes in another database.

Inheritance Differences
Differences in the class-subclass hierarchy of object oriented databases.

6.2.1. Cyrano and Naming Differences

Naming differences occur when different names are used to represent similar data or the same names are used for different data. Figure 6.3 gives a simple example in which this occurs. Figure 6.3 also shows derived classes that resolve the differences.

In the example, two databases are represented, **A** and **B**. Each contains two classes each. (The database name is prepended to the names of its classes.) These classes include the following kinds of naming differences:

1) Two types of structure name differences arise. **A_People** is the name of **A**'s class for persons, while **B_Persons** is the name of **B**'s class for persons. Thus, the databases use different names for the people structures. Further, although they have the same names, **A_Pro** refers to professionals, while **B_Pro** refers to professions. Thus, the databases use the same name for different structures.

2) Two types of attribute name differences arise. **A_People** uses **Name** to represent the person's name, while **B_Persons** uses **Label** for the name. Further, **A_People** uses **Address** to describe the person's home address, while **B_Persons** uses **Address** to contain the title used to address the person (e.g., Mr, Ms, Dr).

Figure 6.3 gives three derived classes that resolve the naming differences. In particular, these resolve the differences by:

1) Resolving structure name differences by deriving structures from the appropriate names. Thus, **Person** derives from both **A_People** and **B_Persons**, **Professional** derives from **A_Pro**, and **Profession** derives from **B_Pro**.

2) Resolving attribute name differences by deriving attributes to

```
CLASS A_People
IS GATEWAY WITH
        STRING Name;                    /* person's name */
        STRING Address;                 /* home address */
END CLASS.

CLASS A_Pro                             /* class of professionals */
IS GATEWAY WITH
        STRING Name;                    /* name of professional */
        STRING Profession;              /* name of his profession */
END CLASS.

CLASS B_Persons
IS GATEWAY WITH
        STRING Label;                   /* person's name */
        STRING Address;                 /* person's title: e.g., Mr, Ms */
END CLASS.

CLASS B_Pro                             /* class of professions */
IS GATEWAY WITH
        STRING ProfessionName;          /* name of the profession */
END CLASS.



CLASS Person
IS DERIVED WITH
        A_People P : TRUE
                STRING Name IS P.Name;
                STRING Address IS P.Address;
                STRING Title IS "";
        END;
        B_Persons P : TRUE
                STRING Name IS P.Label;
                STRING Address IS "";
                STRING Title IS P.Address;
        END;
END.

CLASS Professional
IS DERIVED WITH
        A_Pro P : TRUE
                STRING Name IS P.Name;
                STRING Profession IS P.Profession;
        END;
END CLASS.

CLASS Profession
IS DERIVED WITH
        B_Pro P : TRUE
                STRING Name IS P.ProfessionName;
        END;
END CLASS.
```

Figure 6.3: Cyrano and Naming Differences

normalize the names. Thus, **Person.Name** derives from **A_People.Name** and **B_Persons.Label**, **Person.Address** derives from **A_People.Address**, and **Person.Title** derives from **B_Persons.Address**.

These examples show that Cyrano provides sufficient capabilities to resolve naming differences.

6.2.2. Cyrano and Constraint Differences

Constraint differences arise when different constraints apply to different member database structures that represent similar data items. Resolving constraint differences differs from resolving other types of differences in that these differences are not represented by gateway classes but instead will be reflected in the data that is accessible via those classes. There are two primary ways to resolve these differences:

1) A global database user may want to ignore the differences in constraints and view all included data through the same derived classes. The constraints would not be obeyed by all data at the global database level, but all gateway database data would be accessible at that level.

2) A global database user may want to maintain the constraints, dropping data from gateway databases if that data does not satisfy the constraints. The constraints would then be reflected in all data at the global database level, but some data from gateway classes may not be accessible at the global level.

Cyrano classes support both of these strategies. In particular:

1) Constraint differences can be ignored by using Cyrano to combine classes as it would combine any other gateway classes.

2) Constraints can be maintained at the global level by writing those

constraints into the derivation rules of global classes.

6.2.3. Cyrano and Data Grouping Differences

Data grouping differences arise when different databases use structurally different structures to represent similar data items. These are among the most complicated types of database heterogeneity. In particular, multiple structure differences, in which different structures can be used to represent similar information, can be extremely complicated.

Because of this, this section is broken into two sub-sections. Section 6.2.3.1 describes Cyrano's handling of multiple structure differences. Section 6.2.3.2 describes Cyrano's handling of all other data grouping differences.

6.2.3.1. Cyrano and Multiple Structure Differences

Multiple structure differences arise when two databases use different arrangements of structures to represent similar information. These can be fairly complex, involving several incompatible structures used by different databases. Figure 6.4 shows one simple example of such a difference. The figure also shows derived classes that resolve the heterogeneity.

In the example, two databases each contain records about employees. One database breaks the employees into clerical and production employees. The other database breaks employees into workers and managers. These class breakdowns do not correspond to each other: workers and managers can both be either clerical or production employees.

The solution in figure 6.4 uses the Worker/Manager break-down of employees. It represents clerical and production workers through the use of attributes. This separation is fairly simple using Cyrano's rule-based features.

With more complex multiple structure differences, the most complicated step is likely to be the selection of a normalized format. Often, databases differ in their

```
CLASS CoClericalWorker IS GATEWAY WITH
        STRING Name;
        BOOLEAN IsManager;
END CLASS.

CLASS CoProductionWorker IS GATEWAY WITH
        STRING Name;
        BOOLEAN IsManager;
END CLASS.

CLASS UnWorker IS GATEWAY WITH
        STRING Name;
        BOOLEAN IsClerical;
END CLASS.

CLASS UnManager IS GATEWAY WITH
        STRING Name;
        BOOLEAN IsClerical;
END CLASS.


CLASS Manager IS DERIVED WITH
        UnManager M : TRUE
                STRING Name IS M.Name;
                BOOLEAN IsClerical IS M.IsClerical;
        END;
        CoClerical C : C.IsManager
                STRING Name IS C.Name;
                BOOLEAN IsClerical IS TRUE;
        END;
        CoProduction P : P.IsManager
                STRING Name IS P.Name;
                BOOLEAN IsClerical IS FALSE;
        END;
END CLASS.

CLASS Worker IS DERIVED WITH
        UnWorker W : TRUE
                STRING Name IS W.Name;
                BOOLEAN IsClerical IS W.IsClerical;
        END;
        CoClerical C : NOT C.IsManager
                STRING Name IS C.Name;
                BOOLEAN IsClerical IS TRUE;
        END;
        CoProduction P : NOT P.IsManager
                STRING Name IS P.Name;
                BOOLEAN IsClerical IS FALSE;
        END;
END CLASS.
```

Figure 6.4: Cyrano and Multiple Structure Differences

structures due to their focus. In the example, the database separating employees into clerical and production pools is for use by a company, while the database separating them into workers and management is used by a union. These reflect the different needs of the two groups of database users. The selection of a normalized format will depend on the needs of the federation users.

Often, those needs may lead to a desire to present more than one set of structures. Cyrano handles this by allowing multiple derived classes to be derived from the same gateway classes. Therefore, multiple structuring approaches can be supported by a single Cyrano global database.

Once the strategy is selected, Cyrano's rules can be used to recombine structures as required.

6.2.3.2. Cyrano and Different Data Grouping Differences

Other data grouping differences arise when there are differences in individual structures. Figure 6.5 shows samples of these differences. It also contains derived classes that resolve the differences.

These structures include two sets of derived classes to represent persons. These include the following differences:

1) Missing attribute differences. **A_Person** has an **Age** attribute, which is missing in **B_Person**.

2) Multiple attribute differences. **A_Person** uses a single attribute to represent name, while **B_Person** uses two attributes, **FirstName** and **LastName**.

3) Structure vs attribute differences. **A_Person** uses an **A_Address** attribute to represent address, while **B_Person** uses a flat string attribute.

The derived class **Person** resolves these differences by:

1) Resolving missing attribute differences by leaving out the attribute

```
CLASS A_Person IS GATEWAY WITH
        STRING Name;
        INT Age;
        A_Address Address;
END CLASS.

CLASS A_Address IS GATEWAY WITH
        INT Number;
        STRING Street;
        STRING City;
        STRING State;
        INT Zip;
END CLASS.

CLASS B_Person IS GATEWAY WITH
        STRING LastName;
        STRING FirstName;
        STRING Address;
END CLASS.



CLASS Person IS Derived WITH
        A_Person P : TRUE
                STRING Name IS A.Name;
                INT Age IS P.Age;
                STRING Address IS P.Address.Number.STRING +
                                P.Address.Street + P.Address.City +
                                P.Address.State + P.Address.Zip.STRING;
        END;
        B_Person P : TRUE
                STRING Name IS P.FirstName + P.LastName;
                STRING Address IS P.Address;
        END;
END CLASS.
```

Figure 6.5: Cyrano and Other Data Grouping Differences

when there is no value. Alternately, **Person** could have assigned a default value to the attribute or left out the attribute entirely.

2) Resolving multiple attribute differences by joining the **FirstName** and **LastName** attributes into a single **Name** attribute. Alternately, **Person** could have split **A_Person**'s **Name** attribute.

3) Resolving structure vs attribute differences by flattening the **A_Address** class into a string attribute. Alternately, **Person** could have constructed an **Address** class from **B_Person**'s **Address** attribute.

Although these are simple examples, they show how Cyrano could be used to modify the database structures to unite them. Thus, Cyrano can handle data grouping differences.

6.2.4. Cyrano and Inheritance Differences

Inheritance differences arise when similar classes in different databases inherit from different parent classes. This results in the subclasses inheriting differences from the parent classes. These differences can be from any of the forms of heterogeneity described in this chapter.

As shown in this chapter, Cyrano can handle any of these forms of heterogeneity. Cyrano can also handle this heterogeneity when it is inherited from parent classes. Therefore, Cyrano can handle inheritance differences.

Chapter 7: Roxanne: A Proof-of-Concept for Cyrano

This chapter demonstrates that Cyrano can be used by a functional FDBS. At the heart of this demonstration is Roxanne, a proof-of-concept FDBS that uses Cyrano as a meta model. Roxanne provides a complete implementation of Cyrano, with gateways to the Paradox relational DBMS and a home-grown Datalog DBMS. Roxanne demonstrates that Cyrano can be used as meta model by a functional FDBS.

In addition to this chapter, several appendices deal with Roxanne. Appendix B describes the Booch method used to design Roxanne. Appendix C contains a description of Roxanne's design. Appendix D contains the source code for Roxanne.

7.1. Roxanne's Functional Requirements

Roxanne is a prototype federated database system based on the Cyrano meta model. Roxanne satisfies five primary functional requirements. These are:

1) Roxanne fully implements the Cyrano meta model as described in chapter 3.

2) Roxanne supports a relational database and a Datalog database as member databases. These two databases are sufficient to show that Cyrano and Roxanne provide gateways to member databases. Relational and Datalog databases were selected for ease of implementation and availability of sample database management systems.

3) Roxanne provides sufficient built-in classes to represent the atomic data types available in the supported member databases. These include integers, booleans, and strings. Support for these data types include implementation of standard simple operations on these types (e.g., addition for integers).

4) Roxanne allows the user to create and edit gateway and derived classes by creating class specifications using the Cyrano language.

5) On user request, Roxanne finds and displays all objects that are members of the specified gateway or derived class.

By allowing users to specify new derived classes and find all of their member objects, Roxanne supports queries against a set of databases. Therefore, Roxanne provides a complete implementation of Cyrano querying capabilities.

7.2. A Description of Roxanne

Roxanne supports the Cyrano meta model by allowing users to create, list, edit, and instantiate new gateway and derived classes. This section describes how Roxanne performs these functions.

This section is broken into three sub-sections. Section 7.2.1 describes the development and operational environments of Roxanne. As a way of showing what Roxanne does, section 7.2.2 describes its user interface in some detail. Finally, section 7.2.3 describes how Roxanne performs two processing threads, the specification of a new Cyrano class and the finding of members of a class.

7.2.1. Roxanne's Development and Operational Environments

Roxanne was designed on a Sun SparcStation LX operating under SunOS (version 4.1.3) using the Rational Rose for Unix object oriented CASE tool (version 2.0.15). Rational Rose supports the Booch notation for object-oriented analysis and design.

Roxanne was implemented on an Intel 486 based microcomputer using the Borland C++ development environment (version 2.1).

Roxanne's target production environment is a microcomputer using the Intel 386, 486, or Pentium processor and running Microsoft DOS (4.0 or better) and

Microsoft Windows (3.1 or better).

Roxanne accesses Borland's Paradox relational database as a member database. As an interface to Paradox, Roxanne uses the Borland Paradox Engine Database Framework (version 3.0).

Roxanne accesses a Datalog database as a member database. This Datalog database was developed as part of this research, with its rule-processing engine based on that of Roxanne. This database provides those Datalog features required by Roxanne: the initialization of the database with a static set of rules, and the invocation of a rule to find the tuples that satisfy it.

Roxanne uses Borland's Object Windows Library to support its user interface. Object Windows Library provides access to the windowing features of Microsoft Windows. It is part of the Borland C++ development package.

7.2.2. Roxanne's User Interface

Roxanne's interface centers around one primary window, the list of active classes. Figure 7.1 shows that window. It consists of a panel listing the active classes (i.e., the gateway and derived classes known to Roxanne) and a set of buttons indicating processing. A user selects a class and depresses a button to invoke processing against that class. The buttons are as follows:

1) The **New** button allows the user to create a new class. In response, Roxanne displays an edit screen in which the user can enter the class specification.

2) The **Run** button causes Roxanne to run the selected class, finding all objects that are members of the class.

3) The **Edit** button allows the user to edit the selected class, bringing up an edit screen containing the class specification.

4) The **Delete** button deletes the selected class.

5) The **Save** button allows the user to save the selected class. The

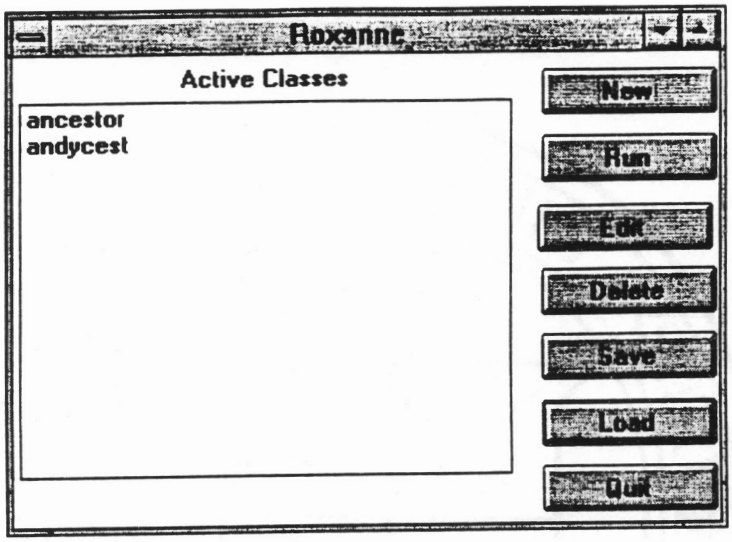

Figure 7.1: Roxanne's Class List Window

class is saved to the file "**CLASS**.cls" in Roxanne's data directory, where **CLASS** is the name of the class.

6) The **Load** button allows the user to load a previously saved class. A file selector window pops up, allowing the user to select a class file. Once selected, Roxanne displays an edit window containing the class, allowing the user to make any desired modifications.

7) The **Quit** button allows the user to quit Roxanne.

When the user selects either the **New**, **Edit**, or **Load** buttons, Roxanne displays an edit window. Figure 7.2 shows this window. The edit window provides a simple text editor to edit the text of the Cyrano class specification. When editing an existing class, Roxanne pre-loads the specification of the class into the window. Otherwise, the user must enter the specification.

Upon completing his edit, the user selects the "Save,Exit" button. Roxanne pops up a dialog box asking the user if he wishes to continue. If so, Roxanne compiles the class and adds it to the list of active classes. If there is a syntax error in the class specification, Roxanne brings up an error window and allows the user to re-edit the specification. Figure 7.3 shows the error window.

Upon selecting the **Run** button, Roxanne runs the selected class, finding all objects that are members of the class. Roxanne displays those objects in the **Results** window. In a **Results** window, Roxanne displays the contents of all objects found to be members of the class.

To display a gateway object (which, for both the Paradox and Datalog gateways, is a relation), Roxanne displays the names and values of all attributes of the relation. Figure 7.4 shows a **Results** window listing members of a gateway class.

To display a derived object, Roxanne displays the list of base objects from

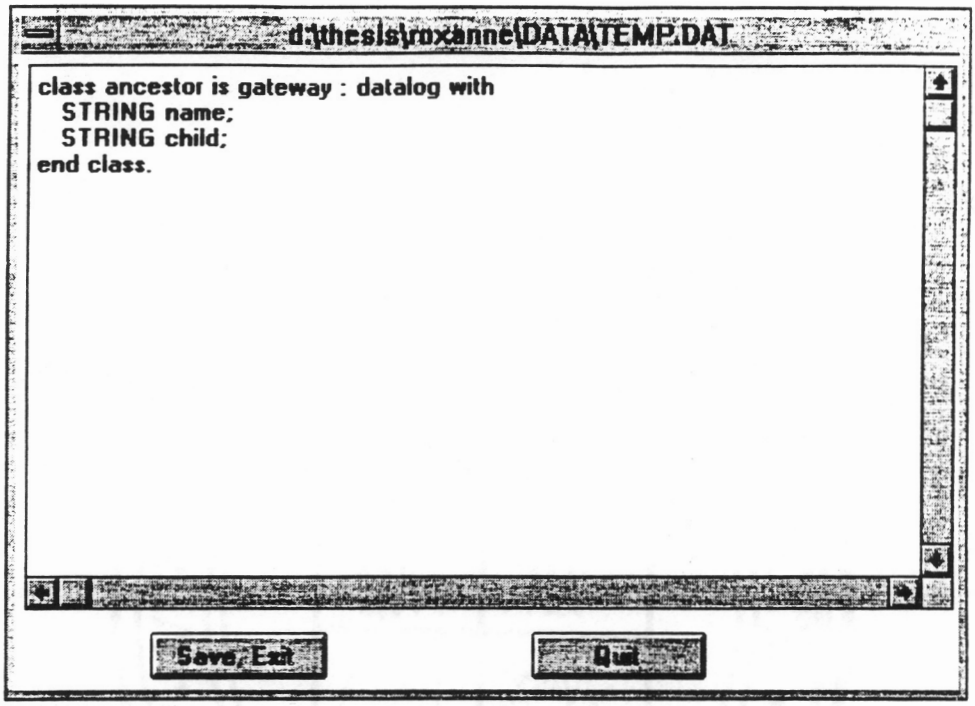

Figure 7.2: Roxanne's Edit Window

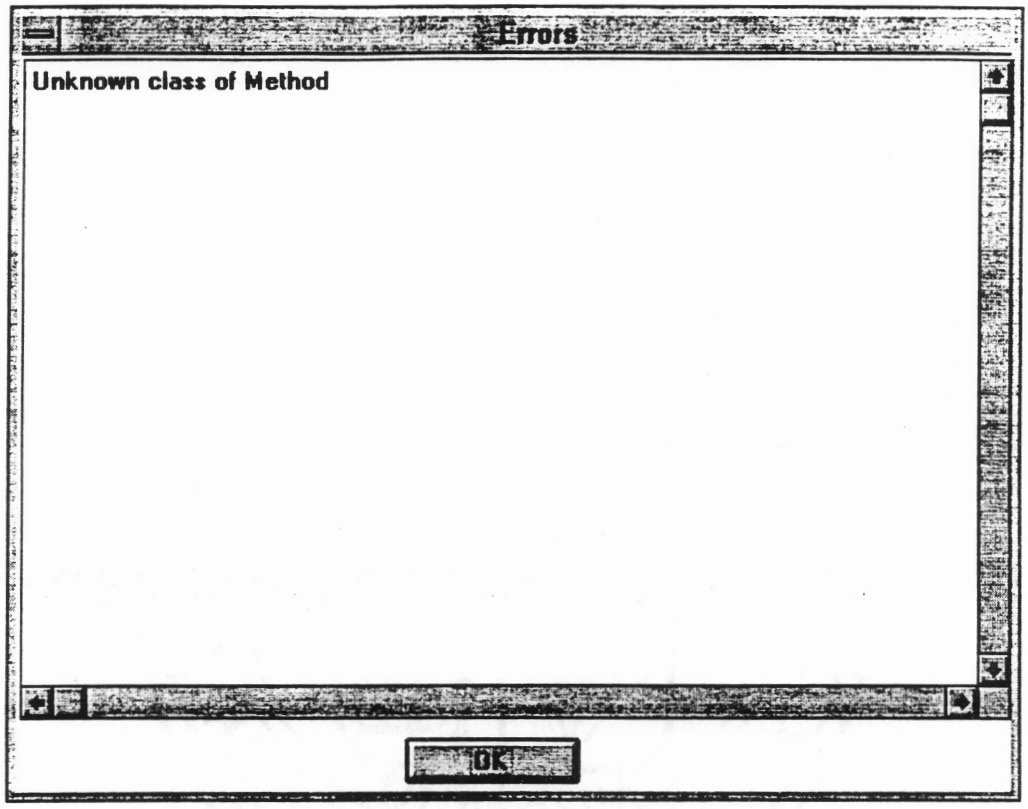

Figure 7.3: Roxanne's Error Window

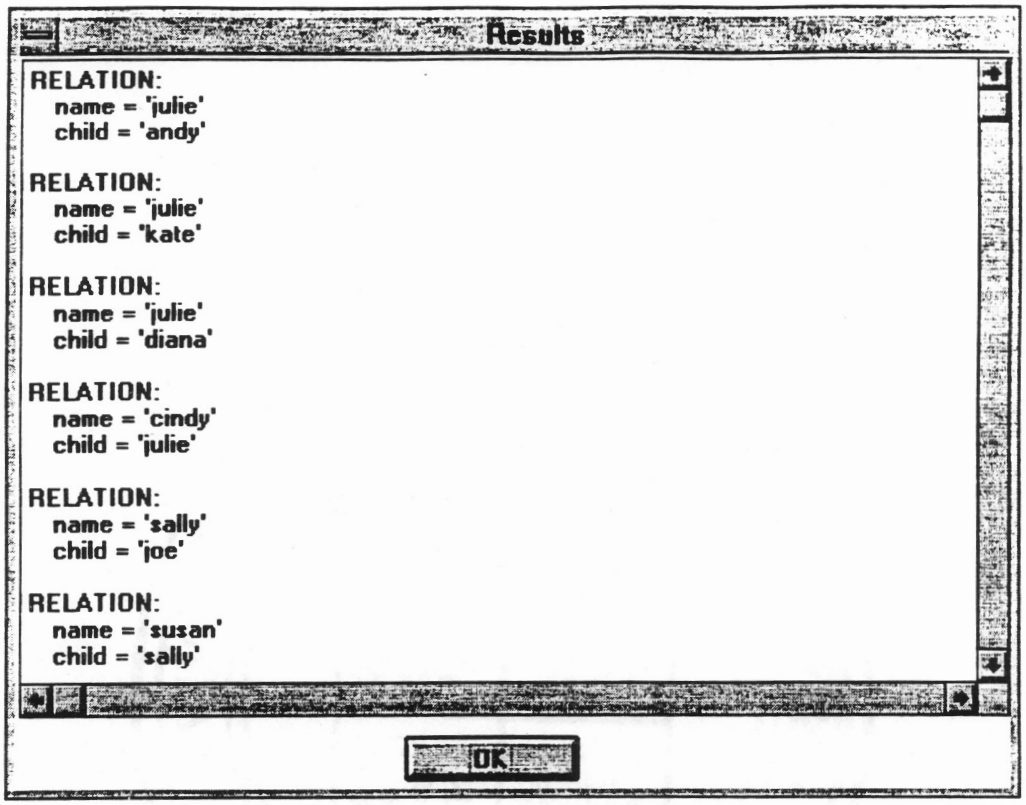

Figure 7.4: Roxanne's Gateway Class Results Window

which the object was derived. This is a recursive procedure: if a base object is itself a derived object, Roxanne displays its base objects, continuing until all nested base objects are displayed. Figure 7.5 shows a **Results** window listing members of a derived class.

These windows complete the displays shown by Roxanne. Although there are not many windows, and these rather simple, they allow the user to specify new classes and find their members, thus satisfying Roxanne's requirements.

7.2.3. Roxanne's Implementation of Cyrano

Roxanne implements two primary Cyrano functions. When a user submits a new Cyrano class, Roxanne compiles the class specification into an internal representation. When a user requests the members of a class, Roxanne uses the class rules to generate its member objects.

When a user enters or modifies a class specification, Roxanne compiles the class into an internal class representation. This representation is a nested structure of C++ objects, each of which represents an element of the class specification. Separate C++ objects represent the class, its derivations, and its methods, with different C++ classes used to represent the different Cyrano class types (derived, gateway, and built-in).

In compiling a class, Roxanne uses an LL(1) parser to parse the class specifications. (For more details on LL(1) parsers, see Barrett et al. [1986].) In keeping with an object-oriented approach, the parser is part of the C++ classes that represent the Cyrano class elements. Each such C++ class has a constructor that takes as input a portion of the specification filtered through a lexical analyzer. This differs from traditional parser/compilers primarily in that the objects parse their own descriptions, invoking each other's parsers as necessary.

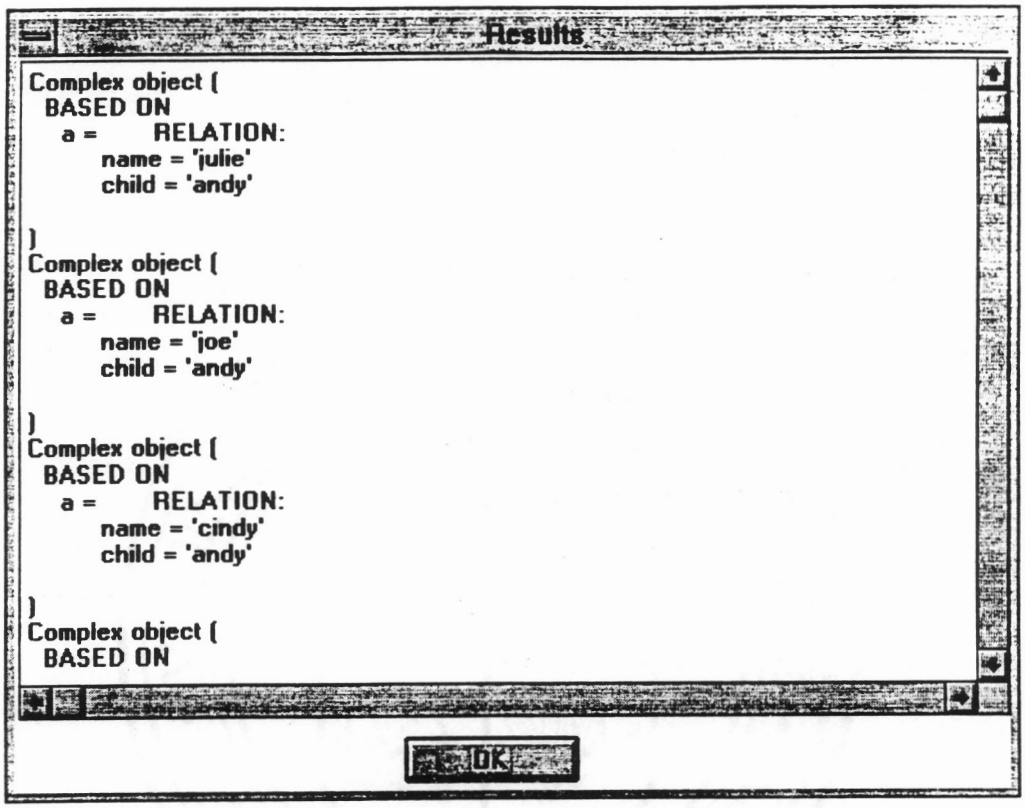

Figure 7.5: Roxanne's Derived Class Results Window

The listing of members of a class is Roxanne's central operation. This processing differs based on the type of Cyrano class. Built-in classes cannot be instantiated. Gateway classes find member objects by retrieving them from the corresponding member database. Derived classes find member Cyrano objects by running the rules associated with their derivations.

Roxanne instantiates a Cyrano gateway class by interacting with the member database management system to retrieve all members of the associated data structure. For each member data item, Roxanne constructs an equivalent Cyrano object. Roxanne uses a relation object to represent each row in a Paradox table or tuple satisfying a Datalog predicate.

Roxanne uses the algorithm shown in figure 7.6 to instantiate derived classes. This algorithm does a bottom-up instantiation of the Cyrano class by repeatedly cycling through all base classes (and their base classes, etc) and, in each cycle, instantiating all classes whose base classes were instantiated in previous cycles. The basic gateway classes are instantiated in the first cycle, classes derived from the gateway classes are instantiated in the next cycle, and so on.

Recursive classes are handled by allowing the instantiation of a Cyrano class to be spread across several cycles. In the first cycle for a recursive class, the members derived from non-recursive derivations are found. Each subsequent cycle recurses once. When there are no more Cyrano objects to be derived from recursive derivations, the cycles end.

In a production-quality Cyrano implementation, this algorithm should be replaced with one that generates the most complex sub-queries possible against the member databases. This would shift query processing to the member database, taking advantage of any optimization done by its DBMS. It would also reduce the amount of data transmitted by the member database to Roxanne, thus

```
Create a Results List to contain and process the results
Add to it the target class (the class to be instantiated)
Add to it all Cyrano classes used to derive the target class
        (This includes superclasses, base classes of the target's derivations, and classes
        used to derive those base classes.)
Tell the Results List to instantiate all classes
Get the members of the target class from the Results List
```

7.6.a: Processing for Class Instantiation.

```
Set DONE = FALSE
DO until DONE
        Set DONE = TRUE
        DO for all classes on the Results List
                IF this is a gateway class
                        IF this is first time in the loop for this class
                                Instantiate the class
                                Add its members to the Results List
                                Set DONE = FALSE
                        ENDIF
                ELSE
                        DO for all derivations of the class
                                Find all possible combinations of base Cyrano
                                        objects of the class built from
                                        objects in the Results List known
                                        to be members of its base classes
                                DO for all such combinations containing at
                                                least one object added in the
                                                previous iteration of the
                                                "DO until DONE" loop
                                        IF the combination satisfies the
                                                        derivation's guard
                                                Make a derived object from those base
                                                        objects
                                                Add it to the Results List as a
                                                        member of this class
                                                Set DONE = FALSE
                                        ENDIF
                                ENDDO
                        ENDDO
                ENDIF
        ENDDO
ENDDO
```

7.6.b. Processing by Results List for Class Instantiation.

Figure 7.6: Class Instantiation Processing

saving bandwidth to remote member databases.

However, given that Roxanne currently does not access remote databases, and given that it is meant as a proof-of-concept for Cyrano and not as a production system, the current algorithm is sufficient.

7.3. A Roxanne User Session

This section describes a user's session with Roxanne, showing the windows that a user sees. In this user session, the user loads a gateway class from file, finds all objects that are members of it, creates a derived class based on the gateway class, and finds all objects that are members of the derived class. In this session, the user performs the following steps:

1) The user starts Roxanne. Roxanne displays an empty class window (figure 7.7).

2) The user selects the **Load** button. Roxanne displays a file selection box (figure 7.8).

3) The user selects the file **ancestor.cls** and selects the **OK** button. Roxanne displays an edit window containing the text of the selected class (figure 7.9).

4) The user selects the **Save, Exit** button. Roxanne displays a pop-up window asking if the user wants to continue (figure 7.10).

5) The user selects **Yes**. Roxanne displays the class list window, now containing the **ancestor** class (figure 7.11).

6) The user selects the **ancestor** class and selects **Run**. Roxanne displays a results window listing the relations in the **ancestor** class (figure 7.12).

7) The user selects **OK**. The class list window reappears.

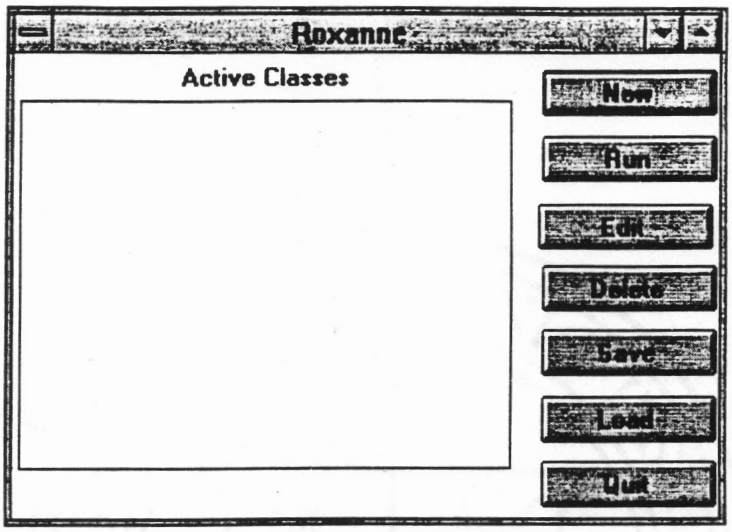

Figure 7.7: Empty Class List Window

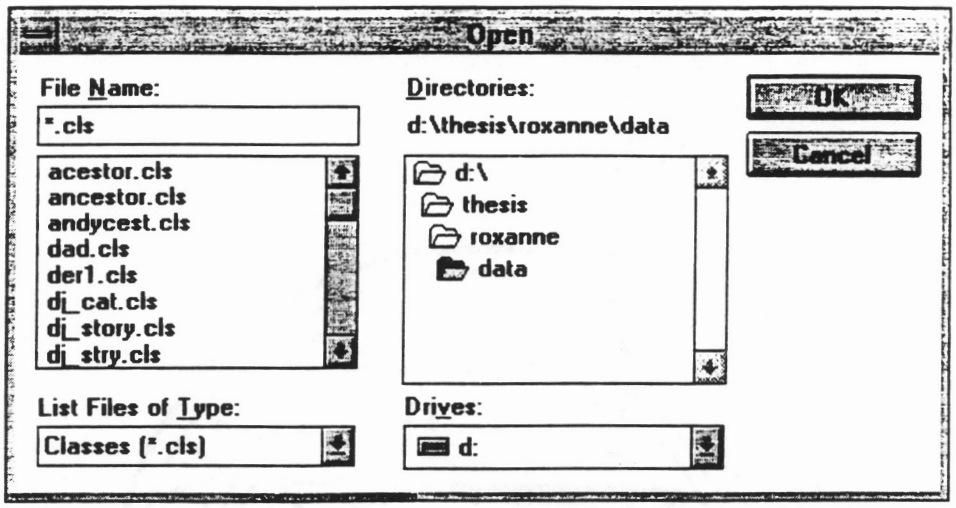

Figure 7.8: File Selection Window

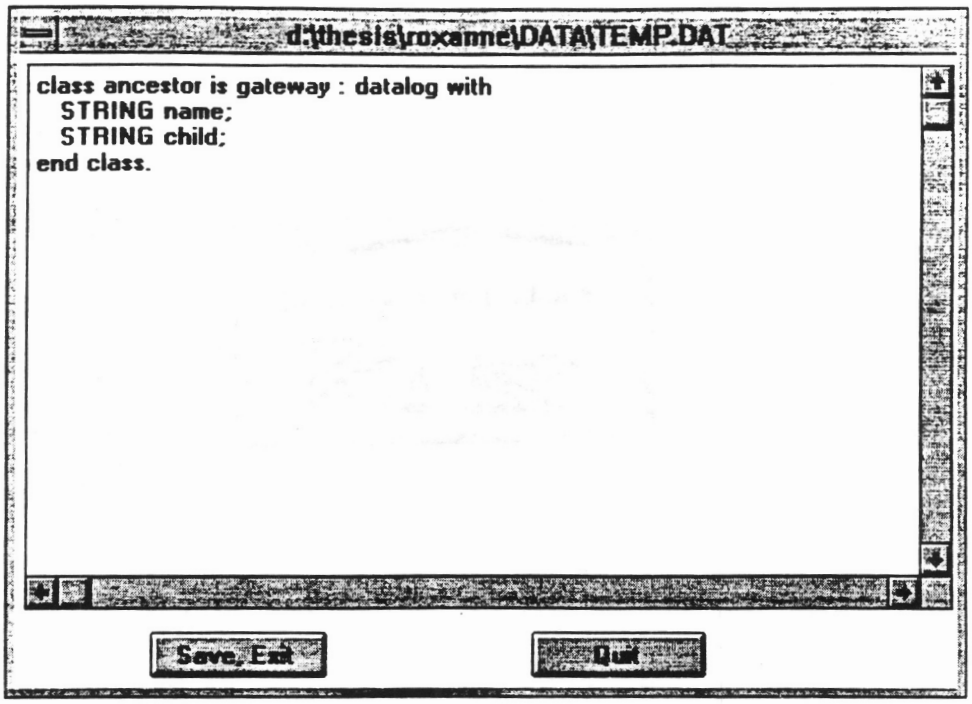

Figure 7.9: Class Edit Window with Ancestor Class

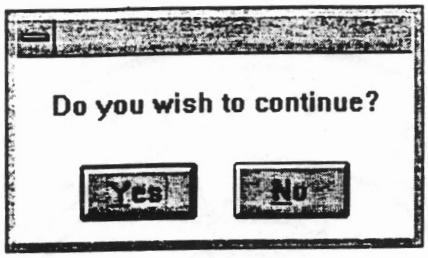

Figure 7.10: Continue Query Pop-Up

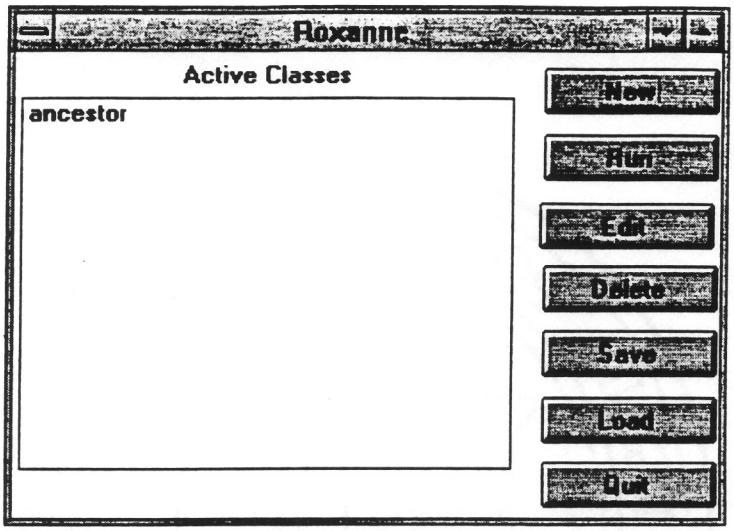

Figure 7.11: Class List Window with Ancestor Class

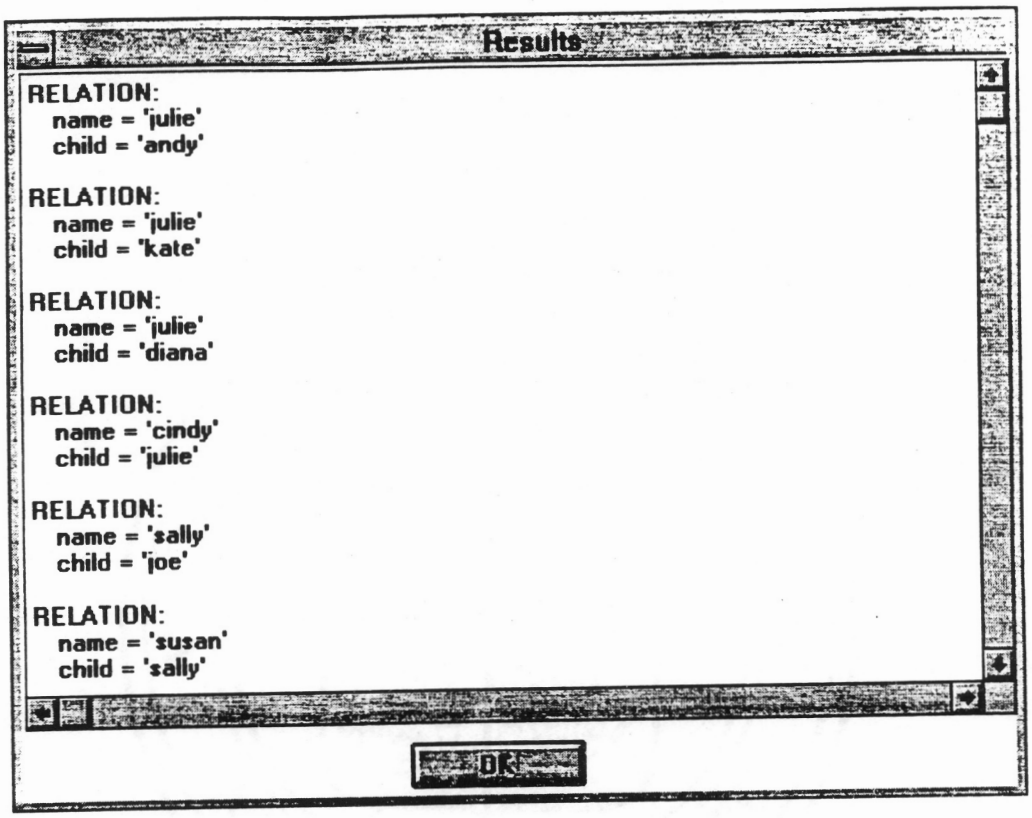

Figure 7.12: Results Window with Ancestor Results

8) The user selects the **New** button. Roxanne displays an empty editor window (figure 7.13).

9) The user enters the text of the new **andycest** class (figure 7.14).

10) The user selects **Save, Exit**. Roxanne displays a pop-up window asking if the user wants to continue.

11) The user selects **Yes**. Roxanne displays the class list window, now containing both the **ancestor** and the **andycest** classes (figure 7.15).

12) The user selects the **andycest** class and selects **Run**. Roxanne displays a results window listing the derived objects in the **andycest** class (figure 7.16).

13) The user exits Roxanne by selecting **OK** in the results window and **Quit** from the class list window.

7.4. Lessons of the Roxanne Proof-of-Concept

This section describes the insights gained through experience with Roxanne. The primary insight has to do with the feasibility of implementing Cyrano as a meta model. Additional insights indicate directions to take in future implementations of Cyrano.

Roxanne demonstrates its fundamental point: that it is possible to implement Cyrano as the meta model for an FDBS. Roxanne demonstrates this by supporting two different member models, as well as providing a complete implementation of Cyrano's data manipulations. Thus, Roxanne serves as a proof-of-concept for Cyrano in particular and object oriented/rule based hybrid meta models in general.

Roxanne in its current state has four primary limitations:

1) It supports too few member data models.

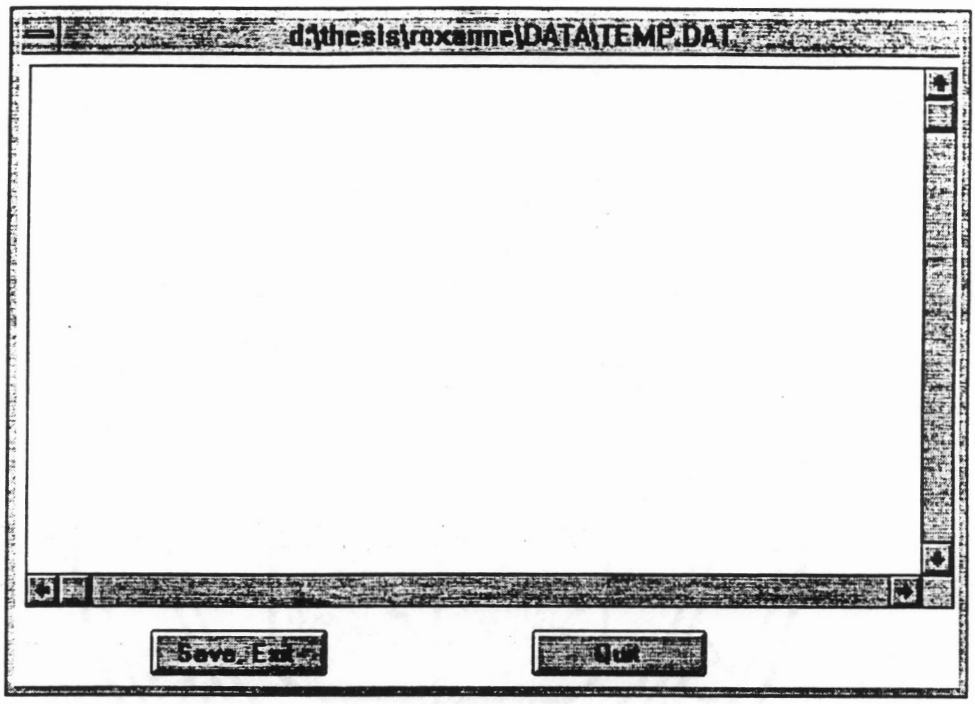

Figure 7.13: Empty Class Edit Window

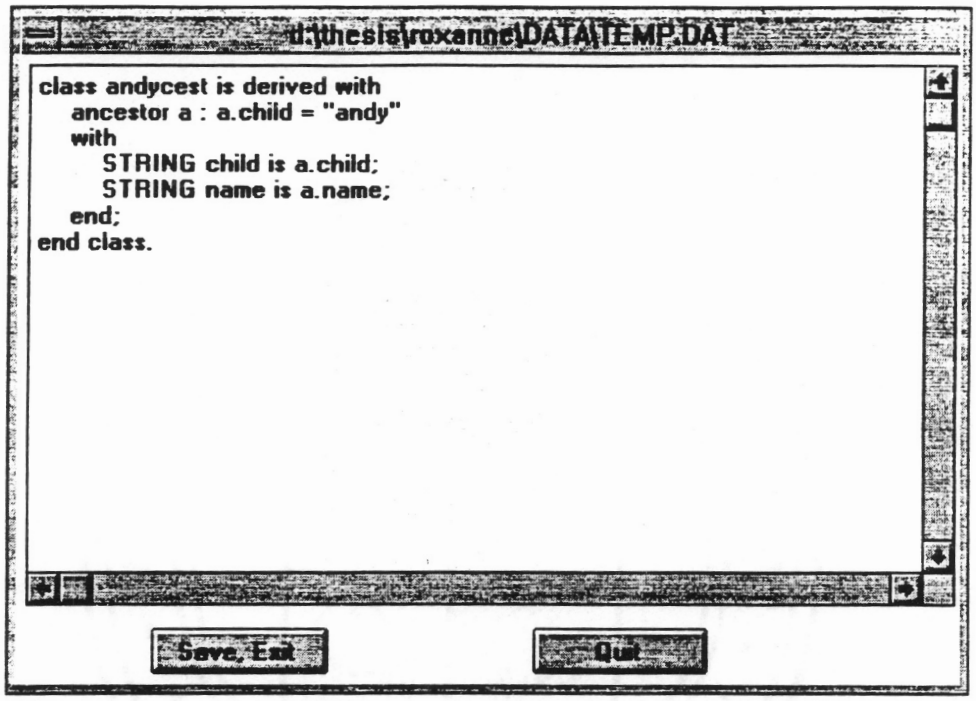

Figure 7.14: Class Edit Window with Andycest Class

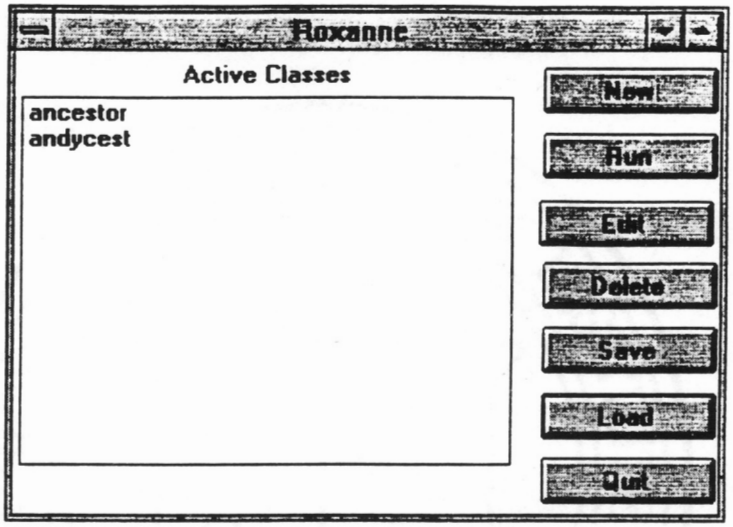

Figure 7.15: Class List Window with Ancestor and Andycest

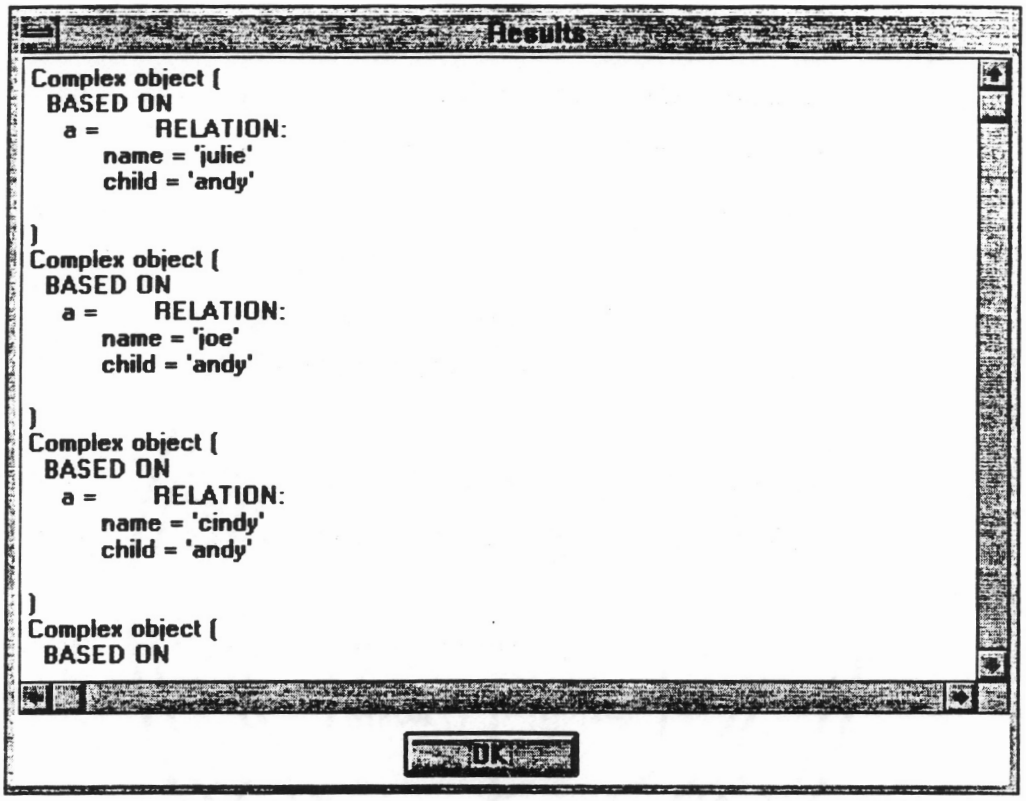

Figure 7.16: Results Window with Andycest Results

2) Its user interface is too awkward for production use.

3) Its algorithms are too slow for production, especially for use in a networked environment.

4) It provides insufficient structures to manipulate class groups.

However, most of these limitations can be easily overcome. For example:

1) It is fairly simple to add interfaces to additional DBMS's. By encapsulating interfaces in specific C++ classes, Roxanne allows easy addition of new member models.

2) The user interface can be easily replaced or modified. In addition, Cyrano's user-unfriendly query language could be supplemented with a more friendly front-end.

3) The Roxanne algorithms demonstrate that Cyrano queries can be executed. These could be improved through use of existing research into query optimization.

4) A production Cyrano environment needs some separation of classes into databases, allowing support for security restrictions and encapsulation of unnecessary information.

Chapter 8: Conclusions

The goal of this dissertation is to contribute to the evolution of meta models for federated database systems (FDBS). In particular, this research addresses the issue of a meta model for an FDBS including relational, object oriented, and rule based data models as member models. This expands on existing work that addresses federations of relational, network, and hierarchical databases.

Insights gained during this research resulted in four primary research contributions:

1) Existing reference architectures are inadequate to represent federations of object oriented and rule based member models. Therefore, this research introduces an object oriented reference architecture for FDBS's.

2) Examination of object oriented and rule based data models indicate that a hybrid of these models would make an effective meta model. Cyrano, a meta model designed for this research, is such a hybrid.

3) There are no established criteria for the evaluation of meta models. Thus, this research introduces a set of evaluation criteria for the purpose of evaluating Cyrano.

4) One of the developed criteria requires a demonstration FDBS using the proposed meta model. To satisfy this criteria, this research presents Roxanne, a proof-of-concept FDBS using Cyrano as meta model.

This chapter discusses these contributions in greater detail.

8.1. The FDBS Reference Architecture

A common first step in examining FDBS's is the development of a reference architecture. Such a reference architecture provides a framework for the

discussion of FDBS characteristics. Several such architectures exist in the FDBS literature [e.g., Sheth and Larson 1990; Deen et al. 1985]. These describe the different schema layers required to support the data translations performed by an FDBS. As such, they emphasize differences between databases in data structuring mechanisms.

This approach is sufficient when dealing with traditional data models. The primary differences between relational, network, and hierarchical systems relate to the ways that they structure data. Schema architectures are sufficient to capture these differences.

However, examination of these architectures in light of the requirement to support the newer member models reveals a flaw. By emphasizing data structure, they give short shrift to differences in data transactions that result from query languages with differing capabilities.

This becomes clear on examining rule based member models. There is no real difference between the data structuring capabilities of rule based and relational data models. Instead, rule based models differ primarily in the types of queries that they support.

In order to address this problem, the first major contribution of this research is the development of a new reference architecture. As presented in section 2.2.2, the proposed reference architecture provides an object oriented view of FDBS's, emphasizing the interfaces between databases instead of the way that a given database structures data.

The object oriented approach provides a strong basis for such a reference architecture. Object orientation, in describing a class of objects, first identifies the interfaces to those objects. The actual implementation of an object is encapsulated within it. In database terms, object orientation emphasizes transactions between objects instead of the structure of their data. This is appropriate when the primary differences between databases are in their interfaces

(which, in the case of a database, corresponds to its supported transactions and query language).

A result of the use of this architecture is an emphasis on interface definitions. Thus, Cyrano, which is based on this architecture, has clear definitions of the interfaces to any particular data object, and the data objects are primarily defined by their supported interfaces.

8.2. Cyrano, a Hybrid Meta Model

In examining the target member models, two things become clear:

1) Of the target member models, object oriented models provide the broadest data structuring capabilities. Object oriented models support complex objects, built-in methods, class inheritance, and a number of other data structuring features that make them far more capable then other models to represent complex data relationships. While other data structure types can be easily represented by objects, objects cannot always be easily represented using other models.

2) Of the target member models, rule based models provide the greatest querying and dynamic data manipulation capabilities. Rule based models support logic-based query languages that in many cases approach Turing completeness. Further, rule based models allow data predicates to map to either stored data or complex queries, providing deductive data structures that blur the line between data and query.

Based on extensive investigations of these observations, this dissertation concludes that a hybrid of object oriented and rule based data models makes an appropriate meta model for a federation of relational, object oriented, and rule based databases. Such a hybrid provides both the complex data structuring of object oriented models and the broad query languages of rule based models. Rule

based features also support the data translations and normalizations required of an FDBS that integrates heterogeneous databases.

Furthermore, such a hybrid allows for an emphasis on analysis over planning. Traditional object oriented systems provide a framework for planning tasks. A designer specifies classes as templates for objects that are yet to be created. Thus, a hammer class is created with the expectation that subsequently there will be a need to drive a nail.

In contrast, examination of federated databases is an analytic task. There already exists a world of objects: what is required is a mechanism for identifying the hammers among them. For this, a different type of object orientation is required. Such an analytic object orientation includes aspects of rule based systems: instead of giving a template for a hammer, an analytic class provides rules which can be used to identify hammers. Thus, analytic object orientation provides users of an FDBS with a framework for data analysis that appropriately reflects their needs and expectations.

Following on these insights, this research presents Cyrano, a hybrid of object oriented and rule based data models based on analytic object orientation. Cyrano is a meta model suitable for use in an FDBS that supports relational, object oriented, and rule based member models. Cyrano combines the broad data structuring capabilities of object oriented models with the query and dynamic data organization capabilities of rule based models.

Cyrano's primary strength is its support for the core functions of a meta model. Cyrano provides full support for its target member models by allowing the representation of any database built under those models to be fully represented by an equivalent Cyrano database. Further, Cyrano's query language is as broad as those of any of its member models: any query against a member database can be translated into an equivalent Cyrano query. Also, Cyrano provides broad functions for the integration of member databases: these features allow the Cyrano

designer to build mechanisms for overcoming the various types of structural and value heterogeneity that arise between different databases.

Currently, Cyrano is limited in that it does not provide a framework for representing entire databases. The only Cyrano mechanism for organizing data is the class. It would be convenient to have a mechanism to group classes in a database. This could be provided by a "Database" mechanism that combines classes. Class access and visibility could be maintained at the database level.

Further, Cyrano's current data and query language is not user friendly. The Cyrano data and query language provides a sophisticated set of options for a skilled user. However, writing a Cyrano query is a complex process. Cyrano would benefit from the addition of a user-friendly front-end language.

8.3. Criteria for the Evaluation of Meta Models

Given the design of Cyrano, it became necessary to determine whether it met the research goals. Such a determination requires a set of evaluation criteria by which Cyrano can be judged. Because there are no such criteria available in the literature, Cyrano's evaluation required the development of a set of criteria for judging meta models. These criteria are the third major contribution of this research.

The criteria proceed from the desire to support the major functions of a meta model within an FDBS. The primary purpose of a meta model is to support the retrieval and integration of data from a set of heterogeneous member databases within a functional FDBS. This leads to the following three criteria, expressed in terminology developed as part of the reference architecture:

1) A meta model must fully support the translation from member database to gateway database. This research defines full support as the ability to use the meta model to retrieve any information that could be retrieved using the member model. In other words, a user does not

sacrifice any data access through use of the meta model.

2) A meta model must fully support the federation of gateway databases into a global database. In other words, a meta model must provide sufficient capabilities to integrate data from member databases.

3) It must be possible to implement the meta model as part of an FDBS. Because a meta model is meant to be part of an FDBS, it must be possible to implement a prototype and demonstrate its operational characteristics for a sample set of member databases.

To evaluate the first of these criteria, this research proposes a formal process by which a meta model can be proven to support a given member model. This process involves providing algorithms for the translation of data across data model boundaries, and proving that these algorithms are valid. This produces a formal proof that the meta model supports the member model. This research uses this process to prove that Cyrano fully supports relational, object oriented, and rule based data models as member models.

To evaluate the second criterion, this research proposes a method of demonstrating that a meta model provides sufficient capabilities to integrate heterogeneous data. This involves providing a demonstration of how the meta model overcomes a sample set of differences that can arise between databases normalized into a single meta model. Although this method does not provide a formal proof of federation support, it provides strong evidence that sufficient support is provided. This research uses this method to demonstrate that Cyrano provides sufficient capabilities to integrate data.

To evaluate the third of these criteria, this research proposes the development of a proof-of-concept FDBS using the Cyrano meta model. Ultimately, the strongest proof that a meta model can be implemented is to develop an operational prototype. This research develops such a prototype called

Roxanne, a proof-of-concept FDBS using Cyrano as the meta model.

In future research, it would be desirable to develop a more formal method of evaluating the second criterion. Unlike the first criterion, there is no existing method for formally proving that a meta model provides sufficient capabilities for database integration. And unlike the third criterion, there is no concrete demonstration that a meta model successfully integrates all types of database heterogeneity.

However, such a formal method of evaluation requires a formal definition of the types of database heterogeneity. At this time, there is no such definition. Instead, discussions of this issue catalog different types of heterogeneity without providing an underlying theory or any proof of completeness [e.g., Kim and Seo 1991]. Given the lack of such formal definitions, informal methods such as those identified in this research are the best that can be done for now.

8.4. Roxanne, a Cyrano Proof-of-Concept

In the course of evaluating Cyrano against the third criterion, this research develops Roxanne, a proof-of-concept FDBS using Cyrano as a meta model. Roxanne provides a complete implementation of the Cyrano language in an FDBS that provides access to relational and rule based member databases.

Roxanne allows the construction of Cyrano gateways and derived classes. The gateway classes provide gateways to relational tables from a Paradox database or rule based tables from a Datalog database. The derived classes integrate gateway data using any of Cyrano's data manipulation features. Therefore, Roxanne exercises Cyrano's full range of features.

Roxanne is not a production-quality FDBS. In particular, its user interface is rudimentary, it provides no support for data security or distributed databases, and it currently supports only a limited number of member databases. Most important, it uses brute-force algorithms to evaluate Cyrano queries. These

algorithms result in poor performance in evaluating complex queries.

However, Roxanne does provide a tool for experimenting with Cyrano. In particular, Roxanne is used to confirm that Cyrano fully supports its member data models and provides sufficient capability for integrating heterogeneous databases.

8.5. Future Research Directions

This section identifies areas of future research indicated by the present work. These areas are related to the primary research contributions of this work.

Cyrano could be enhanced to support meta-data about data objects. Meta-data is data that describes the data in heterogeneous databases. It has been proposed as a tool for resolving semantic heterogeneity in databases. Cyrano could be expanded to provide mechanisms for representing such data.

As noted above, Cyrano could be enhanced to support the combination of classes into databases. Further, Roxanne could be used to perform experiments on Cyrano with the goal of providing experimental proof for Cyrano's suitability as a meta model. Finally, research could be done into expanding Cyrano to support other database types, including text and hyper-media databases.

Should a formal definition of database heterogeneity emerge, the evaluation criteria could be enhanced to provide a formal proof of support of database integration. Short of that, general-purpose procedures for evaluating a meta model against the criteria could be developed. Using those procedures, the criteria could be applied to existing meta models.

Roxanne has room for many enhancements, including a superior user interface and better query processing performance. Work is currently underway to optimize Cyrano queries within Roxanne. Finally, Roxanne serves as the basis for future research into Cyrano, a proposed meta model for an FDBS supporting relational, object oriented, and rule based data model members.

References

ACM, Special issue on federated database systems, ACM Computing Surveys, Vol 22, No 3, September, 1990.

ACM Special Interest Group on Management of Data, "Special Issue: Semantic Issues in Multidatabase Systems", SIGMOD Record, vol 20, no 4, December, 1991.

M. Adiba, D. Portal, "A Cooperative System for Heterogeneous Data Base Management Systems", Information Systems, vol 3, 1978, p. 209-215.

R. Ahmed, P. De Smedt, W. Du, W. Kent, M. Ketabchi, W. Litwin, A. Rafii, M. Shan, "The Pegasus Heterogeneous Multidatabase System", Computer, Vol 24, No 12, December 1991, p. 19-27.

F. Bancilhon, R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing", Proceedings of ACM SIGMOD 86 International Conference on Management of Data, 1986, p. 16-52.

W. Barrett, R. Bates, D. Gustafson, J. Couch, Compiler Construction: Theory and Practice, Science Research Associates, Inc, Chicago, Ill, 1986.

C. Batini, M. Lenzerini, S. Navathe, "A Comparative Analysis of Methodologies for Database Schema Integration", ACM Computing Surveys, vol 18, no 4, Dec. 1986, p. 323-364.

G. Booch, Object Oriented Design with Applications, The Benjamin/Cummings Publishing Company, Inc, Redwood City, California, 1991.

Y. Breitbart, P. Olson, G. Thompson, "Database Integration in a Distributed Heterogeneous Database System", Proceedings of the International Conference on Data Engineering, 1986, p. 301-310.

O. Buneman, S. Davidson, A. Watters, "Querying Independent Databases", Information Sciences, Vol 52, No 1, October, 1990, p. 1-34.

A. Cardenas & M. Pirahesh, "Data Base Communication in a Heterogeneous Data Base Management System Network", Information Systems, vol 5, p. 55-79, 1980.

A. Cardenas, "Heterogeneous Distributed Database Management: The HD-DBMS",

Proceedings of the IEEE, vol 75, no 5, May 1987, p. 588-600.

R. Cattell, ed., T. Atwood, J. Duhl, G. Ferran, M. Loomis, D. Wade, The Object Database Standard: ODMG-93, Morgan Kaufmann Publishers, San Francisco, California, 1994.

A. Chan, D. Ries, "Distributed Database Management Research at Computer Corporation of America", Database Engineering, vol 5, no 4, December 1982, p. 250-255.

W. Chen, D. Warren, "Objects as Intensions", Logic Programming: Proceedings of the Fifth International Conference and Symposium, vol 1, MIT Press, Cambridge, Massachusetts, 1988.

D. Chorafas, H. Steinmann, Solutions for Networked Databases, Academic Press, Inc., San Diego, 1993.

C. Chung, "DATAPLEX: An Access to Heterogeneous Distributed Databases", Communications of the ACM, vol 33, no 1, January 1990, p. 70-80.

E. F. Codd, "A Relational Model of Data for Large Shared Data Banks", Communications of the ACM, vol 13, no 6, June, 1970, p. 377-387.

E. F. Codd, "Extending the Database Relational Model to Capture More Meaning", Transactions on Database Systems, vol 4, no 4, December, 1979, p. 397-434.

E.F. Codd, "Data Models in Database Management", Proceedings of the Workshop on Data Abstraction, Databases, and Conceptual Modelling, June 23-26, 1980, published in SIGART Newsletter, no 7, ACM, New York, New York, January, 1981.

C. Collet, M. Huhns, W. Shen, "Resource Integration Using a Large Knowledge Base in Carnot", Computer, vol 24, no 12, December, 1991, p. 55-62.

U. Dayal, P. A. Bernstein, "On the Correct Translation of Update Operations on Relational Views", ACM Transactions on Database Systems, vol 8, no 3, Sept, 1982, p. 381-416.

U. Dayal, H. Hwang, "View definition and generalization for database integration in a multidatabase system", IEEE Transactions on Software Engineering, vol 10, no 4, November, 1984, p. 628-645.

S. M. Deen, R. R. Amin, G. O. Ofori-Dwumfuo, M. C. Taylor, "The Architecture of a Generalised Distributed Database System - PRECI*", The Computer Journal, vol 28, no 3, 1985, p. 282-290.

S. M. Deen, R. R. Amin, M. C. Taylor, "Data Integration in Distributed Databases", IEEE Transactions on Software Engineering, vol 13, no 7, July 1987, p. 860-864.

S. A. Demurjian, D. K. Hsiao, "Towards a Better Understanding of Data Models through the Multilingual Database System", IEEE Transactions on Software Engineering, vol 14, no 7, July 1988, p. 946-958.

P. Denning, J. Dennis, J. Qualitz, Machines, Languages, and Computation, Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1978.

Dow-Jones, TBD, 1992.

P. A. Dwyer, J. A. Larsen, "Some Experiences with a Distributed Database Testbed System", Proceedings of the IEEE, vol 75, no 5, May 1987, p. 633-648.

J. Dzikiewicz, C. Egyhazy, "Cyrano: A Meta Model For Federated Database Systems," Proceedings of the ISMM International Conference on Intelligent Information Management Systems, International Society of Mini and Microcomputers, Washington, DC, 1994.

W. Effelsberg, M. Mannino, "Attribute equivalence in global schema design for heterogeneous distributed databases", Information Systems, vol 9, no 4, 1984, p. 237-240.

D. Gangopadhyay, T. Barsalou, "On the Semantic Equivalence of Heterogeneous Representations in Multimodel Multidatabase Systems", SIGMOD Record, vol 20, no 4, Dec, 1991, p. 35-39

G. Gardarim, P. Valduries, "Object Orientation in Relational Database Systems", Relational Databases and Knowledge Bases, Addison Wesley, 1989

V. Gligor, R. Popescu-Zeletin, "Transaction Management in Distributed Heterogeneous Database Management Systems", Information Systems, vol 11, no 4, 1986, p. 287-297

J. Grant, T. Sellis, "Extended Database Logic: Complex Objects and Deduction", Information Sciences, vol 52, no 1, October 1990, P. 85-110.

S. Greco, P. Rullo, "Complex-Prolog: A Logic Database Language for Handling Complex Objects", Information Systems, vol 14, no 1, 1989, p. 79-87.

R. Hackathorn, Enterprise Database Connectivity, John Wiley & Sons, Inc., New York, 1993.

D. Hsiao, M. Kamel, "Heterogeneous Databases: Proliferations, Issues, and Solutions", IEEE Transactions on Knowledge and Data Engineering, vol 1, no 1, March 1989
P. 45 - 62

IEEE, Computer, vol 24, no 12, December, 1991.

B. Jacobs, "On Database Logic", Journal of the ACM, vol 29, no 2, April 82, P310-332

B. Jacobs, Applied Database Logic, Prentice Hall, Inc, Englewood Cliffs, New Jersey, 1985.

M. Jarke, R. Gallersdorfer, M.A. Jeusfeld, M. Staudt, S. Eherer, "ConceptBase - a deductive object base for meta data management", Journal of Intelligent Information Systems, Vol. 4, No. 2, 1995, p. 167-192.

R. H. Katz, E. Wong, "Decompiling CODASYL DML into Relational Queries", Transactions on Database Systems, vol 7, no 1, March, 1982, p. 1-23.

W. Kim, Lochovsky, eds, Object-Oriented Concepts, Databases, and Applications, ACM Press, 1989.

W. Kim, "Object-Oriented Databases: Definition and Research Directions", IEEE Transactions on Knowledge and Data Engineering, vol 2, no 3, September 1990a, P. 327 - 341

W. Kim, Introduction to Object-Oriented Databases, The MIT Press, Cambridge, Massachusetts, 1990b.

W. Kim, J. Seo, "Classifying Schematic and Data Heterogeneity in Multidatabase Systems", Computer, Vol 24, No 12, December 1991, p. 12-18.

J. Larson, S. Navathe, R. Elmasri, "A Theory of Attribute Equivalence in Databases with Application to Schema Integration", IEEE Transactions on Software

Engineering, vol 15, No 4, April, 1989, p. 449-463.

W. Litwin, L. Mark, N. Roussopoulos, "Interoperability of Multiple Autonomous Databases", ACM Computing Surveys, Vol 22, No 3, September 1990, p. 267-293.

A. Motro, "Superviews: visual integration of multiple databases", IEEE Transactions on Software Engineering, vol 13, no 7, July, 1987, p. 785-798.

E. Oxborrow, H. Ismail, "KBZ - An Object Oriented Approach to the Specification and Management of Knowledge Bases", Proceedings of the 6th British National Conference on Databases, Ed WA Gray, 1988 p 21-46

J. Peckham, F. Maryanski, "Semantic data models", ACM Computing Surveys, vol 20, no 3, Sept, 1988, p. 154-189

E. Pitoura, O. Bukhres, A. Elmagarmid, "Object Orientation in Multidatabase Systems", ACM Computing Surveys, vol 27, no 2, June, 1995, p. 141-195.

C. Pu, "Superdatabases for Composition of Heterogeneous Databases", IEEE 1988 Data Engineering Conference, 1988, p. 548-555.

Calton Pu, Avraham Leff, Shu-Wie F. Chen, "Heterogeneous and Autonomous Transaction Processing", Computer, Vol 24, No 12, December 1991, p. 64-72.

Reuters News Service, TBD, 1992.

H. Rybinski, "On First-Order-Logic Databases", Transactions on Database Systems, vol 12, no 3, Sepember 1987, p. 325-349.

E. Sciore, "Comparing the Universal Instance and Relational Data Models", Advances in Computing Research, V3, The Theory of Databases, ed Kanellakis, 1986, p136-162

SDDTG of CODASYL System committee, "Stored-data description and data translation: a model and language", Information Systems, vol 2, no 3, 1977, p. 95-148

A. Sheth, J. Larson, "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases", ACM Computing Surveys, vol 22, no 3, September 1990, p. 183-236.

D. Shipman, "The Functional Data Model and the Data Language DAPLEX", Transactions on Database Systems, vol 6, no 1, March 1981, P. 140-173

N. Soparkar, H. Korth, A. Silberschatz, "Failure-Resilient Transaction Management in Multidatabases", Computer, Vol 24, No 12, December, 1991, p. 28-36.

N. Spyratos, "The Partition Model: A Deductive Database Model", Transactions on Database Systems, vol 12, no 1, March, 1987, p. 1-37

M. Staudt, H.W. Nissen, M.A. Jeusfeld, "Query by class, rule and concept", Applied Intelligence, Vol. 4, No. 2, 1994, pp. 133-157.

M. Templeton, D. Brill, A. Chen, S. Dao, E. Lund, "Mermaid - Experiences with Network Operation", Proceedings of the International Conference on Data Engineering, 1986, P. 292-300

G. Thomas, G. Thompson, C. Chung, E. Barkmeyer, F. Carter, M. Templeton, S. Fox, B. Hartman, "Heterogeneous Distributed Database Systems for Production Use", ACM Computing Surveys, vol 22, no 3, September 1990, P. 237-266

J. D. Ullman, Principles of Database Systems, second edition, Computer Science Press, Rockville, MD, 1982.

J. D. Ullman, Principles of Database and Knowledge-Base Systems, vols 1 & 2, Computer Science Press, Rockville, MD, 1989.

I. White, Using the Booch Method: A Rational Approach, The Benjamin/Cummings Publishing Company, Inc, Redwood City, California, 1994.

Appendix A: A BNF for Cyrano

```
<CLASS>
        ::=     "CLASS" <NAME> "IS"
                (
                        <DERIVED_DEFINITION>
                |       <GATEWAY_DEFINITION>
                ) "END" "CLASS" "."

<DERIVED_DEFINITION>
        ::=     "DERIVED"
                [ "IS" <CLASS_NAME> { "," <CLASS_NAME> } ]
                "WITH" { <DERIVATION> ";" }

<DERIVATION>
        ::=     [ <Variable > { "," <VARIABLE> } ]
                ":" <VALUE> "WITH" { <DERIVED_METHOD> ";" } "END"

<VARIABLE>
        ::=     <CLASS_NAME> <VAR_NAME>

<CLASS_NAME>
        ::=     <NAME>

<VAR_NAME>
        ::=     <NAME>

<DERIVED_METHOD>
        ::=     <CLASS_NAME> <NAME>
                [ "(" [ <VARIABLE> { "," <VARIABLE> } ] ")" ] "IS" <VALUE>

<VALUE>
        ::=     <BASE_VALUE> { <COMPLEX_VALUE> }
        |       <COMPLEX_VALUE>

<BASE_VALUE>
        ::=     "(" <VALUE> ")"
        |       <CONSTANT_VALUE>
        |       <VARIABLE_VALUE>
        |       <QUANTIFIED_VALUE>
        |       <COND_VALUE>
```

```
<CONSTANT_VALUE>
        ::=     <Quoted_String>
        |       <Integer>
        |       <True>
        |       <False>

<COMPLEX_VALUE>
        ::=     "." <NAME>
                [ "(" [ <VALUE> { "," <VALUE> } ] ")" ]
        |       <OPERATOR> <VALUE>
        |       "NOT" <VALUE>

<QUANTIFIED_VALUE>
        ::=     ( "EXISTS" | "FORALL") "(" <VARIABLE> ")" <VALUE>

<COND_VALUE>
        ::=     "IF" "(" { "(" <VALUE> "THEN" <VALUE> ")" } ")"

<OPERATOR>
        ::=     "+"
        |       "="
        |       "-"
        |       "/"
        |       "*"
        |       "<"
        |       ">"
        |       "<>"
        |       "<="
        |       ">="
        |       "AND"
        |       "OR"

<GATEWAY_DEFINITION>
        ::=     "GATEWAY"
                [ "IS" <CLASS_NAME> { "," <CLASS_NAME> } ]
                <GATEWAY_TYPE> "WITH" { <METHOD> ";" }

<GATEWAY_TYPE>
        ::=     ":" "PARADOX"
        |       ":" "DATALOG"
```

<METHOD>
::= <CLASS_NAME> <NAME> ["(" [<INTEGER>] ")"]

Appendix B: Booch Method Notations

The Booch method is a methodology for object oriented design and analysis. The design of Roxanne, the Cyrano prototype, follows the Booch method. This appendix describes those aspects of the Booch method used to design Roxanne. The Booch method is described fully in [BOOC91] [WHIT94].

A Booch method design revolves around two types of diagrams. A class diagram shows the classes that comprise the design. The class diagram is the blueprint of the software: each class in it corresponds to a class that needs to be written.

An object diagram provides a snapshot of objects in the running system. It is used to show operational scenarios, depicting how complex objects can be constructed of other objects or how objects interact.

Figure B.1 shows the Booch icons that appear in class diagrams.

The primary building block in a class diagram is a **Class**. A **Class** is a complex data type. It specifies the attributes and operations of its member objects. There are four variations on the basic class.

An **Abstract Class** can have other classes inherit from it, but cannot have objects that are members of it. It is the Booch representation of a C++ virtual class.

An **Instantiated Class** is a class with formal parameters. Creating the class requires the instantiation of the parameters. This is the Booch representation of a C++ template or an Ada generic package.

An **Instantiation** is an instance of an instantiated class with values assigned to its formal parameters.

A **Class Utility** is a class without attributes. In effect, it is a collection of functions.

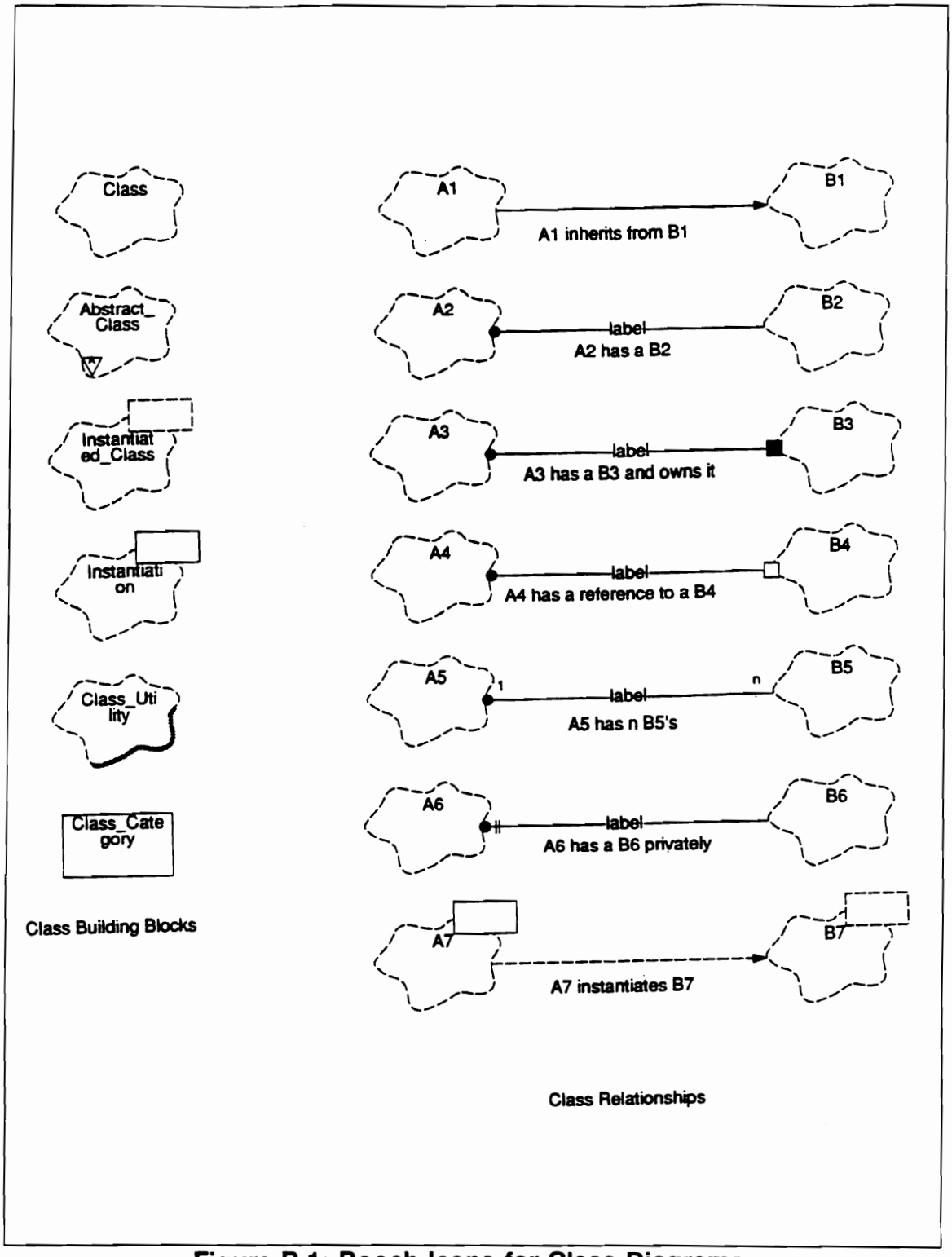

Figure B.1: Booch Icons for Class Diagrams.

A **Class Category** contains several related classes. It is a higher level building block then classes.

Classes can be related in several ways.

The INHERITS or IS-A relationship indicates that, if class A INHERITS from class B, then A is a subclass of B and inherits the attributes and operations of B.

The HAS-A relationship indicates that, if class A HAS-A class B, then an object of class A has an attribute with a value that is an object of class B. A HAS-A relationship can have several adornments. It can indicate ownership, meaning that A is responsible for the destruction of B. It can indicate a reference, meaning that A is not responsible for the destruction of B but only has a reference to B. It can have an associated cardinality, indicating that A has some specified number of B's, or that each B is in the relationship to some specified number of A's. Finally, the HAS-A relationship can be private, meaning that the B is not directly available to objects accessing A.

The INSTANTIATES relationship indicates that, if A INSTANTIATES B, then A is an instantiation and B is its corresponding instantiated class. For example, Roxanne contains an instantiated class called **List** with one parameter, the class of items to be listed. The instantiation **List(Method)** instantiates **List** to list objects of class **Method**.

Figure B.2 shows the Booch icons that appear in object diagrams.

The sole building block for object diagrams is an **Object**. An object is a data item. It is created as a member of a specified class. An object of class A is a member of all classes from which A inherits.

Each object has a primary class. On an object diagram, the class of an object is either clear by the object's name or it will be attached as a label to the object.

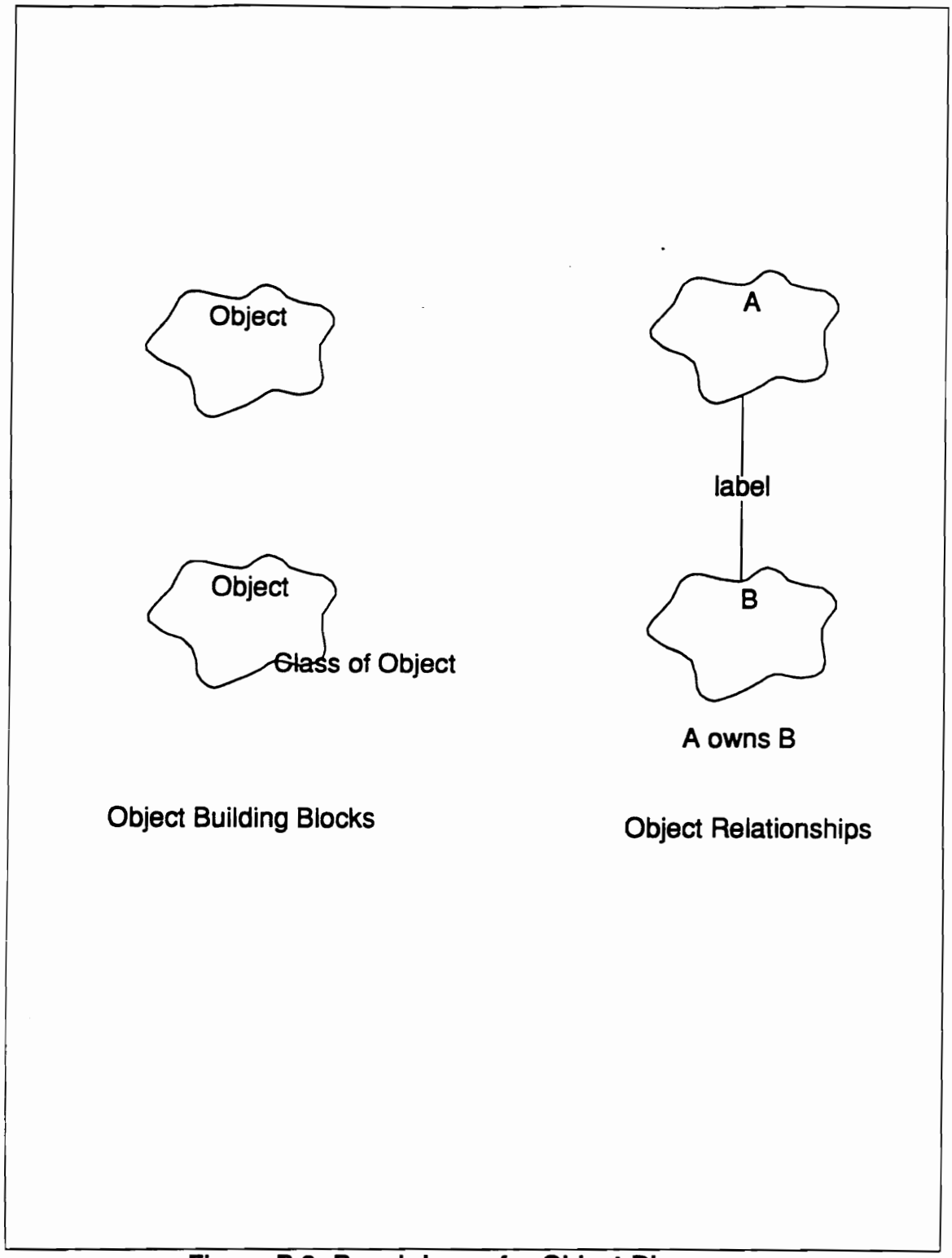

Figure B.2: Booch Icons for Object Diagrams.

The Booch method provides for a single type of relationship between objects. In this work, it is used to represent a reference from one object to another, with the higher object in the diagram holding the reference.

Appendix C: Roxanne Design Description

This appendix contains a description of Roxanne's design. Roxanne was designed using the Booch method for object oriented analysis and design. See appendix B for a description of the Booch method. This chapter contains class diagrams for Roxanne using Booch symbols.

Roxanne's design is broken into a set of class categories. Figure C.1 shows the relationships between these class categories. The following sections describe the class categories and the classes that they contain.

C.1. The Windows Class Category

The Windows class category contains the windows of the Roxanne interface. Because Roxanne uses a windowing interface, this class category also contains the main function for the program. Figure C.2 contains the class diagram for the Windows class category. This section describes these classes in greater detail.

A **Class_List_Window** displays the window containing the list of active Cyrano classes and manages the user's interactions with that window. This is Roxanne's main window. Therefore, this class provides the highest level of control over the user's interactions with Roxanne.

An **Edit_Window** allows a user to edit a file. Roxanne uses this to allow the user to edit class specifications.

A **File_Display** window displays the contents of a file or buffer to the user. Roxanne uses this to display various data to the user, including the results of running a class and any errors encountered during Roxanne's processing.

C.2. The Classes Class Category

The Classes class category contains data objects that implement Cyrano

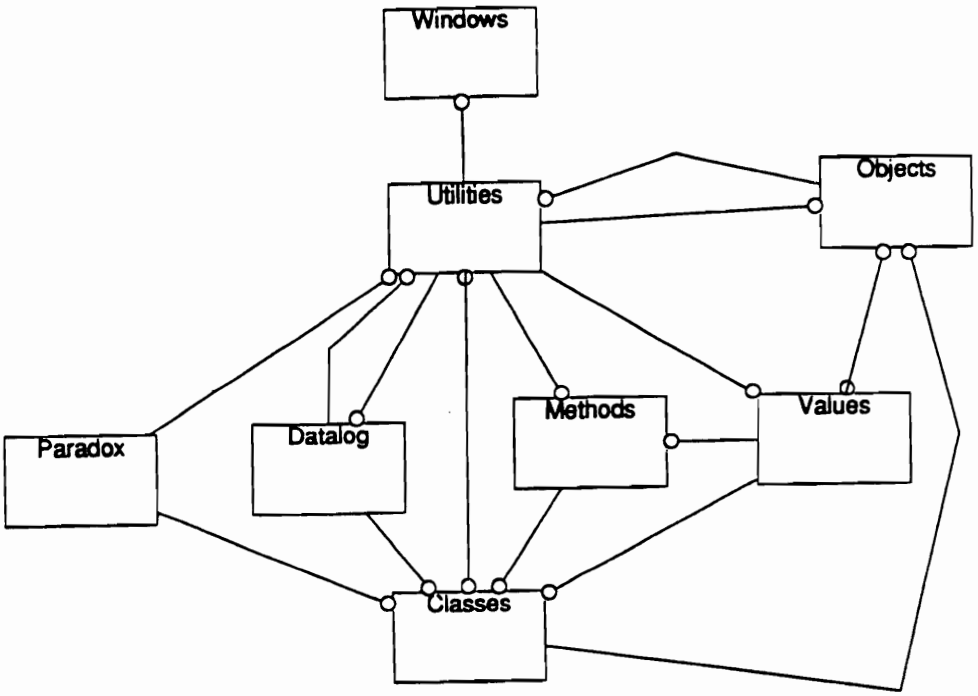

Figure C.1: The Roxanne Class Categories

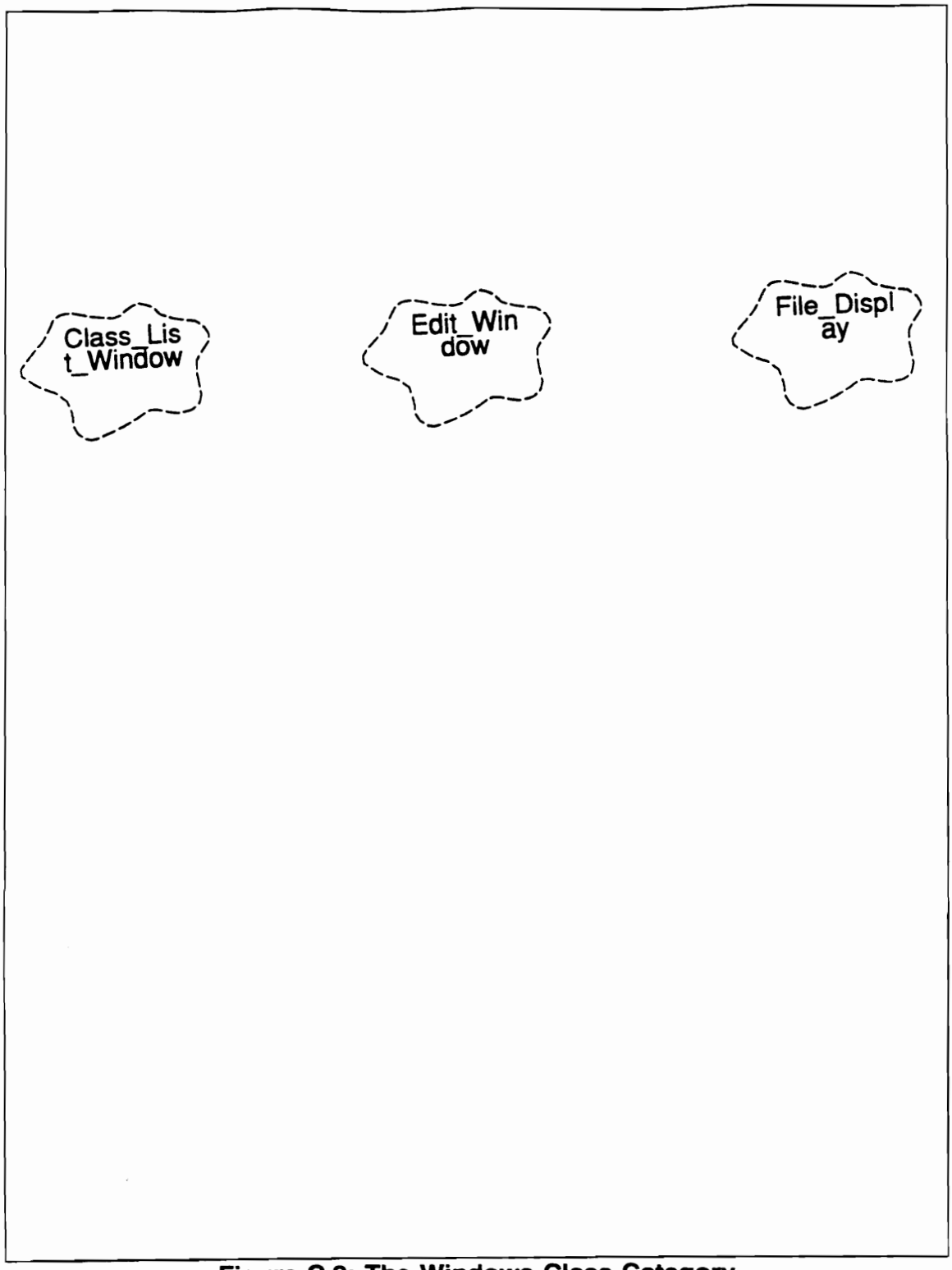

Figure C.2: The Windows Class Category

classes. This consists primarily of the Class class and its subclasses. Figure C.3 contains the class diagram for the Classes class category. The following describe these classes in greater detail.

Class is a virtual class that is the basis of all C++ classes used by Roxanne to represent Cyrano classes. Class maintains those data items common to all Cyrano classes: the class name and the name of the file that contains the class's specification.

Builtin_Class represents a Cyrano built-in class. It is a subclass of Class. It provides methods to find all members of the class, to add a method to the class, and to find a given method of the class.

Gateway_Class is a virtual class that is the basis of all C++ classes representing Cyrano gateway classes. It contains lists of the Cyrano methods of the class and its Cyrano superclasses. It contains methods to read in the body of the class from the class specification.

Derived_Class represents a Cyrano derived class. It contains the list of derivations of the class and of its superclasses. It includes a constructor that builds the class based on its class specification and methods to find all member objects of the class.

Derivation represents one derivation of a derived class. It includes lists of the derivation's base variables and methods, and the value that serves as the guard of the class. It includes methods to find a Cyrano method of the derivation, to find all Cyrano objects that can be derived by this derivation, and to find all Cyrano classes used in this derivation.

Datalog_Class represents those gateway classes that serve as gateways to the Datalog database. It is a subclass of Gateway_Class. It contains the Datalog predicate corresponding to the class. It provides a constructor and a method to find all members of the class.

Paradox_Class represents those gateway classes that serve as gateways

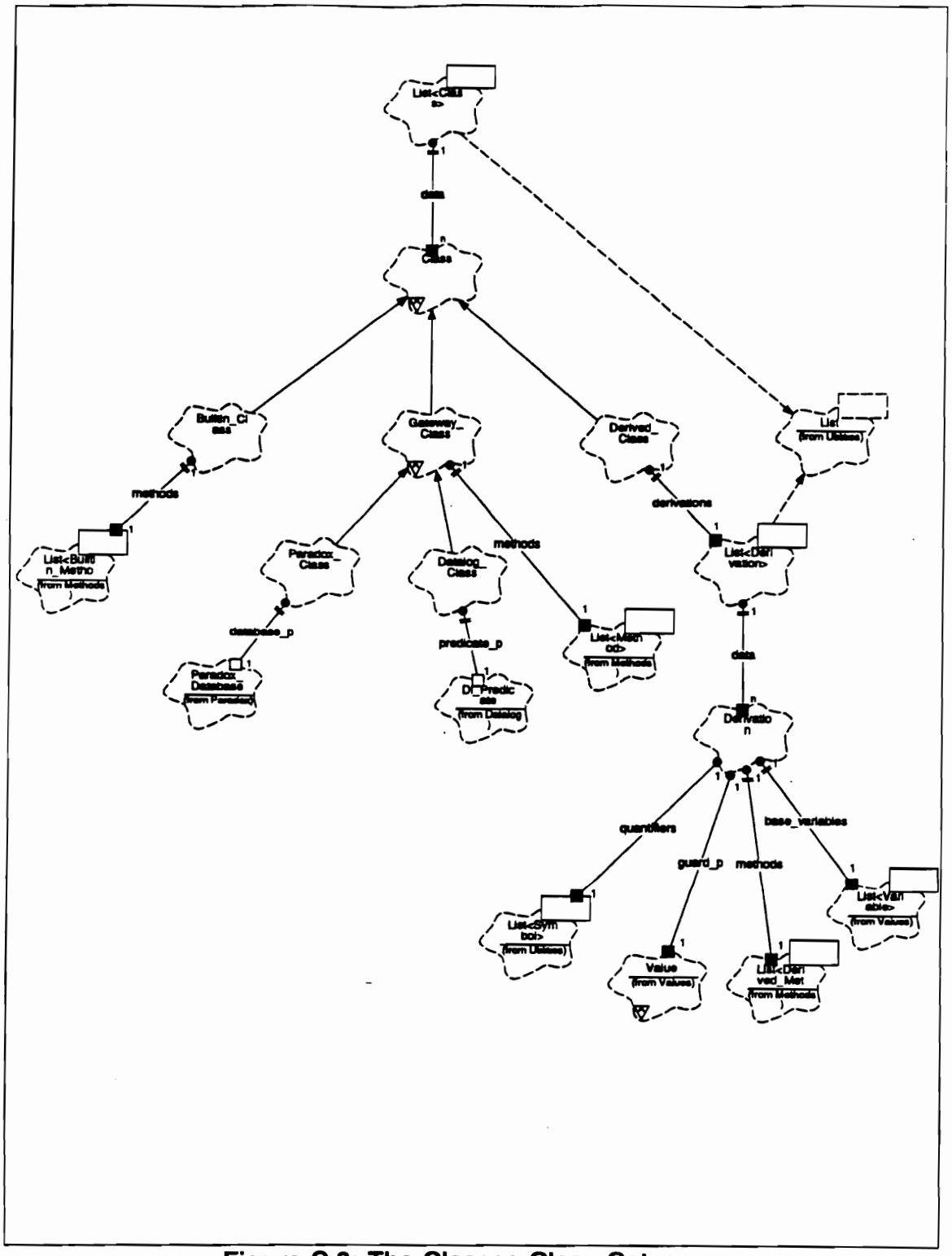

Figure C.3: The Classes Class Category

to the Paradox database. It is a subclass of Gateway_Class. It contains the Paradox data structures needed to retrieve data from the Paradox relation corresponding to this Cyrano class. It provides a constructor and a method to find all members of the class.

C.3. The Methods Class Category

The Methods class category contains classes that represent the various methods of Cyrano classes. Each type of class has a corresponding type of method which handles messages in different ways. Figure C.4 contains the class diagram for the Methods class category. The following paragraphs describe these classes.

Method represents a Cyrano method of any kind of class. It serves as the parent class for other types of methods. It includes data items for the method name, type, and arity. It includes a constructor and methods for setting and getting its components.

There are child classes to Method for representing methods of derived and built-in classes. However, Roxanne uses Method to represent the methods of a gateway class. This is because a gateway method does not provide any additional functionality: resolving the gateway method requires passing it to the member database.

Derived_Method represents a method of a derived class. It is a subclass of Method. It includes a list of its parameter variables and the value which, when instantiated, gives the object returned by the method. It also includes a constructor and an operation to return its value.

Builtin_Method represents a method of a built-in class. It is a subclass of Method. It includes a pointer to a function that, when run, produces the method's value. It also includes an operation to return that function.

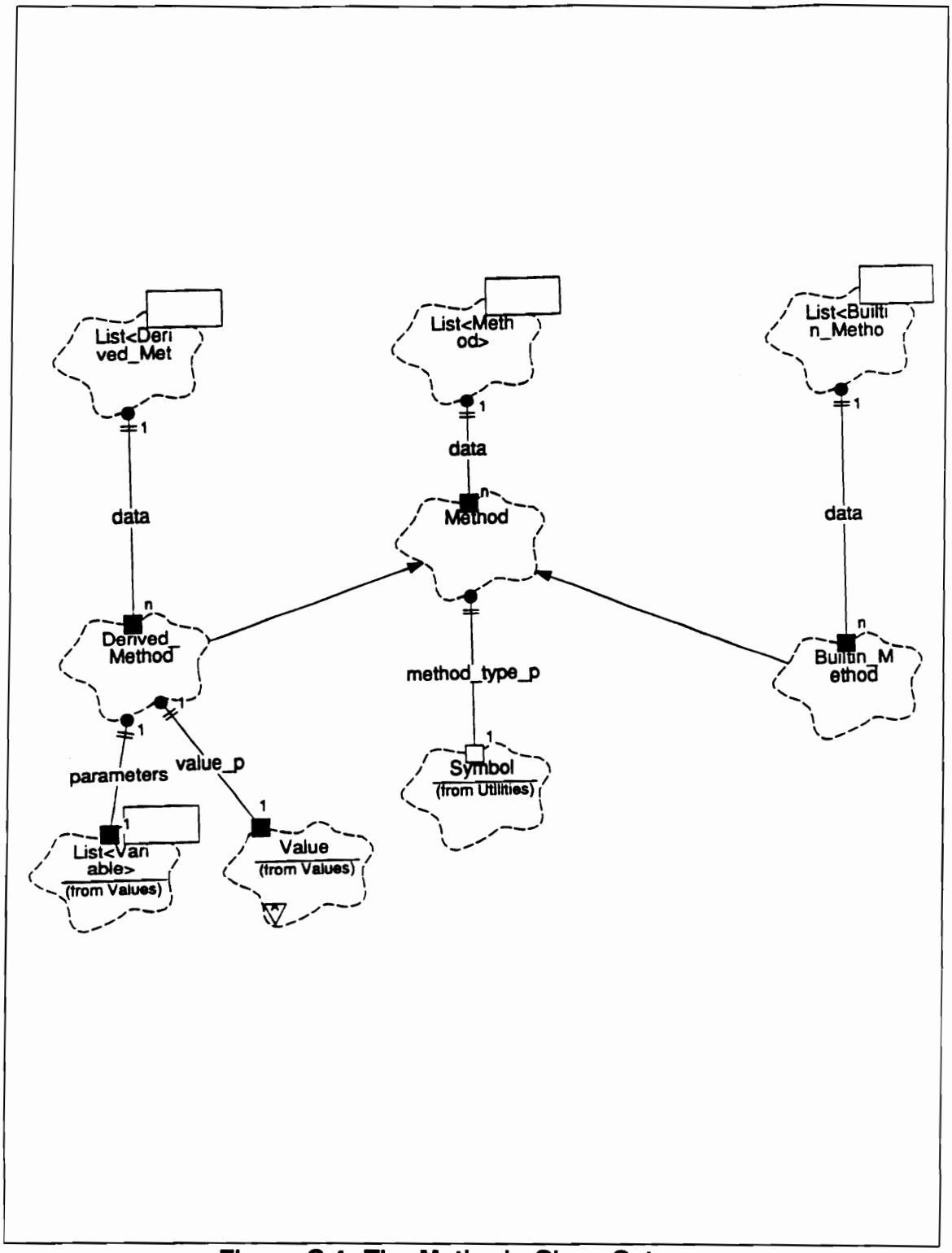

Figure C.4: The Methods Class Category

C.4. The Values Class Category

The Values class category contains classes that represent values in Cyrano. This includes the various ways that values are expressed in a Cyrano database, ncluding constants, method invocations, and quantified expressions. Figure C.5 contains the class diagram for the Values class category. The following paragraphs describe these classes.

Value is a virtual class that represents the different types of values that Cyrano can have. It includes virtual functions to produce the object generated by the value given some set of variable values.

Constant_Value represents the appearance of a constant in a Cyrano class specification. It is a subclass of Value. It contains a data item containing its value. It includes an operation to return that value.

Variable_Value is a value that names some variable. It includes a data item containing the variable's name. It includes an operation to generate the value of that variable.

Complex_Value is a value that corresponds to the passing of some message to a Cyrano object. It is a subclass of Value. It includes a value that should, when instantiated, receive the message. It also includes the message and values for any parameters of it. It includes an operation to generate the object for the value.

Quantified_Value represents values that have quantifiers in them (i.e., values of the form "FORALL X (body)" or "EXISTS Y (body)"). It is a subclass of Value. It includes members for the quantifier type, for the quantifier variable, and for the body. It also includes an operation to generate the associated object given a set of variable values.

Variable represents a Cyrano variable. It contains member items for its type, name, and value. The value member will frequently be null when the value is not yet known. It includes operations to get and set its data members.

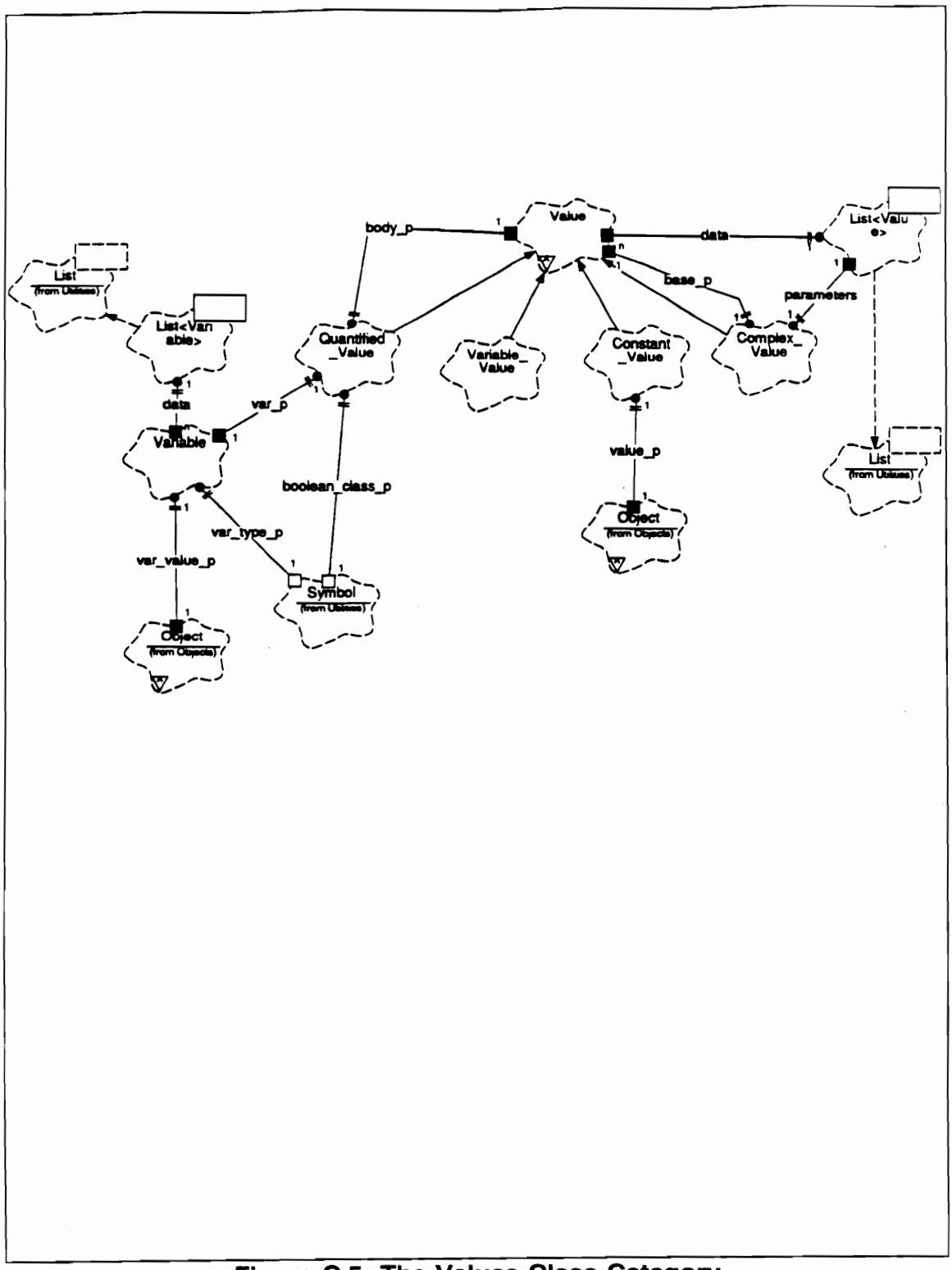

Figure C.5: The Values Class Category

C.5. The Objects Class Category

The Objects class category contains classes that represent Cyrano objects. This includes built-in, gateway, and derived objects. Figure C.6 contains the class diagram for the Objects class category. The following paragraphs describe these classes in greater detail.

Object represents a Cyrano data object. It is a virtual class that serves as a parent for the various types of objects. It includes only virtual operations.

Builtin_Object represents an object that is a member of a built-in class. It is a subclass of Object. It contains its value and a pointer to its type. It includes operations to write it to a stream, copy it, and pass a message to the object.

Relation represents one type of gateway object. In particular, it represents relations, or tuples of name-value pairs. Relations are the data objects returned by both the Paradox and Datalog gateway databases, and therefore are sufficient as gateway objects for the current implementation of Roxanne. Relation is a subclass of Object. It includes as data members a list of variables that serve as the name-value pairs of its members. It also includes operations to copy the relation, add data to it, and pass a message to the relation.

Derived_Object represents objects that are members of a derived class. It is a subclass of Object. It includes the derivation by which the object was derived, a list of the object's base objects, and a list of its inherited super-objects. It includes operations to add base objects, add superobjects, copy the object, and pass a message to the object.

C.6. The Utilities Class Category

The utilities class category contains various utility classes used by the other Roxanne classes. Figure C.7 contains the class diagram for the Utilities class category. The following paragraphs describe these classes.

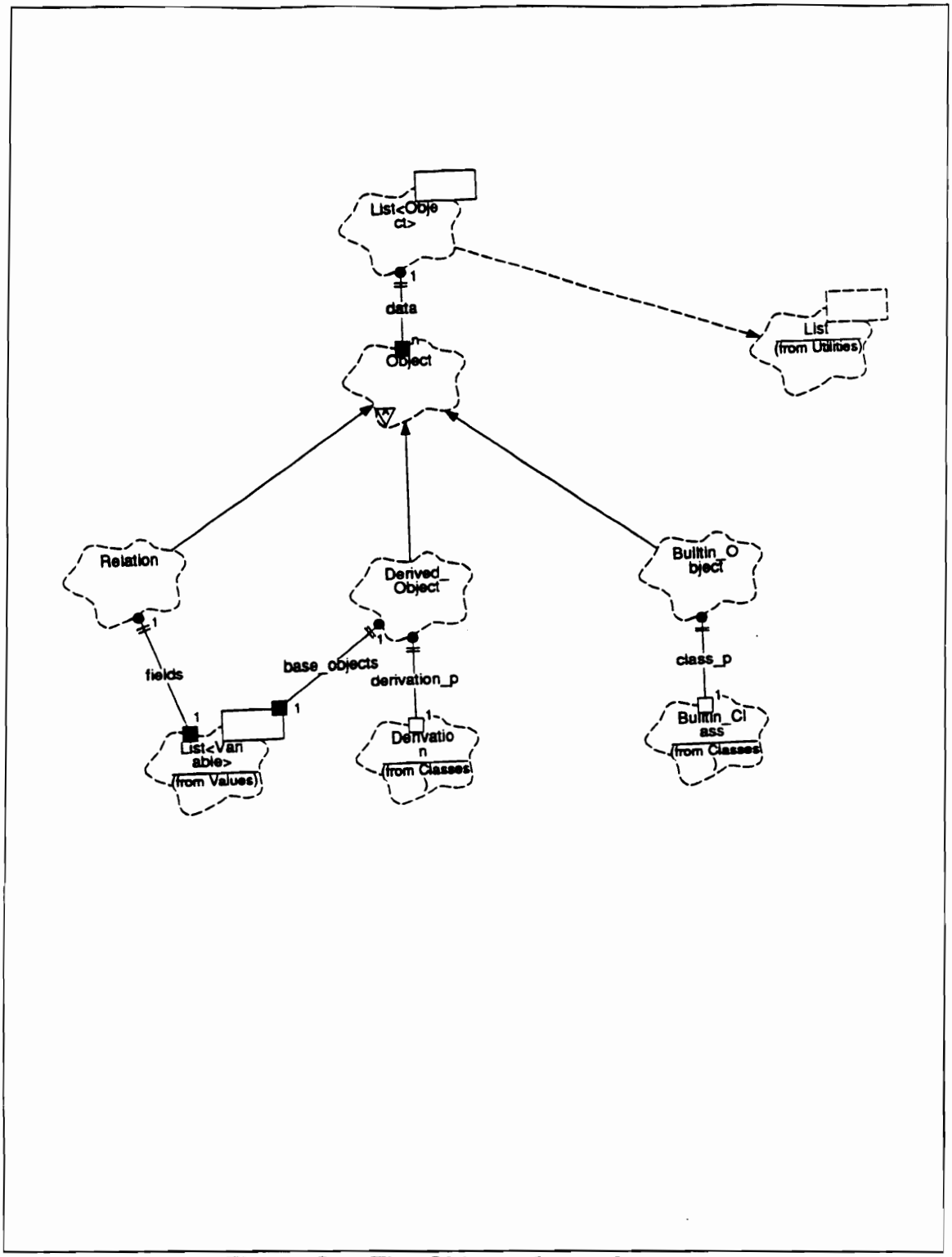

Figure C.6: The Objects Class Category

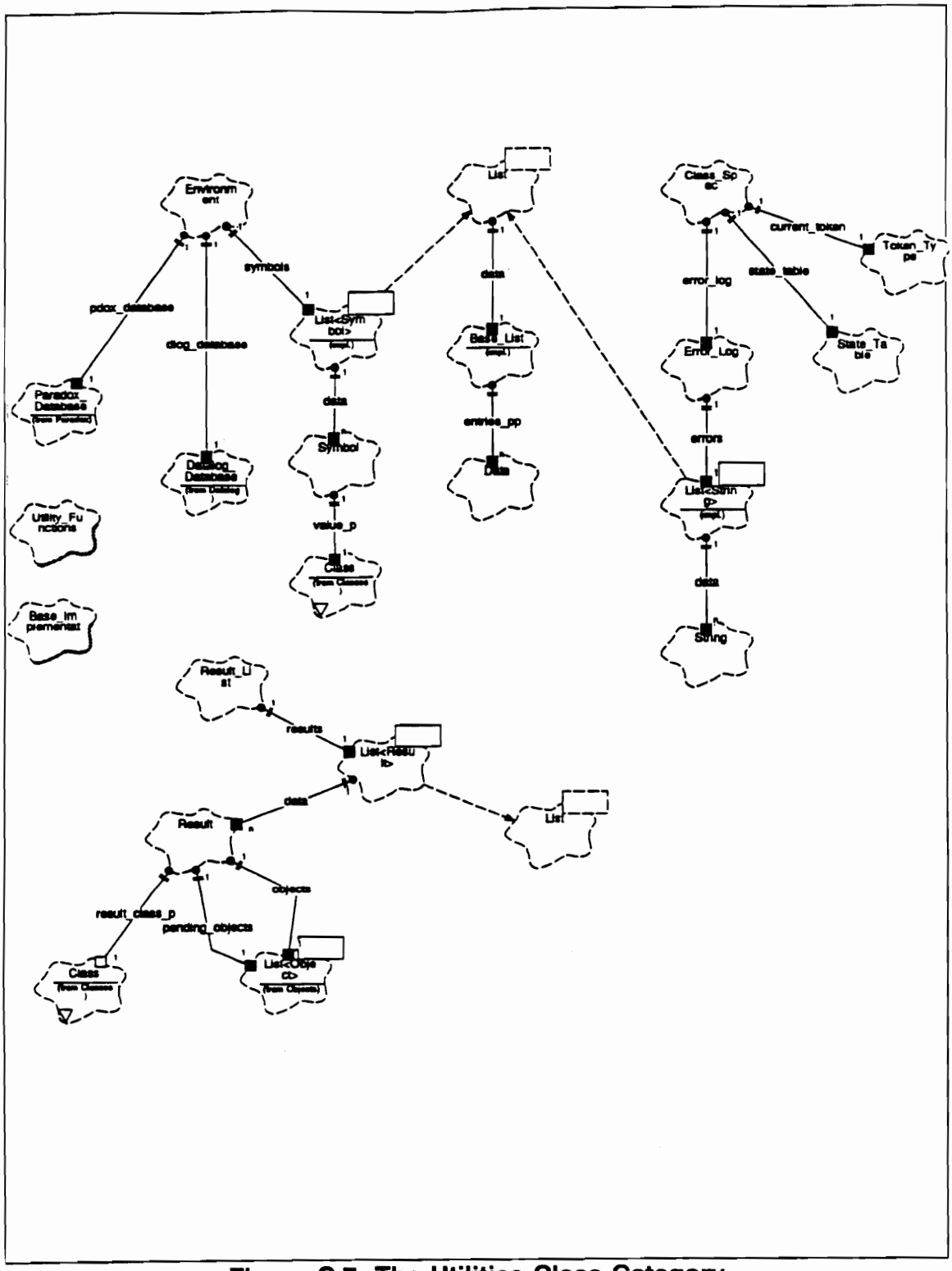

Figure C.7: The Utilities Class Category

List is a template class that serves to list data elements. Because it is a template, it can be instantiated to serve as a list of any specific data type.

Base_List maintains a list of untyped objects. It is used as part of the implementation of List.

Environment stores the operational environment of Roxanne. This includes the list of symbols, which represent active classes, and the gateways to the Paradox and Datalog member databases. It includes operations to find and add symbols and to retrieve the databases.

Symbol maps the name of a class to the class object. At times, its class may be null if no value is currently set for the symbol. This provides a level of indirection in references to classes, simplifying the changing of classes even when other classes reference them. Symbol includes operations to get, set, and clear its class, and to get its name.

Error_Log maintains a log of errors. It includes a list of the errors and operations to add to that list and to write it to an output stream.

Class_Spec represents a Cyrano class specification. Roxanne constructs a Class_Spec whenever it needs to access a new Cyrano class. The Class_Spec is attached to a file containing the class specification. It then returns the contents of that file one token at a time. Therefore, the Class_Spec serves as a lexical analyzer for the Cyrano class specification.

The Class_Spec has members for its filename, input file, current token, current token value, a state table used in lexical analysis, and an error log used to log errors. It includes a constructor and operations to get the next token and its value and to skip to the next token.

State_Table maintains the state table of a finite state machine. It includes a list of its states and their transitions. It also includes operations to construct the state table and to find the next state given the current state and an input character.

Base_Implementations is a set of functions that implement the built-in

functions for Roxanne's supported built-in classes. Currently, Roxanne supports Boolean, integer, and string built-in classes.

Result represents the results of instantiating a class. It includes the class that was instantiated, the list of objects found to be members of the class, the list of objects found to be members of the class in the most recent application of the instantiation algorithm, and a flag indicating whether all objects of the class have been found. It includes methods to get the class and object list, to add new objects, and to note completion of cycles of the class instantiation algorithm.

Result_List is a list of Results. When a derived class is instantiated, Roxanne generates a Result_List containing a result for every class used in the derivation. Roxanne then instantiates all classes in the Result_List, causing the Results to be filled with objects. Because the target derived class is added to the Result_List, when all Results are completely instantiated the target class is also instantiated.

Result_List includes a list of its Results. It includes operations to add a class to the list, to find objects from a given Result, and to instantiate all classes in all of its Results.

C.7. The Paradox Class Category

The Paradox class category contains the class that provide the gateway to the Paradox database. Figure C.8 contains the class diagram for the Paradox class category.

Paradox_Database is the sole class in the Paradox class category. It serves as the gateway to the Paradox member database. It includes the Paradox objects representing the Paradox engine and database. It includes operations to get either of those members.

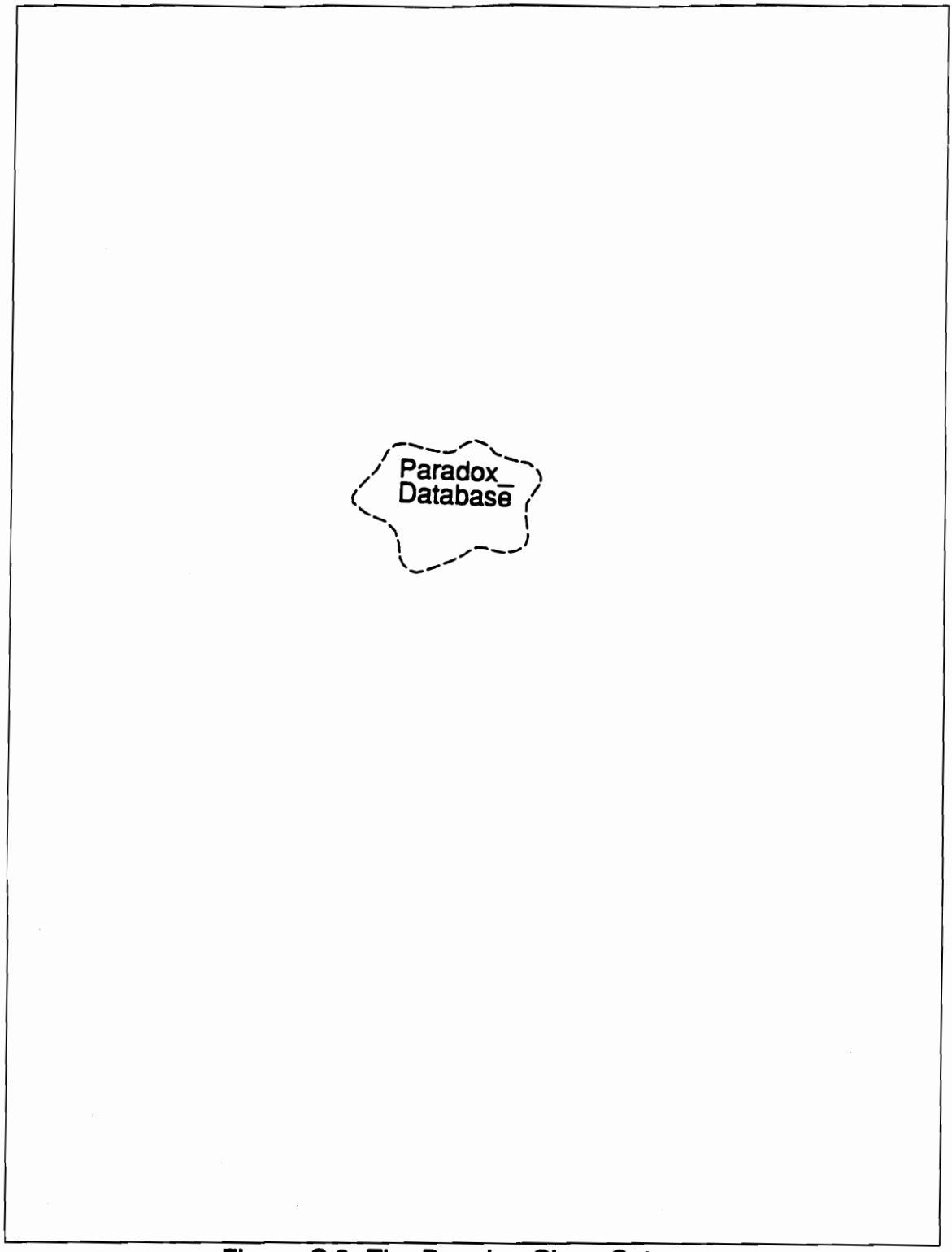

Figure C.8: The Paradox Class Category

C.8. The Datalog Class Category

The Datalog class category contains the classes that implement the Datalog database and provide Roxanne's gateway to that database. The classes in this category generally correspond to other Roxanne classes: the Datalog database implementation is similar to Roxanne's implementation of derived classes. Figure C.9 contains the class diagram for the Datalog class category. The following paragraphs describe these classes.

Datalog_Database keeps the entire Datalog database, including its environment. It includes a constructor and an operation to find a given predicate in the database.

Dl_Environment maintains the run-time environment of the Datalog database. It corresponds to the Roxanne Environment class. It maintains a list of all active Datalog predicates and operations to add and retrieve predicates to that list.

Dl_Predicate represents a single Datalog predicate. It includes the name and arity of a predicate along with a list of Dl_Rules, each of which represents one rule of the predicate. It includes operations to find its arity and name, add rules, and find a result given some set of known variable values. Its design is similar to the Roxanne Derived_Class class.

Dl_Rule represents a single Datalog rule. It includes a list of Dl_Clauses, each of which represents one clause of the rule. It also includes a list of constants set by the rule, the rule's predicate's name, and the arity of that predicate. It includes operations to get its components and to find tuples that match the rule. It corresponds to the Roxanne Derivation class.

Dl_Clause represents a single clause of a Datalog rule. It includes the name of the predicate called by the Clause and a list of its parameters. It includes operations to test whether the clause matches some tuple of values and to get the corresponding predicate.

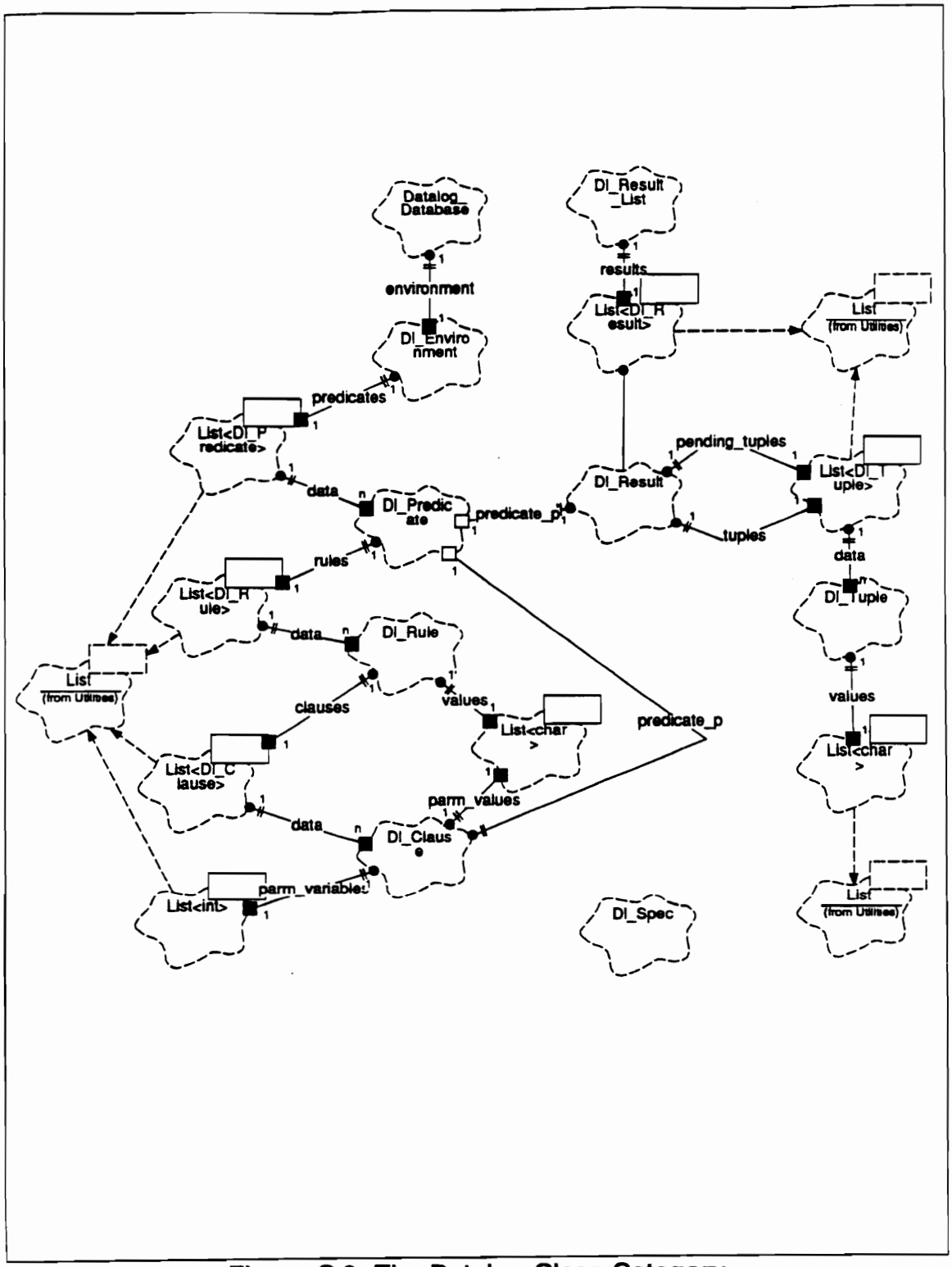

Figure C.9: The Datalog Class Category

DI_Result stores the list of tuples found to match some given predicate. It corresponds to the Roxanne Result class.

DI_Result_List stores all DI_Results used to resolve some query operation. It corresponds to the Roxanne Result_List class.

DI_Spec is the class specification for a Datalog database. It corresponds to the Roxanne Class_Spec class.

DI_Tuple is a tuple generated by the Datalog database. The Datalog database uses this instead of Relation to represent its return values to maintain an isolation between the Datalog implementation and the other class categories.

Appendix D: Roxanne Source Code

D.1. Roxanne Include Files.

```
// baseimps.h

#ifndef _BASEIMPS_H
#define _BASEIMPS_H

#include "environ.h"

// the following are true and false strings
extern char *const true_value;
extern char *const false_value;

// the following are class types
extern char *const boolean_class_name;
extern char *const integer_class_name;
extern char *const string_class_name;

// the following are operator names and lengths
extern const int max_operator_len;
extern char *const plus_message;
extern char *const minus_message;
extern char *const asterisk_message;
extern char *const slash_message;
extern char *const less_than_message;
extern char *const greater_than_message;
extern char *const equals_message;
extern char *const not_equals_message;
extern char *const less_or_equals_message;
extern char *const greater_or_equals_message;
extern char *const not_message;
extern char *const and_message;
extern char *const or_message;

// the following is the function for this
extern void setup_base_classes(Environment *env_p);


#endif
```

```
// bin_obj.h

#ifndef _BIN_OBJ_H
#define _BIN_OBJ_H

#include <iostream.h>

class Builtin_Object;

#include "object.h"
#include "list.h"
#include "binclass.h"
#include "errorlog.h"

// Builtin_Object is a object of a built-in class
class Builtin_Object : public Object
{
    private:
        Builtin_Class *class_p;         // class of this object
        char *value_p;                  // value of this object

    public:
        Builtin_Object(Builtin_Class *in_class_p, char *in_value);
                                        // constructor
        ~Builtin_Object(void);          // destructor
        void output(ostream &out, int indentation);
                                        // output this
        Object *clone(void);            // copy this
        int true(void);                 // is this true?
        Object *run_message(char *message, List<Object> *parms_p,
                Error_Log *log_p);      // run this message
        char *value(void);              // value of this
        Builtin_Class *type(void);      // type of this
};


#endif
```

```
// binclass.h

#ifndef _BINCLASS_H
#define _BINCLASS_H

class Builtin_Class;

#include "binmthd.h"
#include "class.h"
#include "list.h"
#include "object.h"
#include "errorlog.h"
#include "environ.h"

// Builtin_Class is the class of builtin-objects
class Builtin_Class : public Class
{
    private:
        List<Builtin_Method> methods;   // methods of the class

    public:
        Builtin_Class(char *name, Environment *env_p);
                                        // constructor
        List<Object> *run(Error_Log *log);
                                        // run the class
        void add_method(char *msg, int arity, Symbol *type_p,
                Object *(*implementation_p)(Object *, List<Object> *));
                                        // add method of class
        Builtin_Method *find_method(char *msg, int arity);
                                        // Find a method
};

#endif
```

```
// binmthd.h

#ifndef _BINMTHD_H
#define _BINMTHD_H

class Builtin_Method;

#include "method.h"
#include "object.h"
#include "list.h"

// Builtin_Method() is a method of a Builtin_Class
class Builtin_Method : public Method
{
    private:
        Object *(*implementation_p)(Object *, List<Object> *);

    public:
        Builtin_Method(char *msg, int arity, Symbol *type_p,
                Object *(*in_implementation_p)(Object *, List<Object> *));
        Object *(*implementation())(Object *, List<Object> *);
};

#endif
```

```
// class.h

#ifndef _CLASS_H
#define _CLASS_H

class Class;

#include "list.h"
#include "object.h"
#include "classpec.h"
#include "environ.h"
#include "errorlog.h"
#include "result.h"
#include "reslist.h"

// Class is the class of an object.  This is a virtual class.
class Class
{
    private:
        char *class_name;               // name of the class
        char *filename;                 // name of the file it is in

    public:
        Class(char *in_name, Class_Spec *in, Environment *env_p);
                                        // constructor
        virtual ~Class(void);           // destructor
        virtual List<Object> *run(Error_Log *log_p) = 0;
                                       // run the class and get its members
        virtual int run_for_result(Result *result_p, Result_List *list_p,
                                   Error_Log *log_p);
                                        // run and add members to result
        char *text(void);               // get text of class
        char *name(void);               // get name of class
        virtual void get_bases(Result_List *list_p, Error_Log *log_p);
                                        // get base classes of this
};


// read_class() reads in a class of any subtype
Class *read_class(Class_Spec *in, Environment *env_p);

#endif
```

```
// classpec.h

#ifndef _CLASSPEC_H
#define _CLASSPEC_H

#include <fstream.h>

class Class_Spec;

#include "statetbl.h"
#include "errorlog.h"

// token_type is the enumerated list of types of tokens
enum token_type {End_of_Input,
                 Class_Label,
                 Name,
                 Is_Label,
                 End_Label,
                 Period,
                 Derived_Label,
                 With_Label,
                 Semicolon,
                 Colon,
                 Exists_Label,
                 Forall_Label,
                 And_Label,
                 Or_Label,
                 Not_Label,
                 Open_Paren,
                 Comma,
                 Close_Paren,
                 String,
                 Integer,
                 Gateway_Label,
                 Paradox_Label,
                 Datalog_Label,
                 True_Label,
                 False_Label,
                 Plus,
                 Equals,
                 Minus,
                 Slash,
                 Asterisk,
                 Less_Than,
                 Greater_Than,
                 Not_Equals,
                 Less_or_Equals,
                 Greater_or_Equals};

// Class_Spec defines a class specification
class Class_Spec
{
    private:
        char *filename;                 // name of input file
        ifstream *token_input_p;        // token input
        token_type current_token;       // current token type
        char *token_value_buffer;       // buffer for current token
        char *token_value_p;            // ptr to current token
```

```
        State_Table state_table;      // state table for this
        Error_Log error_log;          // list of errors

        void fill_in_state_table(void);
                                      // fill in state table
        void skip_blanks(void);       // skip over blanks

    public:
        Class_Spec(char *filename);   // construct it
        ~Class_Spec(void);            // destruct it
        token_type next_token(void);  // get the next token
        void skip_token(void);        // skip to the next token
        char *get_value(void);        // get the current token value
        void log_error(char *);       // record an error
        int is_errors(void);          // get error flag
        Error_Log *get_errors(void);  // get error log
        char *text(void);             // get full text of input
};

#endif
```

```
// cllstwin.h

#ifndef _CLLSTWIN_H
#define _CLLSTWIN_H

#include <owl\applicat.h>
#include <owl\framewin.h>
#include <owl\dialog.h>
#include <owl\listbox.h>

#include "environ.h"

// Class_List_Window is a class for a class list window
class Class_List_Window : public TDialog
{
    public:
        Class_List_Window(TWindow *parent, TResId res_id);
                                        // constructor
        ~Class_List_Window(void);       // destructor

    protected:
        void handle_new(void);          // handle new button hit
        void handle_delete(void);       // handle delete button
        void handle_run(void);          // handle run event
        void handle_edit(void);         // handle edit event
        void handle_save(void);         // handle save event
        void handle_load(void);         // handle load event
        BOOL CanClose(void);            // can we close window?

        DECLARE_RESPONSE_TABLE(Class_List_Window);

    private:
        TListBox *class_list_p;         // list of classes
        void edit_class(char *buffer = NULL);
                                        // edit a class
        Environment *environment_p;     // processing environment
};


#endif
```

```
// constval.h

#ifndef _CONSTVAL_H
#define _CONSTVAL_H

class Constant_Value;

#include "value.h"
#include "classpec.h"
#include "environ.h"
#include "object.h"
#include "list.h"
#include "variable.h"
#include "errorlog.h"

// Constant_Value is a value that is a constant
class Constant_Value : public Value
{
    private:
        Object *value_p;              // value of this

    public:
        Constant_Value(Class_Spec *in_p, Environment *env_p);
                                      // constructor
        ~Constant_Value(void);        // destructor;
        Object *generate_value(List<Variable> *, Error_Log *);
                                      // get value of this
};

#endif
```

```
// cplxval.h

#ifndef _CPLXVAL_H
#define _CPLXVAL_H

class Complex_Value;

#include "value.h"
#include "list.h"
#include "classpec.h"
#include "environ.h"
#include "variable.h"
#include "errorlog.h"
#include "symbol.h"

// Complex_Value is a complex value object
class Complex_Value : public Value
{
    private:
        Value *base_p;                  // base value of this
        List<Value> parameters;         // parameters of this
        char *message;                  // message of this

    public:
        Complex_Value(Value *in_base_p, Class_Spec *in_p,
                      Environment *env_p);
                                        // constructor
        ~Complex_Value();               // destructor
        Object *generate_value(List<Variable> *vars_p, Error_Log *log_p);
                                        // generate value of this
        void get_quantifiers(List<Symbol> *symbols_p);
                                        // get quantifiers in this
};

#endif
```

```
// derclass.h

#ifndef _DERCLASS_H
#define _DERCLASS_H

class Derived_Class;

#include "classpec.h"
#include "environ.h"
#include "class.h"
#include "list.h"
#include "errorlog.h"
#include "object.h"
#include "result.h"
#include "reslist.h"
#include "derivati.h"

// Derived_Class is a derived class object
class Derived_Class : public Class
{
    private:
        List<Derivation> derivations; // derivations of this
        List<Symbol> superclasses;    // superclasses of this class

    public:
        Derived_Class(char *name, Class_Spec *in_p, Environment *env_p);
                                      // constructor
        ~Derived_Class(void);         // destructor
        List<Object> *run(Error_Log *log_p);
                                      // run this class
        int run_for_result(Result *result_p, Result_List *list_p,
                           Error_Log *log_p);
                                      // run class: add results to result
        void get_bases(Result_List *list_p, Error_Log *log_p);
                                      // get base classes to this
};


#endif
```

```
// derivati.h

#ifndef _DERIVATI_H
#define _DERIVATI_H

class Derivation;

#include "dermthd.h"
#include "list.h"
#include "variable.h"
#include "value.h"
#include "symbol.h"
#include "result.h"
#include "reslist.h"
#include "errorlog.h"
#include "classpec.h"
#include "environ.h"


// Derivation is the derivation of a derived class
class Derivation
{
    private:
        List<Variable> base_variables;// base variables of derivation
        List<Variable> superobjects;  // superobjects of object
        Value *guard_p;               // guard of this
        List<Derived_Method> methods; // the methods of this
        List<Symbol> quantifiers;     // quantifiers within this

        int get_superobject_and_test(int superobject_no,
                        int is_new,
                        Result *result_p, Result_List *list_p,
                        Error_Log *log_p);
                                      // test all possible objects
        int get_base_and_test(int index,
                        int is_new, Result *result_p, Result_List *list_p,
                        Error_Log *log_p);
                                      // test all possible objects
        int quantifiers_complete(Result_List *list_p);
                                      // see if quantifiers full-up

    public:
        Derivation(Class_Spec *in_p, Environment *env_p,
               List<Symbol> *superclasses_p);
                                      // constructor for this
        ~Derivation(void);            // destructor
        int run(Result *result_p, Result_List *list_p, Error_Log *log_p);
                                      // run the derivation
        Derived_Method *find_method(char *name, int arity);
                                      // find method for this
        int complete(Result_List *list_p);
                                      // see if this complete
        void get_bases(Result_List *list_p, Error_Log *log_p);
                                      // get base objects of this
};


#endif
```

```
// dermthd.h

#ifndef _DERMTHD_H
#define _DERMTHD_H

class Derived_Method;

#include "list.h"
#include "variable.h"
#include "method.h"
#include "value.h"
#include "classpec.h"
#include "environ.h"

// Derived_Method is a method of a derived class
class Derived_Method : public Method
{
    private:
        List<Variable> parameters;    // parameters to method
        Value *value_p;               // value of it

    public:
        Derived_Method(Class_Spec *in_p, Environment *env_p);
                                      // constructor for this
        ~Derived_Method(void);        // destructor
        List<Variable> *get_parameters(void);
                                      // parameters of this
        Value *value(void);           // get value of this
};

#endif
```

```
// derobj.h

#ifndef _DEROBJ_H
#define _DEROBJ_H

class Derived_Object;

#include <iostream.h>

#include "object.h"
#include "derivati.h"
#include "list.h"
#include "variable.h"
#include "errorlog.h"


extern char *const superobject_name;  // constant name for superobject

// Derived_Object is a member of a Derived Class
class Derived_Object : public Object
{
    private:
        Derivation *derivation_p;     // derivation of the object
        List<Variable> base_objects;  // base objects of object
        List<Object> superobjects;    // superobjects of object

    public:
        Derived_Object(Derivation * = NULL);
                                      // constructor for this
        void output(ostream &out, int indentation);
                                      // output this object
        void add_base(char *name, Object *value_p);
                                      // add a base object for this
        void add_superobject(Object *value_p);
                                      // add a superobject for this
        Object *clone(void);          // duplicate this
        int true(void);               // test if this is true
        Object *run_message(char *message, List<Object> *parms_p,
                            Error_Log *log_p);
                                      // run message against object
};


#endif
```

```
// dlclause.h

#ifndef _DLCLAUSE_H
#define _DLCLAUSE_H

class Dl_Clause;

#include "dlpred.h"
#include "list.h"
#include "dlspec.h"
#include "dlenv.h"
#include "dltuple.h"
#include "dlreslst.h"

// Dl_Clause is the class of a Clause
class Dl_Clause
{
    private:
        Dl_Predicate *predicate_p;    // predicate called by this
        List<char> parm_values;       // parameter values of this
        List<int> parm_variables;     // variables called by this

    public:
        Dl_Clause(Dl_Spec *in_p, Dl_Environment *env_p,
                        List<char> *vars_p);
                                      // constructor
        int is_complete(Dl_Result_List *results_p);
                                      // is the clause complete?
        void get_calls(Dl_Result_List *results_p);
                                      // get calls made by this
        List<int> *match(Dl_Tuple *tuple_p, List<char> *values_p);
                                      // does this match the tuple?
        Dl_Predicate *get_predicate(void);
                                      // get predicate for this
};

#endif
```

```
// dlenv.h

#ifndef _DLENV_H
#define _DLENV_H

class Dl_Environment;

#include "dlpred.h"

// Dl_Environment is the environment of a run
class Dl_Environment
{
    private:
        List<Dl_Predicate> predicates;// predicates of this

    public:
        void add_predicate(Dl_Predicate *pred_p);
                                        // add predicate to environment
        Dl_Predicate *find_predicate(char *name, int arity);
                                        // find a predicate
};

#endif
```

```
// dlog_db.h

#ifndef _DLOG_DB_H
#define _DLOG_DB_H

#include <iostream.h>

#include "dlenv.h"
#include "dlpred.h"

// Datalog_Database is a database for Datalog
class Datalog_Database
{
    private:
        Dl_Environment environment;   // the environment of it

    public:
        Datalog_Database(ostream &ErrorFile);
                                      // constructor
        Dl_Predicate *find_predicate(char *name, int arity);
                                      // find a specific predicate
};

#endif
```

```
// dlogclas.h

#ifndef _DLOGCLAS_H
#define _DLOGCLAS_H

class Datalog_Class;

#include "list.h"
#include "classpec.h"
#include "environ.h"
#include "gwclass.h"
#include "errorlog.h"
#include "dlpred.h"

// Datalog_Class() is a class from a Datalog database
class Datalog_Class : public Gateway_Class
{
    private:
        Dl_Predicate *predicate_p;    // predicate of the class

    public:
        Datalog_Class(char *name, Class_Spec *in, Environment *env_p);
                                      // constructor
        List<Object> *run(Error_Log *log_p);
                                      // run the class
};


#endif
```

```
// dlpred.h

#ifndef _DLPRED_H
#define _DLPRED_H

class Dl_Predicate;

#include "dlreslst.h"
#include "dlresult.h"
#include "errorlog.h"
#include "list.h"
#include "dltuple.h"
#include "dlrule.h"

// Dl_Predicate stores a predicate
class Dl_Predicate
{
    private:
        char *name;                     // name of the predicate
        int arity;                      // arity of the predicate
        List<Dl_Rule> rules;            // rules for the predicate

    public:
        Dl_Predicate(char *in_name, int in_arity);
                                        // constructor
        ~Dl_Predicate(void);            // destructor
        void add_rule(Dl_Rule *rule_p);
                                        // add rule to predicate
        char *get_name(void);           // get name of predicate
        int get_arity(void);            // get arity of predicate
        List<Dl_Tuple> *run(Error_Log *log_p);
                                        // run the predicate
        int run_for_result(Dl_Result *dst_p, Dl_Result_List *results_p,
                int new_data, Error_Log *log_p);
                                        // run the predicate for a round
        void get_calls(Dl_Result_List *results_p);
                                        // find all predicates called by
this
};

#endif
```

```
// dlreslst.h

#ifndef _DLRESLST_H
#define _DLRESLST_H

class Dl_Result_List;

#include "list.h"
#include "dlresult.h"
#include "dlpred.h"
#include "errorlog.h"

// Dl_Result_List maintains lists of results for Datalog
class Dl_Result_List
{
    private:
        List<Dl_Result> results;      // results in list

    public:
        void add_predicate(Dl_Predicate *pred_p);
                                      // add predicate to list
        Dl_Result *find_result(char *name, int arity);
                                      // find result for a predicate
        Dl_Result *find_result(Dl_Predicate *pred_p);
                                      // find result for a predicate
        void run(Error_Log *log_p);   // fill out all results
};

#endif
```

```
// dlresult.h

#ifndef _DLRESULT_H
#define _DLRESULT_H

class Dl_Result;

#include "dlpred.h"
#include "list.h"
#include "dltuple.h"

// class Dl_Result stores results for a predicate
class Dl_Result
{
    private:
        Dl_Predicate *predicate_p;
        List<Dl_Tuple> tuples;
        List<Dl_Tuple> pending_tuples;
        int complete;
        int new_tuple_offset;

    public:
        Dl_Result(Dl_Predicate *pred_p);
                                        // constructor
        List<Dl_Tuple> *extract(void);
                                        // get tuples and remove them
        List<Dl_Tuple> *get_tuples(void);
                                        // get tuples w/o removing it
        void add_tuple(Dl_Tuple *value_p);
                                        // add tuple to list
        Dl_Predicate *get_predicate(void);
                                        // get a predicate
        void end_cycle(void);           // checkpoint tuples added
        void mark_complete(void);       // mark this as complete
        int is_complete(void);          // is this complete?
        int is_new(int index);          // see if this is new
};

#endif
```

```
// dlrule.h

#ifndef _DLRULE_H
#define _DLRULE_H

class Dl_Rule;

#include "dlresult.h"
#include "dlreslst.h"
#include "list.h"
#include "dlclause.h"
#include "errorlog.h"
#include "dlspec.h"
#include "dlenv.h"

// Dl_Rule is a Datalog rule
class Dl_Rule
{
    private:
        List<Dl_Clause> clauses;      // clauses of the rule
        List<char> values;            // fixed values of the rule
        char *name;                   // name of the rule's predicate
        int arity;                    // arity of the rule
        int test_all(int index, int data_new, Dl_Result *result_p,
                Dl_Result_List *results_p, List<char> *vals_p,
                Error_Log *log_p);    // run the test

    public:
        Dl_Rule(Dl_Spec *in_p, Dl_Environment *env_p);
                                      // constructor
        ~Dl_Rule(void);               // destructor
        char *get_name(void);         // get name of rule
        int get_arity(void);          // get arity of rule
        int run_for_result(Dl_Result *dst_p, Dl_Result_List *results_p,
                int new_data, Error_Log *log_p);
                                      // run the rule
        int is_complete(Dl_Result_List *results_p);
                                      // is the rule complete?
        void get_calls(Dl_Result_List *results_p);
                                      // get calls made by this
};

#endif
```

```
// dlspec.h

#ifndef _DLSPEC_H
#define _DLSPEC_H

#include <fstream.h>

class Dl_Spec;

#include "statetbl.h"
#include "errorlog.h"

// token_type is the enumerated list of types of tokens
enum dl_token_type {Dl_End_of_Input,
                    Dl_Name,
                    Dl_Open_Paren,
                    Dl_Close_Paren,
                    Dl_String,
                    Dl_Comma,
                    Dl_Is_Defined,
                    Dl_Semicolon,
                    Dl_Integer};

// Dl_Spec defines a class specification
class Dl_Spec
{
    private:
        char *filename;              // name of input file
        ifstream *token_input_p;     // token input
        dl_token_type current_token; // current token type
        char *token_value_buffer;    // buffer for current token
        char *token_value_p;         // ptr to current token
        State_Table state_table;     // state table for this
        Error_Log error_log;         // list of errors

        void fill_in_state_table(void);
                                     // fill in state table
        void skip_blanks(void);      // skip over blanks

    public:
        Dl_Spec(char *filename);     // construct it
        ~Dl_Spec(void);              // destruct it
        dl_token_type next_token(void);
                                     // get the next token
        void skip_token(void);       // skip to the next token
        char *get_value(void);       // get the current token value
        void log_error(char *);      // record an error
        int is_errors(void);         // get error flag
        Error_Log *get_errors(void); // get error log
        char *text(void);            // get full text of input
};

#endif
```

```
// dltuple.h

#ifndef _DLTUPLE_H
#define _DLTUPLE_H

#include "list.h"

// Dl_Tuple is a tuple returned by a Datalog predicate
class Dl_Tuple
{
    private:
        List<char> values;

    public:
        void add_value(char *value); // add member to tuple
        char *operator[](int offset);// get a member of the tuple
        int size(void);              // get size of tuple
};


#endif
```

```
// editwin.h

#ifndef _EDITWIN_H
#define _EDITWIN_H

#include <owl\framewin.h>
#include <owl\edit.h>
#include <owl\dialog.h>

#include "roxwin.h"

// Edit_Window is an editor window
class Edit_Window : public TDialog
{
    public:
        Edit_Window(TWindow *, char *);

    protected:
        void SetupWindow();          // override normal SetupWindow()

    private:
        TEdit *edit_box;
        char filename[200];          // file being edited
        void handle_save_exit();     // handler for save button
        DECLARE_RESPONSE_TABLE(Edit_Window);
                                     // response table for this
};


#endif
```

```
// environ.h

#ifndef _ENVIRON_H
#define _ENVIRON_H

#include <iostream.h>

class Environment;

#include "list.h"
#include "symbol.h"
#include "pdox_db.h"
#include "dlog_db.h"

// Environment is the compile-time and system environment
class Environment
{
    private:
        List<Symbol> symbols;         // list of known symbols
        Paradox_Database *pdox_database_p;
                                      // Paradox Database
        Datalog_Database *dlog_database_p;
                                      // Datalog Database

    public:
        Environment(ostream &errors);// construct the environment
        ~Environment(void);           // destruct this
        Symbol *find_symbol(char *name);
                                      // find a symbol
        void add_symbol(Symbol *symbol_p);
                                      // add a symbol
        Paradox_Database *paradox_database(void);
                                      // get the Paradox database
        Datalog_Database *datalog_database(void);
                                      // get the Datalog database
};

#endif
```

```
// errorlog.h

#ifndef _ERRORLOG_H
#define _ERRORLOG_H

#include <iostream.h>
//#include <owl/applicat.h>

class Error_Log;

#include "list.h"

// Error_Log is a log of errors
class Error_Log
{
    private:
        List<char> errors;          // all of the errors in log

    public:
        int is_errors();            // are there errors?
        void display_errors(ostream&);
                                    // display all errors
        void log(char *error);      // add an error
};


#endif
```

```
// filedisp.h

#ifndef _FILEDISP_H
#define _FILEDISP_H

#include <owl\applicat.h>
#include <owl\edit.h>
#include <owl\dialog.h>

class File_Display : public TDialog
{
    protected:
        void SetupWindow();          // override normal SetupWindow()

    private:
        TEdit *edit_box;
        char *display_buffer;        // data being displayed
        DECLARE_RESPONSE_TABLE(File_Display);
                                     // response table for this

    public:
        File_Display(TWindow *parent, char *label, char *buffer);
                                     // create the window
        ~File_Display(void);         // destructor
};

#endif
```

```
// gwclass.h

#ifndef _GWCLASS_H
#define _GWCLASS_H

class Gateway_Class;

#include "classpec.h"
#include "environ.h"
#include "class.h"
#include "list.h"
#include "method.h"
#include "symbol.h"

// Gateway_Class is the class of a Gateway_Object
class Gateway_Class : public Class
{
    protected:
        List<Method> methods;        // the methods of the class
        List<Symbol> superclasses;   // superclasses to this class

    public:
        Gateway_Class(char *name, Class_Spec *in, Environment *env_p);
                                     // constructor
        virtual ~Gateway_Class(void);// destructor
        void read_gateway(Class_Spec *in, Environment *env_p);
                                     // read in body of class
};

Gateway_Class *read_gateway_class(char *, Class_Spec *, Environment *);

#endif
```

```
// list.h

#ifndef _LIST_H
#define _LIST_H

#include <stdlib.h>

// Base_List is a list of things
class Base_List
{
    private:
        int n_entries;              // number of things on list
        int allocated_size;         // size of allocated stuff
        void **entries_pp;          // things on list
    public:
        Base_List(void);            // constructor
        ~Base_List(void);           // destructor
        int size(void);             // get size of list
        void *operator[] (int);     // get entry off of list
        void add(void *);           // add entry to list
        void merge(Base_List *mergelist);
                                    // merge in second list
        void remove(int index);     // delete entry from list
        void set_entry(int index, void *data);
                                    // set the value of a specific entry
        void clear(int index);      // clear a specified entry
        void empty(void);           // empty the list
};


// List is a template of a thing-list
template <class Data> class List
{
    private:
        Base_List data;             // the list to do the work

    public:
        List() {}                   // constructor
        ~List()                     // destructor
        {
            int i;
            for(i=0; i<data.size(); ++i)
            {
                if(data[i] != NULL)
                {
                    delete (Data *)data[i];
                }
            }
        }
        int size()                  // get size of list
        {
            return(data.size());
        }
        Data *operator[](int index)
        {
            return((Data *)data[index]);
        }                           // get an item from the list
        void add(Data *item)
```

```
        {
            data.add((void *)item);
        }                               // add item to list
        void merge(List<Data> *mergelist)
        {
            data.merge(&mergelist->data);
        }                               // merge in elements of mergelist
        void remove(int index)
        {
            data.remove(index);
        }                               // remove item from list
        void set_entry(int index, Data *value)
        {
            clear(index);
            data.set_entry(index, (void *)value);
        }                               // set value of an entry
        void clear(int index)
        {
            if(data[index] != NULL)
                delete (Data *)data[index];
            data.clear(index);
        }
        void empty()
        {
            data.empty();
        }
};

#endif
```

```
// method.h

#ifndef _METHOD_H
#define _METHOD_H

#include <stddef.h>

class Method;

#include "classspec.h"
#include "environ.h"
#include "symbol.h"

// Method is the class for a base method - as for Gateway classes
class Method
{
    private:
        Symbol *method_type_p;        // type of the method
        char *method_name;            // name of method
        int method_arity;             // arity of method

    public:
        Method(Class_Spec *in, Environment *env_p);
                                      // construct from specification
        Method(Symbol *type_p = NULL, char *name = NULL, int arity = 0);
                                      // construct from nothing
        virtual ~Method(void);        // destruct
        void set_name(char *in_name);// set the name of the method
        void set_arity(int in_arity);// set its arity
        void set_type(Symbol *type_p);
                                      // set its type
        char *name(void);             // get its name
        int arity(void);              // get its arity
        Symbol *type(void);           // get its type
};


#endif
```

```
// object.h

#ifndef _OBJECT_H
#define _OBJECT_H

#include <iostream.h>

class Object;

#include "list.h"
#include "errorlog.h"

// Object is an object: this is a virtual class
class Object
{
    public:
        virtual ~Object(void);       // destructor of this
        virtual void output(ostream &out, int indentation = 0) = 0;
                                     // write out an object
        virtual Object *clone(void) = 0;
                                     // clone this object
        virtual int true(void) = 0;  // is the object true?
        virtual Object *run_message(char *msg, List<Object> *parms_p,
                        Error_Log *log_p) = 0;
                                     // run a message
};

// the following outputs an object
ostream &operator<< (ostream &out, Object &object);

#endif
```

```
// pdox_db.h

#ifndef _PDOX_DB_H
#define _PDOX_DB_H

#include <bdatabas.h>
#include <bengine.h>

// Paradox_Database() tracks all that is needed for a Paradox Database
class Paradox_Database
{
    private:
        BEngine *engine_p;          // the Paradox engine
        BDatabase *database_p;      // the database for this

    public:
        Paradox_Database(void);     // construct the database
        ~Paradox_Database(void);    // destruct the database
        BEngine *get_engine(void);  // get engine
        BDatabase *get_database(void);
                                    // get database
};

#endif
```

```
// pdoxclas.h

#ifndef _PDOXCLAS_H
#define _PDOXCLAS_H

class Paradox_Class;

#include <bcursor.h>
#include <brecord.h>

#include "list.h"
#include "classpec.h"
#include "environ.h"
#include "gwclass.h"
#include "errorlog.h"
#include "relation.h"
#include "pdox_db.h"

// a Paradox_Class gets stuff from a Paradox database
class Paradox_Class : public Gateway_Class
{
    private:
        Paradox_Database *database_p;// paradox database structure
        BCursor *cursor_p;           // cursor for field
        BRecord *record_p;           // record to use
        Relation *read_record(Error_Log *log_p);
                                     // read in a record
        char *get_blob(char *name, int number, Error_Log *log_p);
                                     // read in a blob
        char *Paradox_Class::get_field(char *name,
                                int number,
                                PXFieldType type,
                                int fld_len,
                                Error_Log *log_p);
                                     // read in a non-blob field
        void open_cursor(Error_Log *log_p);
                                     // open the cursor
        void close_cursor(void);     // close the cursor

    public:
        Paradox_Class(char *name, Class_Spec *in, Environment *env_p);
                                     // constructor
        ~Paradox_Class();            // destructor
        List<Object> *run(Error_Log *log_p);
                                     // get members of class
};


#endif
```

```
// quanval.h

#ifndef _QUANVAL_H
#define _QUANVAL_H

class Quantified_Value;

#include "value.h"
#include "list.h"
#include "variable.h"
#include "errorlog.h"
#include "classpec.h"
#include "environ.h"
#include "symbol.h"

// Quantifier_Type is the type of a quantifier
enum Quantifier_Type { Q_Forall, Q_Exists };

// Quantified_Value is a FORALL or EXISTS value
class Quantified_Value : public Value
{
    private:
        Symbol *boolean_class_p;      // symbol for boolean class
        enum Quantifier_Type quantifier;
                                      // type of quantifier
        Variable *var_p;              // the variable of this
        Value *body_p;                // body of the value

        int run_forall(List<Variable> *vars_p, Error_Log *log_p,
                       List<Object> *objects_p);
                                      // run a FORALL value
        int run_exists(List<Variable> *vars_p, Error_Log *log_p,
                       List<Object> *objects_p);
                                      // run an EXISTS value

    public:
        Quantified_Value(Class_Spec *in_p, Environment *env_p);
                                      // constructor
        ~Quantified_Value(void);      // destructor
        Object *generate_value(List<Variable> *vars_p, Error_Log *log_p);
                                      // generate value of this
        void get_quantifiers(List<Symbol> *symbols_p);
                                      // get quantifiers in this
};

#endif
```

```
// relation.h

#ifndef _RELATION_H
#define _RELATION_H

#include <iostream.h>

class Relation;

#include "object.h"
#include "list.h"
#include "variable.h"
#include "errorlog.h"

// Relation is a relational object (ie, a tuple)
class Relation : public Object
{
    private:
        List<Variable> fields;       // fields of the relation

    public:
        void output(ostream &out, int indentation);
                                     // output the relation
        Object *clone(void);         // duplicate this
        void add_field(char *name, Object *value_p);
                                     // add a field to the relation
        int true(void);              // is this true?
        Object *run_message(char *msg, List<Object> *parms_p,
                        Error_Log *log_p);
                                     // run a message
};


#endif
```

```
// reslist.h

#ifndef _RESLIST_H
#define _RESLIST_H

class Result_List;

#include "list.h"
#include "result.h"
#include "class.h"
#include "object.h"
#include "errorlog.h"

// Result_List lists results of a derived program run
class Result_List
{
    private:
        List<Result> results;        // all results for this

    public:
        void add_class(Class *class_p, Error_Log *log_p);
                                     // add class to this
        List<Object> *extract(Class *class_p);
                                     // get objects of a class
        void run(Error_Log *log_p);  // run the program of this
        Result *find_result(Class *class_p);
                                     // get one result from list
};


#endif
```

```
// result.h

#ifndef _RESULT_H
#define _RESULT_H

class Result;

#include "class.h"
#include "list.h"
#include "object.h"

// class Result is a result of a search
class Result
{
    private:
        Class *result_class_p;        // class of result
        List<Object> objects;         // objects of this result
        List<Object> pending_objects;// objects not on objects list yet
        int is_complete;              // is this complete?
        int new_object_offset;       // offset to objects since checkpoint

    public:
        Result(Class *class_p);       // constructor
        List<Object> *extract(void); // get object list from this
        Class *get_class(void);       // get class of this
        List<Object> *get_objects(void);
                                      // get object list but no extract
        int complete(void);           // is this complete?
        void end_cycle(void);         // checkpoint this
        void add(Object *);           // add object to this
        void add(List<Object> *);     // add list of objects to this
        void mark_complete();         // mark this as complete
        int is_new(int index);        // is an object new?
};

#endif
```

```
/***************************************************************************

roxwin.h

produced by Borland Resource Workshop

***************************************************************************/

#define DIALOG_4                    4
#define CLASS_LIST_WIN              1
#define FILE_DISPLAY                3
#define FILE_DISPLAY_TEXT           101
#define DIALOG_5                    5
#define EDIT_WINDOW                 2
#define DIALOG_3                    3
#define ID_EDIT_BOX                 101
#define IDC_EXIT                    102
#define IDC_QUIT                    103
#define MENU_1                      1
#define CM_HELPABOUT                24346
#define CM_HELPUSING_HELP           24345
#define CM_HELPPROCEDURES           24344
#define CM_HELPCOMMANDS             24343
#define CM_HELPKEYBOARD             24342
#define CM_HELPINDEX                24341
#define CM_EDITPASTE1               24328
#define CM_EDITCOPY1                24327
#define CM_EDITCUT1                 24326
#define CM_EDITUNDO1                24325
#define CM_FILEEXIT                 24338
#define CM_FILEPRINTER_SETUP        24337
#define CM_FILEPAGE_SETUP           24336
#define CM_FILEPRINT                24335
#define CM_FILESAVEAS               24334
#define CM_FILESAVE                 24333
#define CM_FILEOPEN                 24332
#define CM_FILENEW                  24331
#define DIALOG_2                    2
#define ID_CLASS_LIST               101
#define ID_RUN                      103
#define ID_EXIT                     104
#define ID_EDIT                     205
#define DIALOG_1                    1
#define DB_LIST_WIN                 1
#define ID_QUIT                     105
#define ID_SAVE                     106
#define ID_LOAD                     107
#define ID_DELETE                   104
#define ID_NEW                      102
```

```
// statetbl.h

#ifndef _STATETBL_H
#define _STATETBL_H

extern const no_state;                  // there was no state
extern const no_type;                   // there is no type for it

// State_Table maintains a state table
class State_Table
{
    private:
        int **states_p;                 // list of states
        int *types_p;                   // list of types of states
        int state_ptrs_allocated;       // number of state ptrs

        void expand_states_p(int);      // expand states p
        void create_state(int);         // create a new state

    public:
        State_Table(void);              // construct the beastie
        ~State_Table(void);             // destruct it
        void add_transition(int, char, int);
                                        // add transition to state
        void set_type(int, int);        // set type of state
        int get_next_state(int, char);
                                        // find what next state is
        int get_type(int);              // get type of a state
};

#endif
```

```
// symbol.h

#ifndef _SYMBOL_H
#define _SYMBOL_H

class Symbol;

#include "class.h"

// Symbol links a name to a class
class Symbol
{
    private:
        Class *value_p;              // the class of this
        char *symbol_name;           // the name of this

    public:
        Symbol(char *in_name, Class *class_p = NULL);
                                     // construct this
        ~Symbol(void);               // destruct it
        void set_value(Class *class_p);
                                     // set the value of it
        Class *value(void);          // get the value
        void clear_value(void);      // clear the value
        char *name(void);            // get name of this
};


#endif
```

```
// utils.h

#ifndef _UTILS_H
#define _UTILS_H


#include <iostream.h>

// the following are utility functions
void indent(ostream &out, int n);     // indent n spaces


#endif
```

```
// value.h

#ifndef _VALUE_H
#define _VALUE_H

class Value;

#include "object.h"
#include "list.h"
#include "variable.h"
#include "errorlog.h"
#include "symbol.h"
#include "classpec.h"
#include "environ.h"

// Value is the parent class of all value types.
// This is a virtual class.
class Value
{
    public:
        virtual Object *generate_value(List<Variable> *, Error_Log *) = 0;
                                        // generate value
        virtual void get_quantifiers(List<Symbol> *);
                                        // get quantifiers and add to list
        virtual ~Value(void);           // destructor for this
};

// the following is related but not this
Value *read_value(Class_Spec *in_p, Environment *env_p);

#endif
```

```
// variable.h

#ifndef _VARIABLE_H
#define _VARIABLE_H

#include <iostream.h>

class Variable;

#include "object.h"
#include "environ.h"
#include "symbol.h"
#include "classpec.h"

// Variable is the class of a variable
class Variable
{
    private:
        Symbol *var_type_p;          // type of the variable
        char *var_name;              // its name
        Object *var_value_p;         // its value

    public:
        Variable(char *name, Object *value_p, Symbol *in_type_p = NULL);
                                     // construct from data
        Variable(Class_Spec *in, Environment *env_p);
                                     // construct from spec
        ~Variable();                 // destructor
        void set_value(Object *new_value_p);
                                     // set value of this
        void clear_value(void);      // clear value of it
        void remove_value(void);     // remove value without destructing
        Symbol *type(void);          // get its type
        char *name(void);            // get its name
        Object *value(void);         // get its value
        Variable *clone(void);       // duplicate this
        void output(ostream &out, int indent = 0);
                                     // output this
};

// output a variable
ostream &operator<< (ostream &out, Variable &var);

#endif
```

```
// varval.h

#ifndef _VARVAL_H
#define _VARVAL_H

class Variable_Value;

#include "value.h"
#include "classpec.h"
#include "environ.h"
#include "list.h"
#include "errorlog.h"

class Variable_Value : public Value
{
    private:
        char *variable_name;          // name of the variable

    public:
        Variable_Value(Class_Spec *in_p, Environment *env_p);
                                      // constructor
        ~Variable_Value();            // destructor
        Object *generate_value(List<Variable> *, Error_Log *);
                                      // generate value of this
};

#endif
```

D.2. Roxanne Source Files.

```
// baseimps.cpp

// Contains the code for implementations of methods for the basic
// built-in classes.

#include <stddef.h>
#include <string.h>
#include <stdio.h>

#include "environ.h"
#include "baseimps.h"
#include "object.h"
#include "list.h"
#include "bin_obj.h"
#include "symbol.h"
#include "binclass.h"

// the following are constants for TRUE and FALSE
char *const true_value = "TRUE";
char *const false_value = "FALSE";

// the following are class names
char *const boolean_class_name = "BOOLEAN";
char *const integer_class_name = "INTEGER";
char *const string_class_name = "STRING";

// the following contain the classes for this
static Builtin_Class *integer_class_p = NULL;
static Builtin_Class *boolean_class_p = NULL;
static Builtin_Class *string_class_p = NULL;

// the following are the labels for messages
const int max_operator_len = 11;
char *const plus_message = "operator+";
char *const minus_message = "operator-";
char *const asterisk_message = "operator*";
char *const slash_message = "operator/";
char *const less_than_message = "operator<";
char *const greater_than_message = "operator>";
char *const equals_message = "operator=";
char *const not_equals_message = "operator<>";
char *const less_or_equals_message = "operator<=";
char *const greater_or_equals_message = "operator>=";
char *const not_message = "operatorNOT";
char *const and_message = "operatorAND";
char *const or_message = "operatorOR";

// integer_plus() adds integers
static Object *integer_plus(Object *base_p, List<Object> *parms_p)
{
    char value[100];                // value buffer

    // confirm arity
    if(parms_p->size() != 1)
        return(NULL);
```

```
    // get the operands
    Builtin_Object *loperand_p = (Builtin_Object *)base_p;
    Builtin_Object *roperand_p = (Builtin_Object *)(*parms_p)[0];

    // if either are null...
    if(loperand_p == NULL || roperand_p == NULL)
        return(NULL);

    sprintf(value, "%d",
        atoi(loperand_p->value()) + atoi(roperand_p->value()));

    // construct an object for this
    return(new Builtin_Object(integer_class_p, value));
}

// integer_minus() subtracts integers
static Object *integer_minus(Object *base_p, List<Object> *parms_p)
{
    char value[100];                // value buffer

    // confirm arity
    if(parms_p->size() != 1)
        return(NULL);

    // get the operands
    Builtin_Object *loperand_p = (Builtin_Object *)base_p;
    Builtin_Object *roperand_p = (Builtin_Object *)(*parms_p)[0];

    // if either are null...
    if(loperand_p == NULL || roperand_p == NULL)
        return(NULL);

    sprintf(value, "%d",
        atoi(loperand_p->value()) - atoi(roperand_p->value()));

    // construct an object for this
    return(new Builtin_Object(integer_class_p, value));
}

// integer_times() multiplies integers
static Object *integer_times(Object *base_p, List<Object> *parms_p)
{
    char value[100];                // value buffer

    // confirm arity
    if(parms_p->size() != 1)
        return(NULL);

    // get the operands
    Builtin_Object *loperand_p = (Builtin_Object *)base_p;
    Builtin_Object *roperand_p = (Builtin_Object *)(*parms_p)[0];

    // if either are null...
    if(loperand_p == NULL || roperand_p == NULL)
        return(NULL);

    sprintf(value, "%d",
        atoi(loperand_p->value()) * atoi(roperand_p->value()));
```

```
    // construct an object for this
    return(new Builtin_Object(integer_class_p, value));
}

// integer_divide() divides integers
static Object *integer_divide(Object *base_p, List<Object> *parms_p)
{
    char value[100];                // value buffer

    // confirm arity
    if(parms_p->size() != 1)
        return(NULL);

    // get the operands
    Builtin_Object *loperand_p = (Builtin_Object *)base_p;
    Builtin_Object *roperand_p = (Builtin_Object *)(*parms_p)[0];

    // if either are null...
    if(loperand_p == NULL || roperand_p == NULL)
        return(NULL);

    sprintf(value, "%d",
       atoi(loperand_p->value()) / atoi(roperand_p->value()));

    // construct an object for this
    return(new Builtin_Object(integer_class_p, value));
}

// integer_less_than() tests less-then relationship
static Object *integer_less_than(Object *base_p, List<Object> *parms_p)
{
    char *value;                // value buffer

    // confirm arity
    if(parms_p->size() != 1)
        return(NULL);

    // get the operands
    Builtin_Object *loperand_p = (Builtin_Object *)base_p;
    Builtin_Object *roperand_p = (Builtin_Object *)(*parms_p)[0];

    // if either are null...
    if(loperand_p == NULL || roperand_p == NULL)
        return(NULL);

    if(atoi(loperand_p->value()) < atoi(roperand_p->value()))
        value = true_value;
    else
        value = false_value;

    // construct an object for this
    return(new Builtin_Object(boolean_class_p, value));
}

// integer_greater_than() tests greater-then relationship
static Object *integer_greater_than(Object *base_p, List<Object> *parms_p)
{
    char *value;                // value buffer
```

```
    // confirm arity
    if(parms_p->size() != 1)
        return(NULL);

    // get the operands
    Builtin_Object *loperand_p = (Builtin_Object *)base_p;
    Builtin_Object *roperand_p = (Builtin_Object *)(*parms_p)[0];

    // if either are null...
    if(loperand_p == NULL || roperand_p == NULL)
        return(NULL);

    if(atoi(loperand_p->value()) > atoi(roperand_p->value()))
        value = true_value;
    else
        value = false_value;

    // construct an object for this
    return(new Builtin_Object(boolean_class_p, value));
}

// integer_equals() tests equality
static Object *integer_equals(Object *base_p, List<Object> *parms_p)
{
    char *value;                // value buffer

    // confirm arity
    if(parms_p->size() != 1)
        return(NULL);

    // get the operands
    Builtin_Object *loperand_p = (Builtin_Object *)base_p;
    Builtin_Object *roperand_p = (Builtin_Object *)(*parms_p)[0];

    // if either are null...
    if(loperand_p == NULL || roperand_p == NULL)
        return(NULL);

    if(atoi(loperand_p->value()) == atoi(roperand_p->value()))
        value = true_value;
    else
        value = false_value;

    // construct an object for this
    return(new Builtin_Object(boolean_class_p, value));
}

// integer_less_equals() tests <= relationship
static Object *integer_less_equals(Object *base_p, List<Object> *parms_p)
{
    char *value;                // value buffer

    // confirm arity
    if(parms_p->size() != 1)
        return(NULL);

    // get the operands
    Builtin_Object *loperand_p = (Builtin_Object *)base_p;
```

```
    Builtin_Object *roperand_p = (Builtin_Object *)(*parms_p)[0];

    // if either are null...
    if(loperand_p == NULL || roperand_p == NULL)
        return(NULL);

    if(atoi(loperand_p->value()) <= atoi(roperand_p->value()))
        value = true_value;
    else
        value = false_value;

    // construct an object for this
    return(new Builtin_Object(boolean_class_p, value));
}

// integer_greater_equals() tests >= relationship
static Object *integer_greater_equals(Object *base_p,
                  List<Object> *parms_p)
{
    char *value;                // value buffer

    // confirm arity
    if(parms_p->size() != 1)
        return(NULL);

    // get the operands
    Builtin_Object *loperand_p = (Builtin_Object *)base_p;
    Builtin_Object *roperand_p = (Builtin_Object *)(*parms_p)[0];

    // if either are null...
    if(loperand_p == NULL || roperand_p == NULL)
        return(NULL);

    if(atoi(loperand_p->value()) >= atoi(roperand_p->value()))
        value = true_value;
    else
        value = false_value;

    // construct an object for this
    return(new Builtin_Object(boolean_class_p, value));
}

// integer_not_equals() tests <> relationship
static Object *integer_not_equals(Object *base_p, List<Object> *parms_p)
{
    char *value;                // value buffer

    // confirm arity
    if(parms_p->size() != 1)
        return(NULL);

    // get the operands
    Builtin_Object *loperand_p = (Builtin_Object *)base_p;
    Builtin_Object *roperand_p = (Builtin_Object *)(*parms_p)[0];

    // if either are null...
    if(loperand_p == NULL || roperand_p == NULL)
        return(NULL);
```

```
    if(atoi(loperand_p->value()) != atoi(roperand_p->value()))
        value = true_value;
    else
        value = false_value;

    // construct an object for this
    return(new Builtin_Object(boolean_class_p, value));
}


// boolean_and() tests AND condition
static Object *boolean_and(Object *base_p, List<Object> *parms_p)
{
    char *value;                // value of stuff

    // confirm arity
    if(parms_p->size() != 1)
        return(NULL);

    // do the test
    if(base_p != NULL && !base_p->true())
        value = false_value;
    else if((*parms_p)[0] != NULL && !(*parms_p)[0]->true())
        value = false_value;
    else if(base_p == NULL || (*parms_p)[0] == NULL)
        return(NULL);
    else if(base_p->true() && (*parms_p)[0]->true())
        value = true_value;
    else
        value = false_value;

    // construct an object for this
    return(new Builtin_Object(boolean_class_p, value));
}

// boolean_or() tests OR condition
static Object *boolean_or(Object *base_p, List<Object> *parms_p)
{
    char *value;                // value of stuff

    // confirm arity
    if(parms_p->size() != 1)
        return(NULL);

    // do the test
    if(base_p != NULL && base_p->true())
        value = true_value;
    else if((*parms_p)[0] != NULL && (*parms_p)[0]->true())
        value = true_value;
    else if(base_p == NULL || (*parms_p)[0] == NULL)
        return(NULL);
    else if(base_p->true() || (*parms_p)[0]->true())
        value = true_value;
    else
        value = false_value;

    // construct an object for this
    return(new Builtin_Object(boolean_class_p, value));
```

```
}

// boolean_not() tests NOT condition
static Object *boolean_not(Object *base_p, List<Object> *parms_p)
{
    char *value;                // value of stuff

    // confirm arity
    if(parms_p->size() != 0)
        return(NULL);

    // test for null
    if(base_p == NULL)
        return(NULL);

    // do the test
    if(!base_p->true())
        value = true_value;
    else
        value = false_value;

    // construct an object for this
    return(new Builtin_Object(boolean_class_p, value));
}

// string_equals() tests STRING equality
static Object *string_equals(Object *base_p, List<Object> *parms_p)
{
    char *value;                // value of stuff

    // confirm arity
    if(parms_p->size() != 1)
        return(NULL);

    // get the operands
    Builtin_Object *loperand_p = (Builtin_Object *)base_p;
    Builtin_Object *roperand_p = (Builtin_Object *)(*parms_p)[0];

    // test for null
    if(loperand_p == NULL || roperand_p == NULL)
        return(NULL);

    // do the test
    if(strcmp(loperand_p->value(), roperand_p->value()) == 0)
        value = true_value;
    else
        value = false_value;

    // construct an object for this
    return(new Builtin_Object(boolean_class_p, value));
}

// string_not_equals() tests STRING non-equality
static Object *string_not_equals(Object *base_p, List<Object> *parms_p)
{
    char *value;                // value of stuff

    // confirm arity
```

```
    if(parms_p->size() != 1)
        return(NULL);

    // get the operands
    Builtin_Object *loperand_p = (Builtin_Object *)base_p;
    Builtin_Object *roperand_p = (Builtin_Object *)(*parms_p)[0];

    // test for null
    if(loperand_p == NULL || roperand_p == NULL)
        return(NULL);

    // do the test
    if(strcmp(loperand_p->value(), roperand_p->value()) != 0)
        value = true_value;
    else
        value = false_value;

    // construct an object for this
    return(new Builtin_Object(boolean_class_p, value));
}


// setup_base_classes() sets up all base classes
void setup_base_classes(Environment *env_p)
{
    // construct the classes
    boolean_class_p = new Builtin_Class(boolean_class_name, env_p);
    integer_class_p = new Builtin_Class(integer_class_name, env_p);
    string_class_p = new Builtin_Class(string_class_name, env_p);

    // set symbol values for them
    Symbol *boolean_symbol_p;
    boolean_symbol_p = env_p->find_symbol(boolean_class_name);
    boolean_symbol_p->set_value(boolean_class_p);

    Symbol *string_symbol_p;
    string_symbol_p = env_p->find_symbol(string_class_name);
    string_symbol_p->set_value(string_class_p);

    Symbol *integer_symbol_p;
    integer_symbol_p = env_p->find_symbol(integer_class_name);
    integer_symbol_p->set_value(integer_class_p);

    // add methods to the classes
    integer_class_p->add_method(plus_message, 1, integer_symbol_p,
                        integer_plus);
    integer_class_p->add_method(minus_message, 1, integer_symbol_p,
                        integer_minus);
    integer_class_p->add_method(asterisk_message, 1, integer_symbol_p,
                        integer_times);
    integer_class_p->add_method(slash_message, 1, integer_symbol_p,
                        integer_divide);
    integer_class_p->add_method(equals_message, 1, boolean_symbol_p,
                        integer_equals);
    integer_class_p->add_method(less_than_message, 1, boolean_symbol_p,
                        integer_less_than);
    integer_class_p->add_method(greater_than_message, 1, boolean_symbol_p,
```

```
                        integer_greater_than);
    integer_class_p->add_method(less_or_equals_message, 1,
                    boolean_symbol_p, integer_less_equals);
    integer_class_p->add_method(greater_or_equals_message, 1,
                    boolean_symbol_p, integer_greater_equals);
    integer_class_p->add_method(not_equals_message, 1, boolean_symbol_p,
                        integer_not_equals);
    boolean_class_p->add_method(and_message, 1, boolean_symbol_p,
                        boolean_and);
    boolean_class_p->add_method(or_message, 1, boolean_symbol_p,
                        boolean_or);
    boolean_class_p->add_method(not_message, 0, boolean_symbol_p,
                        boolean_not);
    string_class_p->add_method(equals_message, 1, boolean_symbol_p,
                        string_equals);
    string_class_p->add_method(not_equals_message, 1, boolean_symbol_p,
                        string_not_equals);
}
```

```
// bin_obj.cpp

// Contains the class for Builtin_Object, which is an object
// that is a member of a built-in class.

#include <iostream.h>
#include <string.h>
#include <stddef.h>

#include "bin_obj.h"
#include "binclass.h"
#include "object.h"
#include "list.h"
#include "binmthd.h"
#include "baseimps.h"
#include "errorlog.h"

// Builtin_Object() constructs one of these guys
Builtin_Object::Builtin_Object(Builtin_Class *in_class_p, char *in_value)
{
    class_p = in_class_p;

    if(in_value != NULL)
    {
        value_p = new char[strlen(in_value) + 1];
        strcpy(value_p, in_value);
    }
    else
        value_p = NULL;
}


// ~Builtin_Object() destructs this
Builtin_Object::~Builtin_Object()
{
    if(value_p != NULL)
        delete[] value_p;
}



// output() outputs this class
void Builtin_Object::output(ostream &out, int)
{
    out << "'" << value_p << "'";
}


// clone() duplicates this
Object *Builtin_Object::clone()
{
    return(new Builtin_Object(class_p, value_p));
}


// true() determines if this is true: all but 0 or empty string is true
int Builtin_Object::true()
{
```

```
    return(strcmp(value_p, "") != 0
        && strcmp(value_p, "0") != 0
        && strcmp(value_p, false_value) != 0);
}


// run_message() runs a message against this object
Object *Builtin_Object::run_message(char *message, List<Object> *parms_p,
        Error_Log *log_p)
{
    Builtin_Method *method_p =
                class_p->find_method(message, parms_p->size());

    // if there is a method...
    if(method_p != NULL)
        return((*method_p->implementation())(this, parms_p));
    else
    {
        log_p->log("Unknown method specified for built-in object");
        return(NULL);
    }
}


// value() gets the value of this
char *Builtin_Object::value()
{
    return(value_p);
}


// type() returns the type of this
Builtin_Class *Builtin_Object::type()
{
    return(class_p);
}
```

```
// binclass.cpp

// Contains the code for Builtin_Class, which is a Cyrano
// built-in class.

#include <string.h>
#include <stddef.h>

#include "binclass.h"
#include "list.h"
#include "errorlog.h"
#include "object.h"
#include "binmthd.h"
#include "class.h"


// Builtin_Class() is a constructor
Builtin_Class::Builtin_Class(char *name, Environment *env_p) :
                Class(name, NULL, env_p)
{
}


// run() runs this - you cannot run a Builtin-Class
List<Object> *Builtin_Class::run(Error_Log *log)
{
    log->log("Built-in classes cannot be run");
    return(NULL);
}


// add_method() adds a method to the class
void Builtin_Class::add_method(char *msg, int arity, Symbol *type_p,
        Object *(*imp_p)(Object *, List<Object> *))
{
    Builtin_Method *method_p = new Builtin_Method(msg, arity, type_p,
                                                  imp_p);
    methods.add(method_p);
}


// find_method() finds a method
Builtin_Method *Builtin_Class::find_method(char *name, int arity)
{
    int i;                // loop index

    // check out all methods
    for(i=0; i<methods.size(); ++i)
    {
        if(strcmp(name, methods[i]->name()) == 0
                && methods[i]->arity() == arity)
            return(methods[i]);
    }

    return(NULL);
}
```

```
// binmthd.cpp

// Contains the code for Builtin_Method, which is a method of
// a built-in class.

#include "binmthd.h"
#include "object.h"
#include "list.h"
#include "method.h"

Builtin_Method::Builtin_Method(char *msg, int arity, Symbol *type_p,
        Object *(*in_implementation_p)(Object *, List<Object> *))
        : Method(type_p, msg, arity)
{
    implementation_p = in_implementation_p;
}


Object *(*Builtin_Method::implementation())(Object *, List<Object> *)
{
    return(implementation_p);
}
```

```
// class.cpp

// Contains the code for Class, which is a class representing
// a Cyrano class.

#include <fstream.h>
#include <string.h>
#include <stdio.h>
#include <io.h>

#include "class.h"
#include "list.h"
#include "environ.h"
#include "classpec.h"
#include "object.h"
#include "gwclass.h"
#include "derclass.h"
#include "result.h"
#include "reslist.h"

// Class() is the constructor for this
Class::Class(char *in_name, Class_Spec *in, Environment *env_p)
{
    // store the name
    class_name = new char[strlen(in_name) + 1];
    strcpy(class_name, in_name);

    // create a symbol for it
    Symbol *symbol_p;
    symbol_p = env_p->find_symbol(class_name);

    // if none...
    if(symbol_p == NULL)
    {
        // construct one
        symbol_p = new Symbol(class_name, NULL);
        env_p->add_symbol(symbol_p);
    }

    // store the text of the class to a file
    filename = new char[L_tmpnam + 1];
    tmpnam(filename);

    ofstream output(filename);

    // construct the text file
    if(in != NULL)
    {
        char *text = in->text();                // text of class

        output.write(text, strlen(text));

        delete[] text;
    }
}
```

```
// ~Class() is the destructor for this
Class::~Class()
{
    remove(filename);
    delete[] filename;

    // clear the name
    if(class_name != NULL)
        delete[] class_name;
}


// run() is the default for this function: it runs a class and
//    add results to a result's list
// Return 1 to indicate that stuff was added to result
int Class::run_for_result(Result *result_p, Result_List *,
                  Error_Log *log_p)
{
    // get member objects
    List<Object> *answers_p = run(log_p);

    // store results
    if(!log_p->is_errors() && answers_p != NULL)
    {
        result_p->add(answers_p);
    }

    // delete answer list
    if(answers_p != NULL)
        delete answers_p;

    // note we are complete
    result_p->mark_complete();

    return(1);
}



// text() gets the text of the class
char *Class::text()
{
    ifstream in(filename);                  // the input file
    long size = filelength(in.rdbuf()->fd());
                                            // length of input
    char *contents = new char[size + 1];
                                            // file contents

    in.read(contents, size);
    contents[in.gcount()] = '\0';

    return(contents);
}



// name() gives the name of this
char *Class::name()
```

```
{
    return(class_name);
}


// get_bases() is the default for this: it does nothing
void Class::get_bases(Result_List *, Error_Log *)
{
}


// The following is not a member but is closely associated

// read_class() reads in a class
Class *read_class(Class_Spec *in, Environment *env_p)
{
    // get Class label
    if(in->next_token() != Class_Label)
    {
        in->log_error("Missing CLASS label");
        return(NULL);
    }
    in->skip_token();

    // get name
    if(in->next_token() != Name)
    {
        in->log_error("Missing class name");
        return(NULL);
    }

    char *name = new char[strlen(in->get_value()) + 1];  // name of class

    strcpy(name, in->get_value());
    in->skip_token();

    // get IS
    if(in->next_token() != Is_Label)
    {
        in->log_error("Missing IS in class specification");
        delete[] name;
        return(NULL);
    }
    in->skip_token();

    // get type
    Class *class_p = NULL;
    switch(in->next_token())
    {
        case Derived_Label:
            class_p = new Derived_Class(name, in, env_p);
            break;
        case Gateway_Label:
            if((class_p = read_gateway_class(name, in, env_p)) == NULL)
            {
                delete[] name;
                return(NULL);
            }
```

```
            break;
        default:
            in->log_error("Unknown class type");
            delete[] name;
            return(NULL);
    }

    // done with name now
    delete[] name;

    // get End
    if(in->next_token() != End_Label)
    {
        in->log_error("Missing END of class");
        delete class_p;
        return(NULL);
    }
    in->skip_token();

    // get Class
    if(in->next_token() != Class_Label)
    {
        in->log_error("Missing CLASS at end of class");
        delete class_p;
        return(NULL);
    }
    in->skip_token();

    // get period
    if(in->next_token() != Period)
    {
        in->log_error("Missing period at end of class");
        delete class_p;
        return(NULL);
    }
    in->skip_token();

    return(class_p);
}
```

```
// classpec.cpp

// Contains the code for Class_Spec, which is the lexical
// analyzer for Cyrano.

#include <iostream.h>
#include <fstream.h>
#include <string.h>
#include <io.h>

#include "classpec.h"
#include "errorlog.h"

static const int max_token_len = 256;                    // max length of a
token

// Class_Spec() constructs this
Class_Spec::Class_Spec(char *in_filename)
{
    // store the filename
    filename = new char[strlen(in_filename) + 1];
    strcpy(filename, in_filename);

    // create the input spec
    token_input_p = new ifstream(filename);

    // fill in the state table
    fill_in_state_table();

    // allocate space for token value
    token_value_buffer = new char[max_token_len + 1];

    // get the first token
    skip_token();
}



// ~Class_Spec() destructs this
Class_Spec::~Class_Spec()
{
    delete token_input_p;

    if(token_value_buffer != NULL)
        delete[] token_value_buffer;

    if(filename != NULL)
        delete[] filename;
}


// next_token() gets the type of the next token
token_type Class_Spec::next_token()
{
    return(current_token);
}
```

```
// the following enumeration defines the states to simplify reading
// Capital letters following St_ are characters matched at that state
enum state_types
        {St_base, St_name, St_period, St_semicolon, St_colon,
        St_open_paren, St_close_paren, St_comma, St_string_open,
        St_equals, St_less, St_greater, St_plus, St_minus, St_slash,
        St_asterisk, St_less_equals, St_greater_equals,
        St_less_greater,
        St_string, St_integer,
        St_C, St_CL, St_CLA, St_CLAS, St_CLASS,
        St_I, St_IS,
        St_E, St_EN, St_END,
        St_D, St_DE, St_DER, St_DERI, St_DERIV, St_DERIVE, St_DERIVED,
        St_DA, St_DAT, St_DATA,        St_DATAL, St_DATALO, St_DATALOG,
        St_W, St_WI, St_WIT, St_WITH,
        St_EX, St_EXI, St_EXIS, St_EXIST, St_EXISTS,
        St_F, St_FO, St_FOR, St_FORA, St_FORAL, St_FORALL,
        St_FA, St_FAL, St_FALS, St_FALSE,
        St_A, St_AN, St_AND,
        St_O, St_OR,
        St_G, St_GA, St_GAT, St_GATE, St_GATEW, St_GATEWA, St_GATEWAY,
        St_T, St_TR, St_TRU, St_TRUE,
        St_P, St_PA, St_PAR, St_PARA, St_PARAD, St_PARADO, St_PARADOX,
        St_N, St_NO, St_NOT};

// skip_token() skips over the current token
void Class_Spec::skip_token()
{
    // skip over any blanks
    skip_blanks();

    // initialize token destination stuff
    char *token_p = token_value_buffer;          // ptr at current token
    state_types current_state = St_base;         // current state in FSA
    state_types last_state = St_base;            // last state found

    // initialize token value at top of buffer
    token_value_p = token_value_buffer;

    // do until done
    while(current_state != no_state)
    {
        last_state = current_state;
        if(token_input_p->eof())
            current_state = (state_types)-1;
        else
        {
            token_input_p->get(*token_p);
            current_state =
              (state_types)state_table.get_next_state(last_state,
                                                      *token_p);
        }
        ++token_p;
    }

    if(!token_input_p->eof())
        token_input_p->putback(*(token_p - 1));
```

```
    if((current_token = (token_type)state_table.get_type(last_state))
                == (token_type)no_type)
        current_token = End_of_Input;

    *(token_p - 1) = '\0';

    // if current token is a string, strip quotes
    if(current_token == String)
    {
        token_value_p++;
        *(token_p - 2) = '\0';
    }
}


// get_value() gets the value of the next token
char *Class_Spec::get_value()
{
    return(token_value_p);
}



// the following defines a useful function for state table stuff
static void create_transitions(State_Table &, int, int,
                        unsigned char, unsigned char);
static void create_name_transitions(State_Table &, int, int);

// fill_in_state_table() fills in the state table as needed
void Class_Spec::fill_in_state_table()
{
    // do for all states in great tedium
    create_name_transitions(state_table, St_base, St_name);
    state_table.add_transition(St_base, '.', St_period);
    state_table.add_transition(St_base, ';', St_semicolon);
    state_table.add_transition(St_base, ':', St_colon);
    state_table.add_transition(St_base, '(', St_open_paren);
    state_table.add_transition(St_base, ')', St_close_paren);
    state_table.add_transition(St_base, ',', St_comma);
    state_table.add_transition(St_base, '"', St_string_open);
    state_table.add_transition(St_base, '<', St_less);
    state_table.add_transition(St_base, '>', St_greater);
    state_table.add_transition(St_base, '+', St_plus);
    state_table.add_transition(St_base, '=', St_equals);
    state_table.add_transition(St_base, '-', St_minus);
    state_table.add_transition(St_base, '/', St_slash);
    state_table.add_transition(St_base, '*', St_asterisk);
    create_transitions(state_table, St_base, St_integer, '0', '9');
    create_transitions(state_table, St_integer, St_integer, '0', '9');
    state_table.add_transition(St_base, 'C', St_C);
    state_table.add_transition(St_base, 'I', St_I);
    state_table.add_transition(St_base, 'E', St_E);
    state_table.add_transition(St_base, 'D', St_D);
    state_table.add_transition(St_base, 'W', St_W);
    state_table.add_transition(St_base, 'F', St_F);
    state_table.add_transition(St_base, 'A', St_A);
    state_table.add_transition(St_base, 'O', St_O);
```

```
state_table.add_transition(St_base, 'G', St_G);
state_table.add_transition(St_base, 'P', St_P);
state_table.add_transition(St_base, 'T', St_T);
state_table.add_transition(St_base, 'N', St_N);
state_table.add_transition(St_base, 'c', St_C);
state_table.add_transition(St_base, 'i', St_I);
state_table.add_transition(St_base, 'e', St_E);
state_table.add_transition(St_base, 'd', St_D);
state_table.add_transition(St_base, 'w', St_W);
state_table.add_transition(St_base, 'f', St_F);
state_table.add_transition(St_base, 'a', St_A);
state_table.add_transition(St_base, 'o', St_O);
state_table.add_transition(St_base, 'g', St_G);
state_table.add_transition(St_base, 'p', St_P);
state_table.add_transition(St_base, 't', St_T);
state_table.add_transition(St_base, 'n', St_N);
create_name_transitions(state_table, St_name, St_name);
create_transitions(state_table, St_string_open, St_string_open, 0,
            127);
state_table.add_transition(St_string_open, '"', St_string);
create_name_transitions(state_table, St_C, St_name);
state_table.add_transition(St_C, 'L', St_CL);
state_table.add_transition(St_C, 'l', St_CL);
create_name_transitions(state_table, St_CL, St_name);
state_table.add_transition(St_CL, 'A', St_CLA);
state_table.add_transition(St_CL, 'a', St_CLA);
create_name_transitions(state_table, St_CLA, St_name);
state_table.add_transition(St_CLA, 's', St_CLAS);
state_table.add_transition(St_CLA, 'S', St_CLAS);
create_name_transitions(state_table, St_CLAS, St_name);
state_table.add_transition(St_CLAS, 's', St_CLASS);
state_table.add_transition(St_CLAS, 'S', St_CLASS);
create_name_transitions(state_table, St_CLASS, St_name);
create_name_transitions(state_table, St_I, St_name);
state_table.add_transition(St_I, 's', St_IS);
state_table.add_transition(St_I, 'S', St_IS);
create_name_transitions(state_table, St_IS, St_name);
create_name_transitions(state_table, St_E, St_name);
state_table.add_transition(St_E, 'n', St_EN);
state_table.add_transition(St_E, 'N', St_EN);
state_table.add_transition(St_E, 'x', St_EX);
state_table.add_transition(St_E, 'X', St_EX);
create_name_transitions(state_table, St_EN, St_name);
state_table.add_transition(St_EN, 'd', St_END);
state_table.add_transition(St_EN, 'D', St_END);
create_name_transitions(state_table, St_END, St_name);
create_name_transitions(state_table, St_D, St_name);
state_table.add_transition(St_D, 'e', St_DE);
state_table.add_transition(St_D, 'E', St_DE);
state_table.add_transition(St_D, 'a', St_DA);
state_table.add_transition(St_D, 'A', St_DA);
create_name_transitions(state_table, St_DE, St_name);
state_table.add_transition(St_DE, 'r', St_DER);
state_table.add_transition(St_DE, 'R', St_DER);
create_name_transitions(state_table, St_DER, St_name);
state_table.add_transition(St_DER, 'i', St_DERI);
state_table.add_transition(St_DER, 'I', St_DERI);
create_name_transitions(state_table, St_DERI, St_name);
```

```
state_table.add_transition(St_DERI, 'v', St_DERIV);
state_table.add_transition(St_DERI, 'V', St_DERIV);
create_name_transitions(state_table, St_DERIV, St_name);
state_table.add_transition(St_DERIV, 'e', St_DERIVE);
state_table.add_transition(St_DERIV, 'E', St_DERIVE);
create_name_transitions(state_table, St_DERIVE, St_name);
state_table.add_transition(St_DERIVE, 'd', St_DERIVED);
state_table.add_transition(St_DERIVE, 'D', St_DERIVED);
create_name_transitions(state_table, St_DERIVED, St_name);
create_name_transitions(state_table, St_DA, St_name);
state_table.add_transition(St_DA, 't', St_DAT);
state_table.add_transition(St_DA, 'T', St_DAT);
create_name_transitions(state_table, St_DAT, St_name);
state_table.add_transition(St_DAT, 'a', St_DATA);
state_table.add_transition(St_DAT, 'A', St_DATA);
create_name_transitions(state_table, St_DATA, St_name);
state_table.add_transition(St_DATA, 'l', St_DATAL);
state_table.add_transition(St_DATA, 'L', St_DATAL);
create_name_transitions(state_table, St_DATAL, St_name);
state_table.add_transition(St_DATAL, 'o', St_DATALO);
state_table.add_transition(St_DATAL, 'O', St_DATALO);
create_name_transitions(state_table, St_DATALO, St_name);
state_table.add_transition(St_DATALO, 'g', St_DATALOG);
state_table.add_transition(St_DATALO, 'G', St_DATALOG);
create_name_transitions(state_table, St_DATALOG, St_name);
create_name_transitions(state_table, St_W, St_name);
state_table.add_transition(St_W, 'i', St_WI);
state_table.add_transition(St_W, 'I', St_WI);
create_name_transitions(state_table, St_WI, St_name);
state_table.add_transition(St_WI, 't', St_WIT);
state_table.add_transition(St_WI, 'T', St_WIT);
create_name_transitions(state_table, St_WIT, St_name);
state_table.add_transition(St_WIT, 'h', St_WITH);
state_table.add_transition(St_WIT, 'H', St_WITH);
create_name_transitions(state_table, St_WITH, St_name);
create_name_transitions(state_table, St_EX, St_name);
state_table.add_transition(St_EX, 'i', St_EXI);
state_table.add_transition(St_EX, 'I', St_EXI);
create_name_transitions(state_table, St_EXI, St_name);
state_table.add_transition(St_EXI, 's', St_EXIS);
state_table.add_transition(St_EXI, 'S', St_EXIS);
create_name_transitions(state_table, St_EXIS, St_name);
state_table.add_transition(St_EXIS, 't', St_EXIST);
state_table.add_transition(St_EXIS, 'T', St_EXIST);
create_name_transitions(state_table, St_EXIST, St_name);
state_table.add_transition(St_EXIST, 's', St_EXISTS);
state_table.add_transition(St_EXIST, 'S', St_EXISTS);
create_name_transitions(state_table, St_EXISTS, St_name);
create_name_transitions(state_table, St_F, St_name);
state_table.add_transition(St_F, 'o', St_FO);
state_table.add_transition(St_F, 'O', St_FO);
state_table.add_transition(St_F, 'a', St_FA);
state_table.add_transition(St_F, 'A', St_FA);
create_name_transitions(state_table, St_FO, St_name);
state_table.add_transition(St_FO, 'R', St_FOR);
state_table.add_transition(St_FO, 'r', St_FOR);
create_name_transitions(state_table, St_FOR, St_name);
state_table.add_transition(St_FOR, 'A', St_FORA);
```

```
state_table.add_transition(St_FOR, 'a', St_FORA);
create_name_transitions(state_table, St_FORA, St_name);
state_table.add_transition(St_FORA, 'L', St_FORAL);
state_table.add_transition(St_FORA, 'l', St_FORAL);
create_name_transitions(state_table, St_FORAL, St_name);
state_table.add_transition(St_FORAL, 'L', St_FORALL);
state_table.add_transition(St_FORAL, 'l', St_FORALL);
create_name_transitions(state_table, St_FORALL, St_name);
create_name_transitions(state_table, St_FA, St_name);
state_table.add_transition(St_FA, 'L', St_FAL);
state_table.add_transition(St_FA, 'l', St_FAL);
create_name_transitions(state_table, St_FAL, St_name);
state_table.add_transition(St_FAL, 'S', St_FALS);
state_table.add_transition(St_FAL, 's', St_FALS);
create_name_transitions(state_table, St_FALS, St_name);
state_table.add_transition(St_FALS, 'E', St_FALSE);
state_table.add_transition(St_FALS, 'e', St_FALSE);
create_name_transitions(state_table, St_FALSE, St_name);
create_name_transitions(state_table, St_A, St_name);
state_table.add_transition(St_A, 'n', St_AN);
state_table.add_transition(St_A, 'N', St_AN);
create_name_transitions(state_table, St_AN, St_name);
state_table.add_transition(St_AN, 'd', St_AND);
state_table.add_transition(St_AN, 'D', St_AND);
create_name_transitions(state_table, St_AND, St_name);
create_name_transitions(state_table, St_O, St_name);
state_table.add_transition(St_O, 'r', St_OR);
state_table.add_transition(St_O, 'R', St_OR);
create_name_transitions(state_table, St_OR, St_name);
create_name_transitions(state_table, St_G, St_name);
state_table.add_transition(St_G, 'a', St_GA);
state_table.add_transition(St_G, 'A', St_GA);
create_name_transitions(state_table, St_GA, St_name);
state_table.add_transition(St_GA, 't', St_GAT);
state_table.add_transition(St_GA, 'T', St_GAT);
create_name_transitions(state_table, St_GAT, St_name);
state_table.add_transition(St_GAT, 'e', St_GATE);
state_table.add_transition(St_GAT, 'E', St_GATE);
create_name_transitions(state_table, St_GATE, St_name);
state_table.add_transition(St_GATE, 'w', St_GATEW);
state_table.add_transition(St_GATE, 'W', St_GATEW);
create_name_transitions(state_table, St_GATEW, St_name);
state_table.add_transition(St_GATEW, 'a', St_GATEWA);
state_table.add_transition(St_GATEW, 'A', St_GATEWA);
create_name_transitions(state_table, St_GATEWA, St_name);
state_table.add_transition(St_GATEWA, 'y', St_GATEWAY);
state_table.add_transition(St_GATEWA, 'Y', St_GATEWAY);
create_name_transitions(state_table, St_GATEWAY, St_name);
create_name_transitions(state_table, St_P, St_name);
state_table.add_transition(St_P, 'a', St_PA);
state_table.add_transition(St_P, 'A', St_PA);
create_name_transitions(state_table, St_PA, St_name);
state_table.add_transition(St_PA, 'r', St_PAR);
state_table.add_transition(St_PA, 'R', St_PAR);
create_name_transitions(state_table, St_PAR, St_name);
state_table.add_transition(St_PAR, 'a', St_PARA);
state_table.add_transition(St_PAR, 'A', St_PARA);
create_name_transitions(state_table, St_PARA, St_name);
```

```
state_table.add_transition(St_PARA, 'd', St_PARAD);
state_table.add_transition(St_PARA, 'D', St_PARAD);
create_name_transitions(state_table, St_PARAD, St_name);
state_table.add_transition(St_PARAD, 'o', St_PARADO);
state_table.add_transition(St_PARAD, 'O', St_PARADO);
create_name_transitions(state_table, St_PARADO, St_name);
state_table.add_transition(St_PARADO, 'x', St_PARADOX);
state_table.add_transition(St_PARADO, 'X', St_PARADOX);
create_name_transitions(state_table, St_PARADOX, St_name);
create_name_transitions(state_table, St_T, St_name);
state_table.add_transition(St_T, 'r', St_TR);
state_table.add_transition(St_T, 'R', St_TR);
create_name_transitions(state_table, St_TR, St_name);
state_table.add_transition(St_TR, 'u', St_TRU);
state_table.add_transition(St_TR, 'U', St_TRU);
create_name_transitions(state_table, St_TRU, St_name);
state_table.add_transition(St_TRU, 'e', St_TRUE);
state_table.add_transition(St_TRU, 'E', St_TRUE);
create_name_transitions(state_table, St_TRUE, St_name);
create_name_transitions(state_table, St_N, St_name);
state_table.add_transition(St_N, 'o', St_NO);
state_table.add_transition(St_N, 'O', St_NO);
create_name_transitions(state_table, St_NO, St_name);
state_table.add_transition(St_NO, 't', St_NOT);
state_table.add_transition(St_NO, 'T', St_NOT);
create_name_transitions(state_table, St_NOT, St_name);
state_table.add_transition(St_less, '=', St_less_equals);
state_table.add_transition(St_less, '>', St_less_greater);
state_table.add_transition(St_greater, '=', St_greater_equals);

// create end states
state_table.set_type(St_name, Name);
state_table.set_type(St_period, Period);
state_table.set_type(St_semicolon, Semicolon);
state_table.set_type(St_colon, Colon);
state_table.set_type(St_open_paren, Open_Paren);
state_table.set_type(St_close_paren, Close_Paren);
state_table.set_type(St_comma, Comma);
state_table.set_type(St_string, String);
state_table.set_type(St_integer, Integer);
state_table.set_type(St_plus, Plus);
state_table.set_type(St_equals, Equals);
state_table.set_type(St_minus, Minus);
state_table.set_type(St_slash, Slash);
state_table.set_type(St_asterisk, Asterisk);
state_table.set_type(St_less, Less_Than);
state_table.set_type(St_greater, Greater_Than);
state_table.set_type(St_less_greater, Not_Equals);
state_table.set_type(St_less_equals, Less_or_Equals);
state_table.set_type(St_greater_equals, Greater_or_Equals);
state_table.set_type(St_C, Name);
state_table.set_type(St_CL, Name);
state_table.set_type(St_CLA, Name);
state_table.set_type(St_CLAS, Name);
state_table.set_type(St_CLASS, Class_Label);
state_table.set_type(St_I, Name);
state_table.set_type(St_IS, Is_Label);
state_table.set_type(St_E, Name);
```

```
state_table.set_type(St_EN, Name);
state_table.set_type(St_END, End_Label);
state_table.set_type(St_D, Name);
state_table.set_type(St_DE, Name);
state_table.set_type(St_DER, Name);
state_table.set_type(St_DERI, Name);
state_table.set_type(St_DERIV, Name);
state_table.set_type(St_DERIVE, Name);
state_table.set_type(St_DERIVED, Derived_Label);
state_table.set_type(St_DA, Name);
state_table.set_type(St_DAT, Name);
state_table.set_type(St_DATA, Name);
state_table.set_type(St_W, Name);
state_table.set_type(St_WI, Name);
state_table.set_type(St_WIT, Name);
state_table.set_type(St_WITH, With_Label);
state_table.set_type(St_EX, Name);
state_table.set_type(St_EXI, Name);
state_table.set_type(St_EXIS, Name);
state_table.set_type(St_EXIST, Name);
state_table.set_type(St_EXISTS, Exists_Label);
state_table.set_type(St_F, Name);
state_table.set_type(St_FO, Name);
state_table.set_type(St_FOR, Name);
state_table.set_type(St_FORA, Name);
state_table.set_type(St_FORAL, Name);
state_table.set_type(St_FORALL, Forall_Label);
state_table.set_type(St_FA, Name);
state_table.set_type(St_FAL, Name);
state_table.set_type(St_FALS, Name);
state_table.set_type(St_FALSE, False_Label);
state_table.set_type(St_A, Name);
state_table.set_type(St_AN, Name);
state_table.set_type(St_AND, And_Label);
state_table.set_type(St_O, Name);
state_table.set_type(St_OR, Or_Label);
state_table.set_type(St_G, Name);
state_table.set_type(St_GA, Name);
state_table.set_type(St_GAT, Name);
state_table.set_type(St_GATE, Name);
state_table.set_type(St_GATEW, Name);
state_table.set_type(St_GATEWA, Name);
state_table.set_type(St_GATEWAY, Gateway_Label);
state_table.set_type(St_P, Name);
state_table.set_type(St_PA, Name);
state_table.set_type(St_PAR, Name);
state_table.set_type(St_PARA, Name);
state_table.set_type(St_PARAD, Name);
state_table.set_type(St_PARADO, Name);
state_table.set_type(St_PARADOX, Paradox_Label);
state_table.set_type(St_DATAL, Name);
state_table.set_type(St_DATALO, Name);
state_table.set_type(St_DATALOG, Datalog_Label);
state_table.set_type(St_T, Name);
state_table.set_type(St_TR, Name);
state_table.set_type(St_TRU, Name);
state_table.set_type(St_TRUE, True_Label);
state_table.set_type(St_N, Name);
```

```
    state_table.set_type(St_NO, Name);
    state_table.set_type(St_NOT, Not_Label);
}


// skip_blanks() skips over blanks
void Class_Spec::skip_blanks()
{
    char ch = ' ';                  // character read in
    int in_comment = 0;             // are we in a comment?

    while(!token_input_p->eof() &&
        (ch == ' ' || ch == '\t' || ch == '\n' || ch == '{'
                || in_comment))
    {
        if(ch == '{')
            in_comment++;
        if(ch == '}')
            in_comment--;
        token_input_p->get(ch);
    }

    if(!token_input_p->eof())
        token_input_p->putback(ch);
}



// create_name_transitions() creates transition to name state
static void create_name_transitions(State_Table &table, int from_state,
        int to_state)
{
    create_transitions(table, from_state, to_state, 'A', 'Z');
    create_transitions(table, from_state, to_state, 'a', 'z');
    create_transitions(table, from_state, to_state, '0', '9');
    table.add_transition(from_state, '_', to_state);
}


// create_transitions() fills in state stuff for range of character
static void create_transitions(State_Table &table,
                               int from_state,
                               int to_state,
                               unsigned char start_ch,
                               unsigned char end_ch)
{
    unsigned char ch;                       // alpha characters

    for(ch = start_ch; ch <= end_ch; ++ch)
        table.add_transition(from_state, ch, to_state);
}



// log_error() records an error
void Class_Spec::log_error(char *error)
{
    error_log.log(error);
```

```
}


// is_errors() returns the state of the error flag
int Class_Spec::is_errors()
{
    return(error_log.is_errors());
}


// get_errors() gets the Error_Log
Error_Log *Class_Spec::get_errors()
{
    return(&error_log);
}


// text() gets current text of everything
char *Class_Spec::text()
{
    ifstream in(filename);        // to hold input file
    long size = filelength(in.rdbuf()->fd());
                                  // length of input
    char *contents = new char[size + 1];
                                  // file contents

    in.read(contents, size);
    contents[in.gcount()] = '\0';

    return(contents);
}
```

```
// cllstwin.cpp

// Contains the class Class_List_Window, which is a window
// containing a list of classes.

#include <owl\dialog.h>
#include <owl\listbox.h>
#include <iostream.h>
#include <strstream.h>
#include <stdio.h>
#include <owl\opensave.h>
#include <io.h>

#include "cllstwin.h"
#include "list.h"
#include "object.h"
#include "environ.h"
#include "symbol.h"
#include "roxwin.h"
#include "filedisp.h"
#include "editwin.h"
#include "classpec.h"
#include "baseimps.h"

static char *const data_dir = "d:\\thesis\\roxanne\\data";
                                                    // data directory
static char *const temp_file = "d:\\thesis\\roxanne\\DATA\\TEMP.DAT";
                                                    // temporary file
static char *const classfile_extension = "cls";   // class file extension
static char *const classfile_filter =
        "Classes (*.cls)|*.cls|All files (*.*)|*.*|";
                                                    // class files stuff

// Class_List_Window() is the constructor for this
Class_List_Window::Class_List_Window(TWindow *parent, TResId res_id)
        : TDialog(parent, res_id), TWindow(parent)
{
     // create environment
     ostrstream errors;                 // to hold errors
     environment_p = new Environment(errors);

     // if there were errors...
     if(errors.pcount() > 0)
     {
         errors << '\0';
         char *string = errors.str();         // error string
         File_Display display(this, "Errors", string);
         display.Execute();

         delete[] string;
     }

     // construct built-in classes
     setup_base_classes(environment_p);

     class_list_p = new TListBox(this, ID_CLASS_LIST);
}
```

```
// the following is the event table for this
DEFINE_RESPONSE_TABLE1(Class_List_Window, TDialog)
    EV_CHILD_NOTIFY(ID_NEW, BN_CLICKED, handle_new),
    EV_CHILD_NOTIFY(ID_DELETE, BN_CLICKED, handle_delete),
    EV_CHILD_NOTIFY(ID_RUN, BN_CLICKED, handle_run),
    EV_CHILD_NOTIFY(ID_EDIT, BN_CLICKED, handle_edit),
    EV_CHILD_NOTIFY(ID_SAVE, BN_CLICKED, handle_save),
    EV_CHILD_NOTIFY(ID_LOAD, BN_CLICKED, handle_load),
    EV_CHILD_NOTIFY(ID_QUIT, BN_CLICKED, CmOk),
END_RESPONSE_TABLE;

// ~Class_List_Window() destructs this
Class_List_Window::~Class_List_Window()
{
    delete environment_p;
}


// handle_new() handles the NEW button
void Class_List_Window::handle_new()
{
    edit_class();
}



// handle_delete() handles the DELETE button
void Class_List_Window::handle_delete()
{
    char class_name[200];                // the class name

    // get the name of the currently selected class
    if(class_list_p->GetSelString(class_name, sizeof(class_name) - 1)
                <= 0)
    {
        MessageBox("Invalid selecton");
        return;
    }

    // get user confirmation
    if(MessageBox("Do you want to delete?", "", MB_YESNO) == IDNO)
        return;

    // find the symbol
    Symbol *symbol_p = environment_p->find_symbol(class_name);
                                // the symbol
    if(symbol_p == NULL)
    {
        MessageBox("Unknown class");
        return;
    }

    // clear the symbol
    symbol_p->clear_value();

    // clear the list entry
    class_list_p->DeleteString(class_list_p->GetSelIndex());
```

```
}


// handle_run() handles the RUN button
void Class_List_Window::handle_run()
{
    char class_name[200];                // the class name

    // get the name of the currently selected class
    if(class_list_p->GetSelString(class_name, sizeof(class_name) - 1)
                <= 0)
    {
        MessageBox("Invalid selecton");
        return;
    }

    // find the symbol
    Symbol *symbol_p = environment_p->find_symbol(class_name);
    if(symbol_p == NULL || symbol_p->value() == NULL)
    {
        MessageBox("Inactive class");
        return;
    }

    Error_Log log;                // an error log

    // run the class
    List<Object> *object_list_p = symbol_p->value()->run(&log);

    // if there are errors...
    if(log.is_errors() || object_list_p == NULL)
    {
        ostrstream errors;                // to hold errors

        log.display_errors(errors);

        errors << '\0';
        char *string = errors.str();        // error string

        File_Display display(this, "Errors", string);
        display.Execute();

        delete[] string;

        if(object_list_p != NULL)
            delete object_list_p;
        return;
    }

    ostrstream out;                                // to hold error output

    int i;                                         // loop index

    // do for all objects...
    for(i=0; i<object_list_p->size(); ++i)
        out << *((*object_list_p)[i]);
```

```
    out << '\0';

    // display the results
    char *string = out.str();

    File_Display file_display(this, "Results", string);
    file_display.Execute();

    delete[] string;
    delete object_list_p;
}



// handle_edit() handles the EDIT button
void Class_List_Window::handle_edit()
{
    char class_name[200];                // the class name

    // get the name of the currently selected class
    if(class_list_p->GetSelString(class_name, sizeof(class_name) - 1)
            <= 0)
    {
        MessageBox("Invalid selecton");
        return;
    }

    // find its symbol
    Symbol *symbol_p = environment_p->find_symbol(class_name);
    if(symbol_p == NULL || symbol_p->value() == NULL)
    {
        MessageBox("Unknown class");
        return;
    }

    // get the text and edit it
    char *text = symbol_p->value()->text();
    edit_class(text);
    delete[] text;
}


// handle_save() handles the SAVE button
void Class_List_Window::handle_save()
{
    char class_name[200];                // the class name

    // get the name of the currently selected class
    if(class_list_p->GetSelString(class_name, sizeof(class_name) - 1)
            <= 0)
    {
        MessageBox("Invalid selecton");
        return;
    }

    // find its symbol
    Symbol *symbol_p = environment_p->find_symbol(class_name);
```

```
    if(symbol_p == NULL || symbol_p->value() == NULL)
    {
        MessageBox("Unknown class");
        return;
    }

    char classfilename[300];                // to hold class filename
    sprintf(classfilename, "%s\\%s.%s", data_dir, class_name,
                classfile_extension);

    ofstream out(classfilename);        // the output file

    out << symbol_p->value()->text();
}



// handle_load() handles the LOAD button
void Class_List_Window::handle_load()
{
    TOpenSaveDialog::TData
        data(OFN_HIDEREADONLY | OFN_PATHMUSTEXIST,
                classfile_filter, classfile_filter,
                data_dir,
                classfile_extension);            // filename data

    // get filename of database
    if(TFileOpenDialog(this, data).Execute() == IDCANCEL)
        return;

    // activate database
    ifstream in(data.FileName);                 // input file
    long size = filelength(in.rdbuf()->fd());  // length of file
    char *buffer = new char[size + 1];         // create buffer to hold it

    // read the buffer
    in.read(buffer, size);
    buffer[in.gcount()] = '\0';

    edit_class(buffer);

    delete[] buffer;
}



// CanClose() determines if can close ROXANNE
BOOL Class_List_Window::CanClose()
{
    return(MessageBox("You are about to exit Roxanne",
                        "Roxanne", MB_OKCANCEL) == IDOK);
}


// edit_class() edits a class
void Class_List_Window::edit_class(char *buffer)
{
```

```
{
    ofstream out(temp_file);                // output to the temp file

    // write out the buffer if there is one
    if(buffer != NULL)
        out << buffer;
}

// do forever...
while(1)
{
    // allow edit of the file
    Edit_Window edit_window(this, temp_file);

    // do the editting
    edit_window.Execute();

    // allow the user to continue or not
    if(MessageBox("Do you wish to continue?", "", MB_YESNO) == IDNO)
        return;

    // create a class spec for this
    Class_Spec spec(temp_file);

    // create a class for this
    Class *class_p = read_class(&spec, environment_p);

    // if failed...
    if(class_p == NULL || spec.is_errors())
    {
        // clean up bad class
        if(class_p != NULL)
            delete class_p;

        // ask user if he wants errors...
        if(MessageBox("Errors encountered: do you wish to see them?",
                    "", MB_YESNO) == IDYES)
        {
            ostrstream errors;                  // to hold the errors

            spec.get_errors()->display_errors(errors);
            errors << '\0';

            char *error_string = errors.str();
                                        // string version of errors

            File_Display file_display(this, "Results", error_string);
            file_display.Execute();

            delete[] error_string;
        }

        // ask user if he wants to continue...
        if(MessageBox("Do you wish to re-edit class?", "", MB_YESNO)
                    == IDNO)
            return;
    }
    else
```

```
        {
            // find the symbol for the chosen class
            Symbol *symbol_p =
                   environment_p->find_symbol(class_p->name());

            // if it is not already active...
            if(symbol_p == NULL || symbol_p->value() == NULL)
            {
                // construct a new symbol
                if(symbol_p == NULL)
                {
                    symbol_p = new Symbol(class_p->name(), class_p);
                    environment_p->add_symbol(symbol_p);
                }
                else
                    symbol_p->set_value(class_p);

                // add it to the list
                class_list_p->AddString(class_p->name());
            }
            else
            {
                // ask user if he wants to overwrite...
                if(MessageBox("Class exists: do you wish to overwrite?",
                              "", MB_YESNO) == IDYES)
                {
                    symbol_p->set_value(class_p);
                }
                else
                {
                    delete class_p;
                }
            }

            return;
        }
    }
}
```

```
// constval.cpp

// Contains the class Constant_Value, which is a value that is
// a constant.

#include "constval.h"
#include "value.h"
#include "classpec.h"
#include "environ.h"
#include "object.h"
#include "list.h"
#include "variable.h"
#include "errorlog.h"
#include "bin_obj.h"
#include "baseimps.h"

// Constant_Value() constructs a constant value
Constant_Value::Constant_Value(Class_Spec *in_p, Environment *env_p)
{
    Symbol *symbol_p;              // the symbol of this
    char *in_value;                // the value of this

    // initialize a value
    value_p = NULL;

    // do based on type of symbol
    switch(in_p->next_token())
    {
        case String:
            symbol_p = env_p->find_symbol(string_class_name);
            in_value = in_p->get_value();
            break;

        case Integer:
            symbol_p = env_p->find_symbol(integer_class_name);
            in_value = in_p->get_value();
            break;

        case True_Label:
            symbol_p = env_p->find_symbol(boolean_class_name);
            in_value = true_value;
            break;

        case False_Label:
            symbol_p = env_p->find_symbol(boolean_class_name);
            in_value = false_value;
            break;

        default:
            in_p->log_error("Invalid constant value");
            break;
    }

    // if no errors yet...
    if(!in_p->is_errors())
    {
        // check symbol
        if(symbol_p == NULL)
```

```
            in_p->log_error("Symbol type not defined");
        else
            value_p = new Builtin_Object(
                        (Builtin_Class *)symbol_p->value(),
                                         in_value);

        // skip value
        in_p->skip_token();
    }
}


// ~Constant_Value() is a destructor
Constant_Value::~Constant_Value()
{
    // delete the value
    if(value_p != NULL)
        delete value_p;
}



// generate_value() makes the value of this
Object *Constant_Value::generate_value(List<Variable> *, Error_Log *)
{
    return(value_p->clone());
}
```

```
// cplxval.cpp

// Contains the code for Complex_Value, which is a value that
// consists of a base object and a message to send to it.

#include "cplxval.h"
#include "value.h"
#include "list.h"
#include "classpec.h"
#include "environ.h"
#include "variable.h"
#include "errorlog.h"
#include "symbol.h"
#include "baseimps.h"


// the following are definitions for internal static functions
static char *get_operator_message(token_type token);

// Complex_Value() is the constructor for this
Complex_Value::Complex_Value(Value *in_base_p, Class_Spec *in_p,
                Environment *env_p)
{
    message = NULL;
    base_p = NULL;

    // special stuff for NOT
    if(in_p->next_token() == Not_Label)
    {
        message = new char[strlen(not_message) + 1];
        strcpy(message, not_message);
        in_p->skip_token();
        base_p = read_value(in_p, env_p);
    }
    else
    {
        base_p = in_base_p;

        // if we have an operator...
        if(in_p->next_token() == Period)
        {
            in_p->skip_token();
            if(in_p->next_token() != Name)
            {
                in_p->log_error("Missing name in complex value");
                return;
            }
            message = new char[strlen(in_p->get_value()) + 1];
            strcpy(message, in_p->get_value());
            in_p->skip_token();

            // get parameters if they are there
            if(in_p->next_token() == Open_Paren)
            {
                in_p->skip_token();
                while(in_p->next_token() != Close_Paren)
                {
                    // get an operand
```

```
                    Value *val_p = read_value(in_p, env_p);
                    if(val_p == NULL)
                    {
                        in_p->log_error("Invalid operand");
                        return;
                    }
                    parameters.add(val_p);
                    if(in_p->next_token() != Close_Paren)
                    {
                        // should be comma - test and skip
                        if(in_p->next_token() != Comma)
                        {
                            in_p->log_error("Invalid operand list");
                            return;
                        }
                        in_p->skip_token();
                    }
                }
            }
        }
        else
        {
            if((message = get_operator_message(in_p->next_token()))
                    == NULL)
            {
                in_p->log_error("Unknown operator");
                return;
            }
            in_p->skip_token();

            // get parameter of it
            Value *val_p = read_value(in_p, env_p);
            if(val_p == NULL)
            {
                in_p->log_error("Invalid operand");
                return;
            }
            parameters.add(val_p);
        }
    }
}


// ~Complex_Value() is a destructor
Complex_Value::~Complex_Value()
{
    if(base_p != NULL)
        delete base_p;
    if(message != NULL)
        delete[] message;
}



// generate_value() generates the value for this
Object *Complex_Value::generate_value(List<Variable> *vars_p,
                        Error_Log *log_p)
{
```

```
    // get the base object
    Object *base_obj_p = base_p->generate_value(vars_p, log_p);

    if(base_obj_p == NULL)
        return(NULL);

    int i;                                  // loop index

    // construct a parameter list for this
    List<Object> real_parameters;

    // do for all parameters...
    for(i=0; i<parameters.size(); ++i)
    {
        real_parameters.add(parameters[i]->generate_value(vars_p, log_p));
    }

    Object *result_p = NULL;                // result of this

    // if no errors yet...
    if(!log_p->is_errors())
    {
        result_p = base_obj_p->run_message(message, &real_parameters,
                        log_p);
    }

    delete base_obj_p;

    return(result_p);
}


// get_quantifiers() gets list of quantifiers for this
void Complex_Value::get_quantifiers(List<Symbol> *symbols_p)
{
    base_p->get_quantifiers(symbols_p);

    int i;                          // loop index

    for(i=0; i<parameters.size(); ++i)
    {
        parameters[i]->get_quantifiers(symbols_p);
    }
}



// get_operator_message() gets the message for an operator
static char *get_operator_message(token_type token)
{
    char *message = new char[max_operator_len + 1];
    switch(token)
    {
        case Plus:
            strcpy(message, plus_message);
            break;
```

```
        case Minus:
            strcpy(message, minus_message);
            break;
        case Slash:
            strcpy(message, slash_message);
            break;
        case Asterisk:
            strcpy(message, asterisk_message);
            break;
        case Less_Than:
            strcpy(message, less_than_message);
            break;
        case Greater_Than:
            strcpy(message, greater_than_message);
            break;
        case Equals:
            strcpy(message, equals_message);
            break;
        case Not_Equals:
            strcpy(message, not_equals_message);
            break;
        case Less_or_Equals:
            strcpy(message, less_or_equals_message);
            break;
        case Greater_or_Equals:
            strcpy(message, greater_or_equals_message);
            break;
        case And_Label:
            strcpy(message, and_message);
            break;
        case Or_Label:
            strcpy(message, or_message);
            break;
        default:
            delete message;
            message = NULL;
            break;
    }

    return(message);
}
```

```
// derclass.cpp

// Contains the code for Derived_Class, which is a derived
// class.

#include <stddef.h>

#include "derclass.h"
#include "classpec.h"
#include "environ.h"
#include "class.h"
#include "list.h"
#include "errorlog.h"
#include "object.h"
#include "result.h"
#include "derivati.h"
#include "reslist.h"


// Derived_Class() is constructor for this
Derived_Class::Derived_Class(char *name, Class_Spec *in_p,
                  Environment *env_p)
        : Class(name, in_p, env_p)
{
    // get "derived"
    if(in_p->next_token() != Derived_Label)
    {
        in_p->log_error("Syntax error: missing DERIVED");
        return;
    }
    in_p->skip_token();

    // get superclasses if they are there
    if(in_p->next_token() == Is_Label)
    {
        do
        {
            in_p->skip_token();
            if(in_p->next_token() != Name)
            {
                in_p->log_error("Missing superclass name");
                return;
            }
            Symbol *sym_p = env_p->find_symbol(in_p->get_value());
            if(sym_p == NULL)
            {
                in_p->log_error("Unknown superclass");
                return;
            }
            superclasses.add(sym_p);
            in_p->skip_token();
        } while(in_p->next_token() == Comma);
    }

    // get "with"
    if(in_p->next_token() != With_Label)
    {
```

```
        in_p->log_error("Missing WITH");
        return;
    }
    in_p->skip_token();

    // do for all derivations
    while(in_p->next_token() != End_Label)
    {
        Derivation *derive_p;        // the derivation

        // get the derivation
        derive_p = new Derivation(in_p, env_p, &superclasses);

        // check it out
        if(in_p->is_errors())
        {
            delete derive_p;
            return;
        }

        // get semi-colon terminating it
        if(in_p->next_token() != Semicolon)
        {
            in_p->log_error("Missing semi-colon on end of derivation");
            return;
        }
        in_p->skip_token();

        derivations.add(derive_p);
    }
}


// ~Derived_Class() destructs a Derived_Class
Derived_Class::~Derived_Class()
{
    superclasses.empty();
}


// run() gets all members of this class
List<Object> *Derived_Class::run(Error_Log *log_p)
{
    Result_List results;              // results for all objects in this
    int i;                               // loop index

    // find classes referenced in this
    results.add_class(this, log_p);

    // do for all derivations...
    for(i=0; i<derivations.size(); ++i)
    {
        derivations[i]->get_bases(&results, log_p);
    }

    // if errors...
    if(log_p->is_errors())
        return(NULL);
```

```
    // run the result
    results.run(log_p);

    List<Object> *members_p = NULL;            // members of this class

    // if no errors...
    if(!log_p->is_errors())
    {
        members_p = results.extract(this);
    }

    return(members_p);
}


// run_for_result() runs this to add stuff to results
// Return whether new objects were found.
int Derived_Class::run_for_result(Result *result_p, Result_List *list_p,
                        Error_Log *log_p)
{
    int i;                          // loop index
    int status = 0;                  // processing status

    // do for all derivations...
    for(i=0; i<derivations.size(); ++i)
    {
        // run the derivation
        if(derivations[i]->run(result_p, list_p, log_p))
            status = 1;
    }

    // see if class is complete
    int complete = 1;
    for(i=0; i<derivations.size(); ++i)
    {
        if(!derivations[i]->complete(list_p))
        {
            complete = 0;
            break;
        }
    }

    // if is complete...
    if(complete)
        result_p->mark_complete();

    return(status);
}


// get_bases() gets all base classes referenced by this class and adds
//     to a result list
void Derived_Class::get_bases(Result_List *list_p, Error_Log *log_p)
{
    int i;                          // loop index
```

```
    for(i=0; i<derivations.size(); ++i)
        derivations[i]->get_bases(list_p, log_p);
}
```

```
// derivati.cpp

// Contains the code for Derivation, which is a derivation of
// a derived class.

#include <stddef.h>

#include "derivati.h"
#include "dermthd.h"
#include "list.h"
#include "variable.h"
#include "value.h"
#include "symbol.h"
#include "result.h"
#include "reslist.h"
#include "errorlog.h"
#include "classpec.h"
#include "environ.h"
#include "derobj.h"

// Derivation() is the constructor for this
Derivation::Derivation(Class_Spec *in_p, Environment *env_p,
        List<Symbol> *superclasses_p)
{
    // initialize stuff
    guard_p = NULL;

    // do while name indicates there is a variable to follow
    while(in_p->next_token() == Name)
    {
        Variable *var_p;                // a base variable

        // get the variable
        if((var_p = new Variable(in_p, env_p)) == NULL)
        {
            in_p->log_error("Cannot get base variable");
            return;
        }
        base_variables.add(var_p);

        // skip comma if there is one
        if(in_p->next_token() != Colon)
        {
            if(in_p->next_token() != Comma)
            {
                in_p->log_error("Invalid base variable list");
                return;
            }
            in_p->skip_token();
        }
    }

    // make sure this is a colon
    if(in_p->next_token() != Colon)
    {
        in_p->log_error("Invalid base variable list");
        return;
    }
```

```
    in_p->skip_token();

    // get the guard
    guard_p = read_value(in_p, env_p);

    // get WITH
    if(in_p->next_token() != With_Label)
    {
        in_p->log_error("Missing methods of derivation");
        return;
    }
    in_p->skip_token();

    // do as long as there is a method
    while(in_p->next_token() != End_Label)
    {
        // get derived method
        Derived_Method *method_p = new Derived_Method(in_p, env_p);

        // make sure it is all right
        if(in_p->is_errors())
        {
            return;
        }

        methods.add(method_p);

        // get semi-colon for this
        if(in_p->next_token() != Semicolon)
        {
            in_p->log_error("Missing semicolon on method");
            return;
        }
        in_p->skip_token();
    }

    // make sure there is an END here
    if(in_p->next_token() != End_Label)
    {
        in_p->log_error("Missing END on derivation");
        return;
    }
    in_p->skip_token();

    // get the quantifiers of this
    guard_p->get_quantifiers(&quantifiers);

    // add a superobject for each superclass
    int i;                  // loop index

    for(i=0; i<superclasses_p->size(); ++i)
    {
        superobjects.add(new Variable("This", NULL,
                         (*superclasses_p)[i]));
    }
}
```

```
// ~Derivation() is the destructor
Derivation::~Derivation()
{
    if(guard_p != NULL)
        delete guard_p;
}


// run runs the derivation and returns 1 if anything found
int  Derivation::run(Result  *result_p,  Result_List  *list_p,  Error_Log
*log_p)
{
    int status = 0;                // processing status

    // if this is ready to run...
    if(quantifiers_complete(list_p))
        if(get_superobject_and_test(0, 0, result_p, list_p, log_p))
            status = 1;

    return(status);
}


// find_method() finds a method of this
Derived_Method *Derivation::find_method(char *name, int arity)
{
    int i;                         // loop index

    // do for all methods
    for(i=0; i<methods.size(); ++i)
    {
        if(strcmp(methods[i]->name(), name) == 0
                && methods[i]->arity() == arity)
            return(methods[i]);
    }

    return(NULL);
}


// complete() tells if the derivation is complete
int Derivation::complete(Result_List *list_p)
{
    int complete = 1;                 // completion flag
    int i;                         // loop index

    // do for all base variables
    for(i=0; i<base_variables.size(); ++i)
    {
        // get class of variable
        Class *class_p = base_variables[i]->type()->value();

        // if incomplete or cannot be gotten...
        if(class_p == NULL
                || list_p->find_result(class_p) == NULL
                || !list_p->find_result(class_p)->complete())
```

```
        {
            complete = 0;
            break;
        }
    }

    if(complete)
    {
        // do for all superobjects
        for(i=0; i<superobjects.size(); ++i)
        {
            // get class of variable
            Class *class_p = superobjects[i]->type()->value();

            // if incomplete or cannot be gotten...
            if(class_p == NULL
                    || list_p->find_result(class_p) == NULL
                    || !list_p->find_result(class_p)->complete())
            {
                complete = 0;
                break;
            }
        }
    }

    // check quantifiers
    if(complete)
        complete = quantifiers_complete(list_p);

    return(complete);
}

// get_bases() gets base variables of this derivation
void Derivation::get_bases(Result_List *list_p, Error_Log *log_p)
{
    int i;                          // loop index

    // do for all base classes...
    for(i=0; i<superobjects.size(); ++i)
    {
        // make sure the variable's type exists
        if(superobjects[i]->type()->value() == NULL)
        {
            log_p->log("Invalid variable class");
            return;
        }
        list_p->add_class(superobjects[i]->type()->value(),
                          log_p);
    }

    // do for all base variables...
    for(i=0; i<base_variables.size(); ++i)
    {
        // make sure the variable's type exists
        if(base_variables[i]->type()->value() == NULL)
        {
            log_p->log("Invalid variable class");
            return;
```

```
        }
        list_p->add_class(base_variables[i]->type()->value(),
                          log_p);
    }

    // add classes of all derivations to this
    for(i=0; i<quantifiers.size(); ++i)
    {
        if(quantifiers[i]->value() == NULL)
        {
            log_p->log("Invalid quantifier class");
            return;
        }
        list_p->add_class(quantifiers[i]->value(),
                          log_p);
    }
}


// get_superobject_and_test() recursively gets all potential superobject
// combinations and tests them
int Derivation::get_superobject_and_test(int superobject_no, int is_new,
        Result *result_p, Result_List *list_p, Error_Log *log_p)
{
    int status = 0;                 // processing status

    // if we must find superobjects...
    if(superobject_no < superobjects.size())
    {
        // if there are any superobjects...
        if(superobject_no > 0)
        {
            Result *base_result_p;                  // result for this

            // Cycle through all possible values for current superobject
            // and recursively call this function for the next one.

            // find the result
            base_result_p = list_p->find_result(
                        superobjects[superobject_no]->type()->value());
            if(base_result_p == NULL)
            {
                log_p->log("Missing result for superobject");
            }

            // get values of this
            List<Object> *base_objects_p = base_result_p->get_objects();
            int i;                                  // loop index

            // do for all objects...
            for(i=0; i<base_objects_p->size(); ++i)
            {
                superobjects[superobject_no]->set_value(
                                        (*base_objects_p)[i]);
                if(get_superobject_and_test(superobject_no + 1,
                        base_result_p->is_new(i) | is_new, result_p,
                      list_p, log_p))
```

```
                    status = 1;
                superobjects[superobject_no]->remove_value();
            }
        }
    }
    else
    {
        // if there are superobjects...
        if(superobjects.size() != 0)
        {
            Derived_Object *superobject_p;        // superobject of this

            // construct a super object of this
            superobject_p = new Derived_Object;

            int i;                          // loop index

            for(i=0; i<superobjects.size(); ++i)
            {
                superobject_p->add_superobject(
                        superobjects[i]->value()->clone());
            }

            base_variables.add(new Variable(superobject_name,
                        superobject_p));
        }

        status = get_base_and_test(0, is_new, result_p, list_p, log_p);

        // if there was a superobject added...
        if(superobjects.size() != 0)
        {
            // delete it
            base_variables.clear(base_variables.size() - 1);
            base_variables.remove(base_variables.size() - 1);
        }
    }

    return(status);
}


// quantifiers_complete() tests if all quantifiers are complete
int Derivation::quantifiers_complete(Result_List *list_p)
{
    int complete = 1;                          // completion flag
    int i;                                  // loop index

    for(i=0; i<quantifiers.size(); ++i)
    {
        Result *result_p;                // result of the quantifier

        result_p = list_p->find_result(quantifiers[i]->value());

        // if not complete...
        if(result_p == NULL || !result_p->complete())
        {
```

```
            complete = 0;
            break;
        }
    }

    return(complete);
}




// get_base_and_test() gets the next base variable and, when all are
//      gotten, runs the test
int Derivation::get_base_and_test(int index,
                int is_new,
                Result *result_p,
                Result_List *list_p,
                Error_Log *log_p)
{
    int status = 0;                         // processing status

    // run the test
    Object *guard_obj_p = guard_p->generate_value(&base_variables, log_p);
    if(guard_obj_p != NULL && !guard_obj_p->true())
    {
        status = 0;
    }
    // if there is one to get...
    else if(index < base_variables.size())
    {
        // Cycle through all possible values for the current base variable
        // and recursively call this function for the next variable.

        Result *base_result_p;              // result for this

        // find the result
        base_result_p = list_p->find_result(
                        base_variables[index]->type()->value());
        if(base_result_p == NULL)
        {
            log_p->log("Missing result for variable");
        }

        // get values of this
        List<Object> *base_objects_p = base_result_p->get_objects();
        int i;                                  // loop index

        // do for all objects...
        for(i=0; i<base_objects_p->size(); ++i)
        {
            if(index == 0)
                index = 0;
            if(index == 1)
                index = 1;

            base_variables[index]->set_value((*base_objects_p)[i]);
```

```
            if(get_base_and_test(index + 1,
                  base_result_p->is_new(i) | is_new, result_p, list_p,
                  log_p))
                status = 1;
            base_variables[index]->remove_value();
        }
    }
    else
    {
        Derived_Object *new_obj_p;                // new object for this

        new_obj_p = new Derived_Object(this);

        int i;                                    // loop index

        for(i=0; i<base_variables.size(); ++i)
        {
            new_obj_p->add_base(base_variables[i]->name(),
                              base_variables[i]->value()->clone());
        }

        for(i=0; i<superobjects.size(); ++i)
        {
            new_obj_p->add_superobject(superobjects[i]->value()->clone());
        }

        result_p->add(new_obj_p);
        status = 1;
    }

    // if there was a guard object
    if(guard_obj_p != NULL)
        delete guard_obj_p;

    return(status);
}
```

```
// dermthd.cpp

// Contains the code for Derived_Method, which is a method of
// a derived class.

#include "dermthd.h"
#include "list.h"
#include "variable.h"
#include "method.h"
#include "value.h"
#include "classpec.h"
#include "environ.h"


// Derived_Method() constructs this
Derived_Method::Derived_Method(Class_Spec *in_p, Environment *env_p)
{
    // initialize stuff
    value_p = NULL;

    Symbol *type_p;                // type of this

    // get this thing's type
    if(in_p->next_token() != Name)
    {
        in_p->log_error("Missing type name for method");
        return;
    }
    if((type_p = env_p->find_symbol(in_p->get_value())) == NULL)
    {
        in_p->log_error("Unknown type of method");
        return;
    }
    set_type(type_p);
    in_p->skip_token();

    // get name
    if(in_p->next_token() != Name)
    {
        in_p->log_error("Missing name of method");
        return;
    }
    set_name(in_p->get_value());
    in_p->skip_token();

    // if there are parameters...
    if(in_p->next_token() == Open_Paren)
    {
        in_p->skip_token();

        // do for all parameters
        while(in_p->next_token() != Close_Paren)
        {
            Variable *var_p;               // the parameters

            if((var_p = new Variable(in_p, env_p)) == NULL)
            {
```

```
                in_p->log_error("Invalid parameter");
                return;
            }
            parameters.add(var_p);

            // skip over any comma here
            if(in_p->next_token() != Close_Paren)
            {
                // should be comma
                if(in_p->next_token() != Comma)
                {
                    in_p->log_error("Invalid parameter list");
                    return;
                }
                in_p->skip_token();
            }
        }

        in_p->skip_token();
    }

    set_arity(parameters.size());

    // get IS
    if(in_p->next_token() != Is_Label)
    {
        in_p->log_error("Missing IS in derived method");
        return;
    }
    in_p->skip_token();

    // get value of this
    if((value_p = read_value(in_p, env_p)) == NULL)
    {
        in_p->log_error("Invalid value of derived method");
    }
}


// ~Derived_Method() destructs it
Derived_Method::~Derived_Method()
{
    if(value_p != NULL)
        delete value_p;
}


// parameters() gets parameters of this
List<Variable> *Derived_Method::get_parameters()
{
    return(&parameters);
}


// value() gets value of this
```

```
Value *Derived_Method::value()
{
    return(value_p);
}
```

```
// derobj.cpp

// Contains the code for Derived_Object, which implements a
// derived object.

#include <iostream.h>

#include "derobj.h"
#include "object.h"
#include "derivati.h"
#include "list.h"
#include "variable.h"
#include "dermthd.h"
#include "utils.h"

char *const superobject_name = "THIS";

// Derived_Object() constructs a derived object
Derived_Object::Derived_Object(Derivation *in_derivation_p)
{
    derivation_p = in_derivation_p;
}



// output() outputs an object
void Derived_Object::output(ostream &out, int indentation)
{
    indent(out, indentation);

    out << "Complex object (\n";

    int i;                  // loop index

    if(superobjects.size() > 0)
    {
        indent(out, indentation + 2);
        out << "IS\n";
        for(i=0; i<superobjects.size(); ++i)
            superobjects[i]->output(out, indentation + 4);
    }

    indent(out, indentation + 2);
    out << "BASED ON\n";

    // do for all base objects
    for(i=0; i<base_objects.size(); ++i)
    {
        // output the object
        base_objects[i]->output(out, indentation + 4);
    }

    indent(out, indentation);

    out << ")\n";
}
```

```
// add_base() adds a base object to the object
void Derived_Object::add_base(char *name, Object *value_p)
{
    base_objects.add(new Variable(name, value_p));
}


// clone() clones this object
Object *Derived_Object::clone()
{
    // construct an object for this
    Derived_Object *obj_p = new Derived_Object(derivation_p);

    int i;                          // loop index

    // do for all derived objects
    for(i=0; i<base_objects.size(); ++i)
    {
        obj_p->add_base(base_objects[i]->name(),
                        base_objects[i]->value()->clone());
    }

    // do for all base objects
    for(i=0; i<superobjects.size(); ++i)
    {
        obj_p->add_superobject(superobjects[i]->clone());
    }

    return(obj_p);
}



// true() returns that a derived object is always true
int Derived_Object::true()
{
    return(1);
}



// run_message() runs a message against the object
Object *Derived_Object::run_message(char *message, List<Object> *parms_p,
                Error_Log *log_p)
{
    Object *result_p = NULL;        // results of this
    Derived_Method *method_p = NULL;
                                    // the method to run

    if(derivation_p != NULL)
    {
        // find the method to use
        method_p = derivation_p->find_method(message, parms_p->size());
    }

    if(method_p == NULL)
    {
        // try to get it from superobjects
```

```
        int i;                          // loop index

        for(i=0; i<superobjects.size(); ++i)
        {
            // try this superobject
            if((result_p = superobjects[i]->run_message(message, parms_p,
                        log_p)) != NULL)
                break;
        }

        // if it was not found...
        if(result_p == NULL)
            log_p->log("Unknown method in the object");
    }
    else
    {
        // run the found method
        List<Variable> var_list;        // base variable holder
        int i;                                  // loop index

        // add this to resolve superobject stuff
        var_list.add(new Variable(superobject_name, clone()));

        // do for all base variables...
        for(i=0; i<base_objects.size(); ++i)
            var_list.add(base_objects[i]->clone());

        // do for all parameters...
        for(i=0; i<method_p->get_parameters()->size(); ++i)
        {
            Variable *v_p = (*method_p->get_parameters())[i]->clone();
            v_p->set_value((*parms_p)[i]->clone());
            var_list.add(v_p);
        }

        // generate the value in the method
        result_p = method_p->value()->generate_value(&var_list, log_p);
    }

    return(result_p);
}


// add_superobject() adds a superobject to the derived object
void Derived_Object::add_superobject(Object *value_p)
{
    superobjects.add(value_p);
}
```

```
// dlclause.cpp

// Contains the code for Dl_Clause, which maintains a single
// clause of a Datalog rule.

#include <stddef.h>
#include <string.h>

#include "dlenv.h"
#include "dlclause.h"
#include "dlpred.h"
#include "list.h"
#include "dlspec.h"
#include "dltuple.h"
#include "dlreslst.h"

// Dl_Clause() is a constructor
Dl_Clause::Dl_Clause(Dl_Spec  *in_p,  Dl_Environment  *env_p,  List<char>
*vars_p)
{
    // get the predicate name
    if(in_p->next_token() != Dl_Name)
    {
        in_p->log_error("Missing call name");
    }
    char *predicate_name;                // to hold the name
    predicate_name = new char[strlen(in_p->get_value()) + 1];
    strcpy(predicate_name, in_p->get_value());
    in_p->skip_token();

    // next token should be (
    if(in_p->next_token() != Dl_Open_Paren)
    {
        in_p->log_error("Missing open parenthesis");
    }
    in_p->skip_token();

    // do until at close paren
    while(in_p->next_token() != Dl_Close_Paren)
    {
        // if is name...
        if(in_p->next_token() == Dl_Name)
        {
            int i;                // loop index

            // find the variable if possible
            for(i=0; i<vars_p->size(); ++i)
            {
                if((*vars_p)[i] != NULL &&
                        strcmp(in_p->get_value(), (*vars_p)[i]) == 0)
                    break;
            }
            if(i >= vars_p->size())
            {
                // it's a new variable: add it to list
                char *name_p = new char[strlen(in_p->get_value()) + 1];
                strcpy(name_p, in_p->get_value());
                vars_p->add(name_p);
```

```
            }

            // store variable offset to this
            int *int_p = new int;
            *int_p = i;
            parm_variables.add(int_p);
            parm_values.add(NULL);
        }
        else if(in_p->next_token() == Dl_String
                || in_p->next_token() == Dl_Integer)
        {
            parm_variables.add(NULL);

            // copy and store the value
            char *value_p = new char[strlen(in_p->get_value()) + 1];
            strcpy(value_p, in_p->get_value());
            parm_values.add(value_p);
        }
        else
        {
            in_p->log_error("Invalid rule call parameter");
        }
        in_p->skip_token();

        // check next token is comma or )
        if(in_p->next_token() == Dl_Comma)
            in_p->skip_token();
        else if(in_p->next_token() != Dl_Close_Paren)
        {
            in_p->log_error("Missing close parenthesis");
        }
    }
    in_p->skip_token();

    // find the predicate
    if((predicate_p = env_p->find_predicate(predicate_name,
                        parm_variables.size())) == NULL)
    {
        in_p->log_error("Unknown parameter called");
    }
    delete[] predicate_name;
}



// is_complete() sees if clause is complete
int Dl_Clause::is_complete(Dl_Result_List *results_p)
{
    Dl_Result *result_p = results_p->find_result(predicate_p);

    return(result_p != NULL && result_p->is_complete());
}



// get_calls() adds calls of this to result list
```

```
void Dl_Clause::get_calls(Dl_Result_List *results_p)
{
    results_p->add_predicate(predicate_p);
}


// match() sees if the clause matches a tuple and variable set
//      It returns a list of new variable values set in this
List<int> *Dl_Clause::match(Dl_Tuple *tuple_p, List<char> *values_p)
{
    List<int> *v_list_p = new List<int>;        // return list
    int hit = 1;                                // is this a hit?
    int i;                                        // loop index

    // do for all parameters
    for(i=0; i<parm_values.size(); ++i)
    {
        // if this is a constant...
        if(parm_values[i] != NULL)
        {
            // if this is the thing...
            if(strcmp((*tuple_p)[i], parm_values[i]) != 0)
            {
                hit = 0;
                break;
            }
        }
        else
        {
            // if variable not set yet...
            if((*values_p)[*(parm_variables[i])] == NULL)
            {
                // set the variable value
                char *val_p = new char[strlen((*tuple_p)[i]) + 1];
                strcpy(val_p, (*tuple_p)[i]);
                values_p->set_entry(*(parm_variables[i]), val_p);

                // add variable offset to offset list
                int *i_p = new int;
                *i_p = *(parm_variables[i]);
                v_list_p->add(i_p);
            }
            else
            {
                // if this is not a match...
                if(strcmp((*tuple_p)[i],
                   (*values_p)[(*parm_variables[i])]) != 0)
                {
                    hit = 0;
                    break;
                }
            }
        }
    }

    // if not a hit...
    if(!hit)
```

```
    {
        // clear out values set herein
        for(i=0; i<v_list_p->size(); ++i)
        {
            values_p->clear(*((*v_list_p)[i]));
        }

        delete v_list_p;
        v_list_p = NULL;
    }

    return(v_list_p);
}


// get_predicate() gets a predicate of this
Dl_Predicate *Dl_Clause::get_predicate()
{
    return(predicate_p);
}
```

```
// dlenv.cpp

// Contains the code for Dl_Environment, which is the operational
// environment for the Datalog database.

#include <stddef.h>
#include <string.h>

#include "dlenv.h"
#include "list.h"
#include "dlpred.h"

// add_predicate() adds a predicate to the environment
void Dl_Environment::add_predicate(Dl_Predicate *pred_p)
{
    predicates.add(pred_p);
}



// find_predicate() finds a predicate
Dl_Predicate *Dl_Environment::find_predicate(char *name, int arity)
{
    int i;                          // loop index

    for(i=0; i<predicates.size(); ++i)
    {
        if(strcmp(predicates[i]->get_name(), name) == 0
                && predicates[i]->get_arity() == arity)
            return(predicates[i]);
    }

    return(NULL);
}
```

```
// dlog_db.cpp

// Contains code for Datalog_Database, which implements the
// gateway database for Datalog.

#include <iostream.h>

#include "dlog_db.h"
#include "dlenv.h"
#include "dlpred.h"

// DATALOG_INPUT_FILE contains the Datalog spec to use
#define DATALOG_INPUT_FILE "d:\\thesis\\roxanne\\data\\datalog.in"

// Datalog_Database() is the constructor for this
Datalog_Database::Datalog_Database(ostream &error_file)
{
    Dl_Spec spec(DATALOG_INPUT_FILE);       // input spec for the program

    // do as long as there is stuff in it
    while(spec.next_token() != Dl_End_of_Input && !spec.is_errors())
    {
        Dl_Rule *rule_p = new Dl_Rule(&spec, &environment);

        // if there is a failure...
        if(spec.is_errors())
            continue;

        Dl_Predicate *pred_p;                // the predicate of this

        // if the predicate of the rule is unknown...
        if((pred_p = find_predicate(rule_p->get_name(),
                    rule_p->get_arity())) == NULL)
        {
            // create it
            pred_p = new Dl_Predicate(rule_p->get_name(),
                  rule_p->get_arity());
            environment.add_predicate(pred_p);
        }

        // add the rule to the predicate
        pred_p->add_rule(rule_p);
    }

    // if there are errors...
    if(spec.is_errors())
    {
        spec.get_errors()->display_errors(error_file);
    }
}


// find_predicate() finds a predicate
Dl_Predicate *Datalog_Database::find_predicate(char *name, int arity)
{
    return(environment.find_predicate(name, arity));
}
```

```
// dlogclas.cpp

// Contains the code for Datalog_Class, which is a gateway to
// a Datalog class.

#include <stddef.h>

#include "dlogclas.h"
#include "list.h"
#include "classpec.h"
#include "environ.h"
#include "gwclass.h"
#include "errorlog.h"
#include "dlpred.h"
#include "dltuple.h"
#include "relation.h"
#include "bin_obj.h"
#include "binclass.h"

// Datalog_Class() is the constructor for this
Datalog_Class::Datalog_Class(char *name, Class_Spec *in_p, Environment
*env_p)
        : Gateway_Class(name, in_p, env_p)
{
    // make sure this is a Datalog class
    if(in_p->next_token() != Datalog_Label)
    {
        in_p->log_error(
            "Missing Datalog label in Datalog class specification");
        return;
    }
    in_p->skip_token();

    // read in the gateway class body
    read_gateway(in_p, env_p);

    // check error status
    if(in_p->is_errors())
        return;

    // find the predicate
    predicate_p = env_p->datalog_database()->
                find_predicate(name, methods.size());
    if(predicate_p == NULL)
    {
        in_p->log_error("Class has no corresponding Datalog predicate");
    }
}


// run() runs the class
List<Object> *Datalog_Class::run(Error_Log *log_p)
{
    List<Object> *results_p;                // the results of this
    List<Dl_Tuple> *answers_p;                // query answers

    // run the datalog query
    if((answers_p = predicate_p->run(log_p)) == NULL)
```

```
        return(NULL);

    // construct the results
    results_p = new List<Object>;

    int i;                  // loop index

    // do for all answers...
    for(i=0; i<answers_p->size(); ++i)
    {
        Relation *rel_p = new Relation;         // corresponding relation

        int j;                                  // loop index

        for(j=0; j<methods.size(); ++j)
        {

            rel_p->add_field(methods[j]->name(),
                new Builtin_Object(
                  (Builtin_Class *)methods[j]->type()->value(),
                                (*((*answers_p)[i]))[j]));
        }
        results_p->add(rel_p);
    }

    delete answers_p;

    return(results_p);
}
```

```
// dlpred.cpp

// Contains the code for Dl_Predicate, which implements a
// single Datalog predicate.

#include <stddef.h>
#include <string.h>

#include "dlenv.h"
#include "dlpred.h"
#include "dlreslst.h"
#include "dlresult.h"
#include "errorlog.h"
#include "list.h"
#include "dltuple.h"
#include "dlrule.h"

// Dl_Predicate() is a constructor for this
Dl_Predicate::Dl_Predicate(char *in_name, int in_arity)
{
    if(in_name != NULL)
    {
        name = new char[strlen(in_name) + 1];
        strcpy(name, in_name);
    }
    else
        name = NULL;

    arity = in_arity;
}


// ~Dl_Predicate is the destructor
Dl_Predicate::~Dl_Predicate()
{
    if(name != NULL)
        delete[] name;
}


// add_rule() adds a rule to the predicate
void Dl_Predicate::add_rule(Dl_Rule *rule_p)
{
    rules.add(rule_p);
}



// get_name() gets the name of the predicate
char *Dl_Predicate::get_name()
{
    return(name);
}



// get_arity() gets the arity of the predicate
int Dl_Predicate::get_arity()
```

```
{
    return(arity);
}


// run() runs the predicate, getting all tuples that satisfy it
List<Dl_Tuple> *Dl_Predicate::run(Error_Log *log_p)
{
    Dl_Result_List results;                 // the result list for this

    // setup the result list
    results.add_predicate(this);

    // run the predicate list
    results.run(log_p);

    if(log_p->is_errors())
        return(NULL);

    Dl_Result *result_p = results.find_result(this);

    if(result_p == NULL)
        return(NULL);

    return(result_p->extract());
}



// run_for_result() runs the predicate for a round
int Dl_Predicate::run_for_result(Dl_Result *dst_p,
      Dl_Result_List *results_p, int new_data, Error_Log *log_p)
{
    int found = 0;                  // track if anything has been found
    int i;                          // loop index

    // do for all rules of the predicate
    for(i=0; i<rules.size(); ++i)
    {
        if(rules[i]->run_for_result(dst_p, results_p, new_data, log_p))
            found = 1;
    }

    // if nothing found...
    if(!found)
    {
        int complete = 1;                   // is the class complete?

        // do for all rules
        for(i=0; i<rules.size(); ++i)
        {
            // if not complete...
            if(!rules[i]->is_complete(results_p))
            {
                // predicate is not complete
                complete = 0;
```

```
                break;
            }
        }
        if(complete)
            dst_p->mark_complete();
    }

    return(found);
}


// get_calls() gets all calls of this
void Dl_Predicate::get_calls(Dl_Result_List *results_p)
{
    int i;                  // loop index

    // get calls of all rules
    for(i=0; i<rules.size(); ++i)
    {
        rules[i]->get_calls(results_p);
    }
}
```

```
// dlreslst.cpp

// Contains the code for Dl_Result_List, which maintains a list
// of results from Datalog queries.

#include <stddef.h>
#include <string.h>

#include "dlenv.h"
#include "dlreslst.h"
#include "list.h"
#include "dlresult.h"
#include "dlpred.h"
#include "errorlog.h"

// add_predicate() adds a predicate to the list
void Dl_Result_List::add_predicate(Dl_Predicate *pred_p)
{
    // if not already on list...
    if(find_result(pred_p) == NULL)
    {
        results.add(new Dl_Result(pred_p));
        pred_p->get_calls(this);
    }
}


// find_result() finds a result for a predicate
Dl_Result *Dl_Result_List::find_result(char *name, int arity)
{
    int i;                  // loop index

    // look at all results to try to find the desired one
    for(i=0; i<results.size(); ++i)
    {
        if(strcmp(results[i]->get_predicate()->get_name(), name) == 0
                && results[i]->get_predicate()->get_arity() == arity)
            return(results[i]);
    }

    return(NULL);
}


// find_result() finds the result given the predicate
Dl_Result *Dl_Result_List::find_result(Dl_Predicate *pred_p)
{
    int i;                  // loop index

    // look at all results to try to find the desired one
    for(i=0; i<results.size(); ++i)
    {
        if(results[i]->get_predicate() == pred_p)
            return(results[i]);
    }
```

```
    return(NULL);
}


// run() runs all predicates in a result list
void Dl_Result_List::run(Error_Log *log_p)
{
    int done = 0;                  // are we done?
    int new_data = 1;                  // is there new data?

    // do until done
    while(!done)
    {
        // mark as done this time through
        done = 1;

        int i;                          // loop index

        // do for all results...
        for(i=0; i<results.size(); ++i)
        {
            // if result not done...
            if(!results[i]->is_complete())
            {
                // run the predicate, seeing if it added anything
                if(results[i]->get_predicate()->run_for_result(results[i],
                                        this, new_data, log_p))
                    done = 0;
            }
        }

        // end the cycle for all
        for(i=0; i<results.size(); ++i)
            results[i]->end_cycle();

        // note we're through first cycle
        new_data = 0;
    }
}
```

```
// dlresult.cpp

// Contains the code for Dl_Result, which is a single result
// of running a Datalog query.

#include "dlenv.h"
#include "dlresult.h"
#include "dlpred.h"
#include "list.h"
#include "dltuple.h"

// Dl_Result() constructs this
Dl_Result::Dl_Result(Dl_Predicate *pred_p)
{
    predicate_p = pred_p;
    complete = 0;
    new_tuple_offset = 0;
}



// extract() gets tuples out of this, passing over ownership
List<Dl_Tuple> *Dl_Result::extract()
{
    List<Dl_Tuple> *out_p;                  // output of this

    // construct output and move stuff from tuples to it
    out_p = new List<Dl_Tuple>;
    out_p->merge(&tuples);

    return(out_p);
}



// get_tuples() gets tuples from this without passing over ownership
List<Dl_Tuple> *Dl_Result::get_tuples()
{
    return(&tuples);
}



// add_tuple() adds a tuple to the list
void Dl_Result::add_tuple(Dl_Tuple *value_p)
{
    pending_tuples.add(value_p);
}



// get_predicate() gets the predicate of this
Dl_Predicate *Dl_Result::get_predicate()
{
    return(predicate_p);
}
```

```
// end_cycle() checkpoints this by noting the new tuples and
//    putting pending tuples to the final list
void Dl_Result::end_cycle()
{
    new_tuple_offset = tuples.size();
    tuples.merge(&pending_tuples);
}


// mark_complete() marks the result as finished
void Dl_Result::mark_complete()
{
    complete = 1;
}


// is_complete() checks if the result is complete
int Dl_Result::is_complete()
{
    return(complete);
}


// is_new() sees if a tuple is new (since last end_cycle() call)
int Dl_Result::is_new(int index)
{
    return(index >= new_tuple_offset);
}
```

```
// dlrule.cpp

// Contains the code for Dl_Rule, which implements a Datalog
// rule.

#include <stddef.h>
#include <string.h>

#include "dlenv.h"
#include "dlrule.h"
#include "dlclause.h"
#include "dlresult.h"
#include "dlreslst.h"
#include "list.h"
#include "errorlog.h"
#include "dlspec.h"



// Dl_Rule() is the constructor of a rule
Dl_Rule::Dl_Rule(Dl_Spec *in_p, Dl_Environment *env_p)
{
    // get name of rule
    if(in_p->next_token() != Dl_Name)
    {
        in_p->log_error("Missing rule name");
        return;
    }
    name = new char[strlen(in_p->get_value()) + 1];
    strcpy(name, in_p->get_value());
    in_p->skip_token();

    // check for open paren
    if(in_p->next_token() != Dl_Open_Paren)
    {
        in_p->log_error("Missing paren in rule");
        return;
    }
    in_p->skip_token();

    List<char> variables;                // list of variable names

    // do until end of parameters
    while(in_p->next_token() != Dl_Close_Paren)
    {
        if(in_p->next_token() == Dl_Name)
        {
            // it is a variable
            char *val_p = new char[strlen(in_p->get_value()) + 1];
            strcpy(val_p, in_p->get_value());
            variables.add(val_p);
            values.add(NULL);
        }
        else if(in_p->next_token() == Dl_String
                || in_p->next_token() == Dl_Integer)
        {
            // a constant: store in values
            char *val_p = new char[strlen(in_p->get_value()) + 1];
```

```
            strcpy(val_p, in_p->get_value());
            values.add(val_p);
            variables.add(NULL);
        }
        else
        {
            in_p->log_error("Invalid predicate parameter");
            return;
        }
        in_p->skip_token();

        // make sure next is ) or ,
        if(in_p->next_token() == Dl_Comma)
            in_p->skip_token();
        else if(in_p->next_token() != Dl_Close_Paren)
        {
            in_p->log_error("Invalid token in predicate");
            return;
        }
    }
    in_p->skip_token();

    arity = variables.size();

    // if is clauses...
    if(in_p->next_token() == Dl_Is_Defined)
    {
        in_p->skip_token();
        while(in_p->next_token() != Dl_Semicolon)
        {
            Dl_Clause *clause_p = new Dl_Clause(in_p, env_p, &variables);
            if(in_p->is_errors())
                return;
            clauses.add(clause_p);

            if(in_p->next_token() == Dl_Comma)
                in_p->skip_token();
            else if(in_p->next_token() != Dl_Semicolon)
            {
                in_p->log_error("Invalid character in rule definition");
                return;
            }
        }
    }

    // check for semicolon
    if(in_p->next_token() != Dl_Semicolon)
    {
        in_p->log_error("Missing semicolon on rule");
        return;
    }
    in_p->skip_token();
}



// ~Dl_Rule() destructs a rule
```

```
Dl_Rule::~Dl_Rule()
{
    if(name != NULL)
        delete[] name;
}


// get_name() gets name of this
char *Dl_Rule::get_name()
{
    return(name);
}



// get_arity() gets arity of this
int Dl_Rule::get_arity()
{
    return(arity);
}



// run_for_result() runs the rule and puts results in a Dl_Result
int Dl_Rule::run_for_result(Dl_Result *dst_p, Dl_Result_List *results_p,
                int new_data, Error_Log *log_p)
{
    List<char> answers;                 // the results of this
    int i;                          // loop index

    // do for all known results...
    for(i=0; i<values.size(); ++i)
    {
        if(values[i] != NULL)
        {
            char *val_p = new char[strlen(values[i]) + 1];
            strcpy(val_p, values[i]);
            answers.add(val_p);
        }
        else
            answers.add(NULL);
    }

    return(test_all(0, new_data, dst_p, results_p, &answers, log_p));
}



// is_complete() sees if the rule is complete
int Dl_Rule::is_complete(Dl_Result_List *results_p)
{
    int i;                          // loop index

    for(i=0; i<clauses.size(); ++i)
    {
        if(!clauses[i]->is_complete(results_p))
            return(0);
    }
```

```
    return(1);
}


// get_calls() gets all calls called by this rule
void Dl_Rule::get_calls(Dl_Result_List *results_p)
{
    int i;                // loop index

    for(i=0; i<clauses.size(); ++i)
        clauses[i]->get_calls(results_p);
}


// test_all() does the work of running the rule
int Dl_Rule::test_all(int index, int new_data, Dl_Result *result_p,
        Dl_Result_List *list_p, List<char> *vals_p, Error_Log *log_p)
{
    int status = 0;               // processing status

    if(index >= clauses.size())
    {
        // all clauses run - add as tuple if new
        if(new_data)
        {
            Dl_Tuple *tuple_p = new Dl_Tuple;        // new tuple
            int i;                                  // loop index

            // construct the tuple from found values
            for(i=0; i<values.size(); ++i)
            {
                tuple_p->add_value((*vals_p)[i]);
            }

            // add the tuple
            result_p->add_tuple(tuple_p);
            status = 1;
        }
    }
    else
    {
        // not all clauses run - test this one and call for next one
        Dl_Result *base_result_p;        // result for this clause
        int i;                                  // loop index

        // find the predicate for this clause
        base_result_p =
            list_p->find_result(clauses[index]->get_predicate());

        // test all tuples
        for(i=0; i<base_result_p->get_tuples()->size(); ++i)
        {
            List<int> *var_sets_p;        // offsets to gotten variables

            // see if the current tuple matches the clause
            if((var_sets_p = clauses[index]->match(
                        (*base_result_p->get_tuples())[i], vals_p))
```

```
                    != NULL)
            {
                // recurse with this one
                if(test_all(index + 1,
                        base_result_p->is_new(i) | new_data,
                        result_p, list_p, vals_p, log_p))
                    status = 1;

                // clear variables that were set
                int j;                  // loop index

                for(j=0; j<var_sets_p->size(); ++j)
                    vals_p->clear(*((*var_sets_p)[j]));

                delete var_sets_p;
            }
        }
    }

    return(status);
}
```

```
// dlspec.cpp

// Contains the code for Dl_Spec, which is the lexical analyzer
// for the Datalog system.

#include <iostream.h>
#include <fstream.h>
#include <string.h>
#include <io.h>

#include "dlspec.h"
#include "errorlog.h"

static const int max_token_len = 256;           // max length of a token

// Dl_Spec() constructs this
Dl_Spec::Dl_Spec(char *in_filename)
{
    // store the filename
    filename = new char[strlen(in_filename) + 1];
    strcpy(filename, in_filename);

    // create the input spec
    token_input_p = new ifstream(filename);

    // fill in the state table
    fill_in_state_table();

    // allocate space for token value
    token_value_buffer = new char[max_token_len + 1];

    // get the first token
    skip_token();
}



// ~Dl_Spec() destructs this
Dl_Spec::~Dl_Spec()
{
    delete token_input_p;

    if(token_value_buffer != NULL)
        delete[] token_value_buffer;

    if(filename != NULL)
        delete[] filename;
}


// next_token() gets the type of the next token
dl_token_type Dl_Spec::next_token()
{
    return(current_token);
}
```

```
// the following enumeration defines the states to simplify reading
// Capital letters following St_ are characters matched at that state
enum state_types
        {St_base, St_name, St_semicolon, St_comma, St_integer, St_colon,
         St_is_defined, St_open_paren, St_close_paren, St_string_open,
         St_string_close, St_string};

// skip_token() skips over the current token
void Dl_Spec::skip_token()
{
    // skip over any blanks
    skip_blanks();

    // initialize token destination stuff
    char *token_p = token_value_buffer;          // ptr at current token
    state_types current_state = St_base;         // current state in FSA
    state_types last_state = St_base;            // last state found

    // initialize token value at top of buffer
    token_value_p = token_value_buffer;

    // do until done
    while(current_state != no_state)
    {
        last_state = current_state;
        if(token_input_p->eof())
            current_state = (state_types)-1;
        else
        {
            token_input_p->get(*token_p);
            current_state =
              (state_types)state_table.get_next_state(last_state,
                                                      *token_p);
        }
        ++token_p;
    }

    if(!token_input_p->eof())
        token_input_p->putback(*(token_p - 1));

    if((current_token = (dl_token_type)state_table.get_type(last_state))
                == (dl_token_type)no_type)
        current_token = Dl_End_of_Input;

    *(token_p - 1) = '\0';

    // if current token is a string, strip quotes
    if(current_token == Dl_String)
    {
        token_value_p++;
        *(token_p - 2) = '\0';
    }
}


// get_value() gets the value of the next token
char *Dl_Spec::get_value()
{
```

```
    return(token_value_p);
}



// the following defines a useful function for state table stuff
static void create_transitions(State_Table &, int, int,
                        unsigned char, unsigned char);
static void create_name_transitions(State_Table &, int, int);

// fill_in_state_table() fills in the state table as needed
void Dl_Spec::fill_in_state_table()
{
    // do for all states in great tedium
    create_name_transitions(state_table, St_base, St_name);
    state_table.add_transition(St_base, ';', St_semicolon);
    state_table.add_transition(St_base, ':', St_colon);
    state_table.add_transition(St_base, '(', St_open_paren);
    state_table.add_transition(St_base, ')', St_close_paren);
    state_table.add_transition(St_base, ',', St_comma);
    state_table.add_transition(St_base, '"', St_string_open);
    create_transitions(state_table, St_base, St_integer, '0', '9');
    create_transitions(state_table, St_integer, St_integer, '0', '9');

    create_name_transitions(state_table, St_name, St_name);
    create_transitions(state_table, St_string_open, St_string_open, 0,
                  127);
    state_table.add_transition(St_string_open, '"', St_string);
    state_table.add_transition(St_colon, '-', St_is_defined);

    // create end states
    state_table.set_type(St_name, Dl_Name);
    state_table.set_type(St_semicolon, Dl_Semicolon);
    state_table.set_type(St_open_paren, Dl_Open_Paren);
    state_table.set_type(St_close_paren, Dl_Close_Paren);
    state_table.set_type(St_comma, Dl_Comma);
    state_table.set_type(St_string, Dl_String);
    state_table.set_type(St_integer, Dl_Integer);
    state_table.set_type(St_is_defined, Dl_Is_Defined);
}


// skip_blanks() skips over blanks
void Dl_Spec::skip_blanks()
{
    char ch = ' ';                  // character read in
    int in_comment = 0;                  // are we in a comment?

    while(!token_input_p->eof() &&
        (ch == ' ' || ch == '\t' || ch == '\n' || ch == '{'
                || in_comment))
    {
        if(ch == '{')
            in_comment++;
        if(ch == '}')
            in_comment--;
        token_input_p->get(ch);
```

```
    }

    if(!token_input_p->eof())
        token_input_p->putback(ch);
}


// create_name_transitions() creates transition to name state
static void create_name_transitions(State_Table &table, int from_state,
        int to_state)
{
    create_transitions(table, from_state, to_state, 'A', 'Z');
    create_transitions(table, from_state, to_state, 'a', 'z');
    create_transitions(table, from_state, to_state, '0', '9');
    table.add_transition(from_state, '_', to_state);
}


// create_transitions() fills in state stuff for range of character
static void create_transitions(State_Table &table,
                               int from_state,
                               int to_state,
                               unsigned char start_ch,
                               unsigned char end_ch)
{
    unsigned char ch;                          // alpha characters

    for(ch = start_ch; ch <= end_ch; ++ch)
        table.add_transition(from_state, ch, to_state);
}



// log_error() records an error
void Dl_Spec::log_error(char *error)
{
    error_log.log(error);
}



// is_errors() returns the state of the error flag
int Dl_Spec::is_errors()
{
    return(error_log.is_errors());
}


// get_errors() gets the Error_Log
Error_Log *Dl_Spec::get_errors()
{
    return(&error_log);
}


// text() gets current text of everything
char *Dl_Spec::text()
```

```
{
    ifstream in(filename);        // to hold input file
    long size = filelength(in.rdbuf()->fd());
                                  // length of input
    char *contents = new char[size + 1];
                                  // file contents

    in.read(contents, size);
    contents[in.gcount()] = '\0';

    return(contents);
}
```

```
// dltuple.cpp

// Contains the code for Dl_Tuple, which the the result of
// running a Datalog query.

#include <stddef.h>
#include <string.h>

#include "dltuple.h"
#include "list.h"



// add_value() adds a value to a tuple
void Dl_Tuple::add_value(char *value)
{
    char *new_value_p = new char[strlen(value) + 1];
                                  // the value

    strcpy(new_value_p, value);

    values.add(new_value_p);
}



// operator[] gets an element of a tuple
char *Dl_Tuple::operator[](int offset)
{
    return(values[offset]);
}



// size() gets the size of the tuple
int Dl_Tuple::size()
{
    return(values.size());
}
```

```
// editwin.cpp

// Contains the code for Edit_Window, which allows a user to
// edit a file.

#include <owl\framewin.h>
#include <owl\edit.h>
#include <owl\dialog.h>
#include <fstream.h>
#include <io.h>

#include "editwin.h"
#include "roxwin.h"

DEFINE_RESPONSE_TABLE1(Edit_Window, TDialog)
    EV_CHILD_NOTIFY(IDC_EXIT, BN_CLICKED, handle_save_exit),
    EV_CHILD_NOTIFY(IDC_QUIT, BN_CLICKED, CmOk),
END_RESPONSE_TABLE;



// Edit_Window() is the constructor for an edit window
Edit_Window::Edit_Window(TWindow *parent, char *file)
        : TDialog(parent, EDIT_WINDOW)
{
    // store filename and put on as caption
    strcpy(filename, file);
    SetCaption(file);

    // create an edit box
    edit_box = new TEdit(this, ID_EDIT_BOX, 0);
}



// handle_save_exit() handles the save,exit button
void Edit_Window::handle_save_exit()
{
    ofstream out_file(filename, ios::out | ios::binary);
                                          // output file
    char far *buffer = edit_box->LockBuffer();
                                          // buffer of data

    // write out stuff
    out_file.write(buffer, strlen(buffer));

    // lose the lock buffer
    edit_box->UnlockBuffer(buffer);

    // Exit the window
    CmOk();
}



// SetupWindow() overrides normal SetupWindow() function
void Edit_Window::SetupWindow()
{
```

```
    // do normal SetupWindow()
    TDialog::SetupWindow();

    //
    // fill edit box
    //

    // if the file exists...
    if(access(filename, 0) == 0)
    {
        ifstream in_file(filename, ios::in | ios::binary);
                                                    // the file
        long size = filelength(in_file.rdbuf()->fd());
                                                    // length of file
        char *buffer = edit_box->LockBuffer(size + 1);
                                                    // data buffer

        // read the buffer
        in_file.read(buffer, size);
        buffer[size] = '\0';

        // unlock and close it
        edit_box->UnlockBuffer(buffer, TRUE);

        // clear the selection
        edit_box->SetSelection(12,15);
    }
}
```

```
// environ.cpp

// Contains the code for Environment, which maintains the
// system environment, including the symbol table and classes
// to access the member databases.

#include <stddef.h>
#include <string.h>
#include <iostream.h>

#include "environ.h"
#include "symbol.h"
#include "list.h"
#include "baseimps.h"
#include "pdox_db.h"
#include "dlog_db.h"


// Environment() constructs the environment
Environment::Environment(ostream &errors)
{
    pdox_database_p = new Paradox_Database;
    dlog_database_p = new Datalog_Database(errors);
}


// ~Environment() destructs the environment
Environment::~Environment()
{
    delete pdox_database_p;
    delete dlog_database_p;
}



// find_symbol() finds a specific symbol
Symbol *Environment::find_symbol(char *name)
{
    int i;                          // loop index

    for(i=0; i<symbols.size(); ++i)
    {
        if(strcmp(name, symbols[i]->name()) == 0)
            return(symbols[i]);
    }

    return(NULL);
}


// add() adds a symbol to the list
void Environment::add_symbol(Symbol *symbol_p)
{
    symbols.add(symbol_p);
}
```

```
// Paradox_Database() gets the Paradox database
Paradox_Database *Environment::paradox_database()
{
    return(pdox_database_p);
}


// datalog_database() gets the Datalog database
Datalog_Database *Environment::datalog_database()
{
    return(dlog_database_p);
}
```

```
// errorlog.cpp

// Contains the code for Error_Log, which logs errors.

#include <string.h>
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>

#include "errorlog.h"
#include "list.h"

// is_errors() tells if there are errors
int Error_Log::is_errors()
{
    return(errors.size() > 0);
}


// display_errors() displays all errors in a window
void Error_Log::display_errors(ostream &Output)
{
    // do for all errors
    for(int i=0; i<errors.size(); ++i)
    {
        Output << errors[i] << "\r\n";
    }
}



// log() adds an error
void Error_Log::log(char *error)
{
    char *tmp = new char[strlen(error) + 1];        // holding variable

    strcpy(tmp, error);

    errors.add(tmp);
}
```

```
// filedisp.cpp

// contains the code for File_Display, which displays the
// contents of a buffer to a window.

#include <owl\framewin.h>
#include <owl\edit.h>
#include <owl\dialog.h>
#include <fstream.h>
#include <io.h>
#include <string.h>

#include "filedisp.h"
#include "roxwin.h"

DEFINE_RESPONSE_TABLE1(File_Display, TDialog)
    EV_CHILD_NOTIFY(IDOK, BN_CLICKED, CmOk),
END_RESPONSE_TABLE;



// File_Display() is the constructor for a file display window
File_Display::File_Display(TWindow *parent, char *label, char *buffer)
        : TDialog(parent, FILE_DISPLAY)
{
    // store filename and put on as caption
    display_buffer = new char[strlen(buffer) + 1];
    strcpy(display_buffer, buffer);
    SetCaption(label);

    // create an edit box
    edit_box = new TEdit(this, FILE_DISPLAY_TEXT, 0);
}



// ~File_Display() is the destructor of this
File_Display::~File_Display()
{
    if(display_buffer != NULL)
        delete[] display_buffer;
}



// SetupWindow() overrides normal SetupWindow() function
void File_Display::SetupWindow()
{
    // do normal SetupWindow()
    TDialog::SetupWindow();

    //
    // fill edit box
    //

    char *buffer = edit_box->LockBuffer(strlen(display_buffer) + 1);
                                                    // data buffer
```

```
    // put in the buffer
    strcpy(buffer, display_buffer);

    // unlock and close it
    edit_box->UnlockBuffer(buffer, TRUE);

    // clear the selection
    edit_box->SetSelection(12,15);
}
```

```
// gwclass.cpp

// Contains code for Gateway_Class, which represents a Cyrano
// gateway class.

#include "gwclass.h"
#include "classpec.h"
#include "environ.h"
#include "class.h"
#include "list.h"
#include "method.h"
#include "pdoxclas.h"
#include "dlogclas.h"


// Gateway_Class() is the constructor for this
Gateway_Class::Gateway_Class(char *name, Class_Spec *in,
                Environment *env_p) : Class(name, in, env_p)
{
}


// Gateway_Class() is the destructor for this
Gateway_Class::~Gateway_Class()
{
    superclasses.empty();
}


// read_gateway() reads the body of a gateway class
void Gateway_Class::read_gateway(Class_Spec *in_p, Environment *env_p)
{
    // get superclasses, if any are there
    if(in_p->next_token() == Is_Label)
    {
        do
        {
            in_p->skip_token();
            if(in_p->next_token() != Name)
            {
                in_p->log_error("Missing superclass name");
                return;
            }
            Symbol *sym_p = env_p->find_symbol(in_p->get_value());
            if(sym_p == NULL)
            {
                in_p->log_error("Unknown superclass");
                return;
            }
            superclasses.add(sym_p);
            in_p->skip_token();
        } while(in_p->next_token() == Comma);
    }

    // get WITH label
    if(in_p->next_token() != With_Label)
    {
        in_p->log_error("Missing WITH label in Gateway Class");
```

```
            return;
        }
        in_p->skip_token();

        // do as long as next token indicates a method
        while(in_p->next_token() == Name)
        {
            // get the method
            Method *method_p = new Method(in_p, env_p);

            // check that there are no errors
            if(in_p->is_errors())
            {
                delete method_p;
                return;
            }

            methods.add(method_p);

            // get next semi-colon
            if(in_p->next_token() != Semicolon)
            {
                in_p->log_error("Missing semicolon on method");
                return;
            }
            in_p->skip_token();
        }
}



// read_gateway_class() is not a member: rather, it reads
//        any type of gateway class
Gateway_Class *read_gateway_class(char *name, Class_Spec *in,
                Environment *env_p)
{
    // check that this is GATEWAY
    if(in->next_token() != Gateway_Label)
    {
        in->log_error("Missing GATEWAY label in Gateway Class");
        return(NULL);
    }
    in->skip_token();

    // check that this is :
    if(in->next_token() != Colon)
    {
        in->log_error("Missing colon in Gateway Class definition");
        return(NULL);
    }
    in->skip_token();

    // do based on next token
    Gateway_Class *class_p = NULL;
    switch(in->next_token())
    {
        case Paradox_Label:
```

```
            class_p = new Paradox_Class(name, in, env_p);
            break;

        case Datalog_Label:
            class_p = new Datalog_Class(name, in, env_p);
            break;

        default:
            in->log_error("Unknown Gateway Class type");
            return(NULL);
    }

    // if errors have been encountered...
    if(in->is_errors())
    {
        // undo everything
        delete class_p;
        class_p = NULL;
    }

    return(class_p);
}
```

```
// list.cpp

// Contains code for List, which is a template for lists of
// data items.

#include <stddef.h>

#include "list.h"

#define ALLOC_QUANTUM           10          // allocate this much stuff



// Base_List() constructs a list
Base_List::Base_List()
{
    entries_pp = NULL;
    n_entries = 0;
    allocated_size = 0;
}


// ~Base_List() destructs a list
Base_List::~Base_List()
{
    if(entries_pp != NULL)
        delete[] entries_pp;
}


// size() gets size of list
int Base_List::size()
{
    return(n_entries);
}


// add() adds an entry to the list
void Base_List::add(void *entry_p)
{
    if(n_entries + 1 > allocated_size)
    {
        void **new_list_pp = new void *[allocated_size + ALLOC_QUANTUM];

        int i;                  // loop index

        for(i=0; i<n_entries; ++i)
                new_list_pp[i] = entries_pp[i];

        if(entries_pp != NULL)
                delete[] entries_pp;

        entries_pp = new_list_pp;

        allocated_size += ALLOC_QUANTUM;
    }
```

```
    entries_pp[n_entries++] = entry_p;
}


// remove() deletes an entry from the list
void Base_List::remove(int offset)
{
    // move all entries past this one down one
    int i;                          // loop index

    for(i=offset + 1; i < n_entries; ++i)
        entries_pp[i - 1] = entries_pp[i];

    // decrement size
    n_entries--;
}


// [] gets an entry off of the list
void *Base_List::operator[](int offset)
{
    static void *result_p = NULL;                // a constant result

    if(offset < 0 || offset >= n_entries)
        return(result_p);
    else
        return(entries_pp[offset]);
}


// merge() merges contents of one list to another
void Base_List::merge(Base_List *mergelist)
{
    int i;                  // loop index

    // do for all entries on base list
    for(i=0; i<mergelist->size(); ++i)
    {
        // move entry to target list
        add((*mergelist)[i]);
    }

    // do for all entries on merge list from rear
    for(i=mergelist->size() - 1; i >= 0; --i)
    {
        // remove it
        mergelist->remove(i);
    }
}


// set_entry() sets a specified entry to a specified value
void Base_List::set_entry(int index, void *value)
{
```

```
    // do as long as index is past the end of the array
    while(index >= n_entries)
    {
        add(NULL);
    }

    // set the value of the thing accordingly
    entries_pp[index] = value;
}


// clear() clears a specified value
void Base_List::clear(int index)
{
    if(index < n_entries)
            entries_pp[index] = NULL;
}



// empty() empties out a list without trashing its members
void Base_List::empty()
{
    n_entries = 0;
}
```

```
// method.cpp

// Contains code for Method, which is a method of a derived
// class.

#include <stddef.h>
#include <string.h>

#include "method.h"
#include "classpec.h"
#include "environ.h"
#include "symbol.h"


// Method() constructs a method from a specification of it
Method::Method(Class_Spec *in, Environment *env_p)
{
    // preset values unless have to exit quickly
    method_name = NULL;
    method_type_p = NULL;
    method_arity = 0;

    // get the type
    if(in->next_token() != Name)
    {
        in->log_error("Missing Method type");
        return;
    }
    if((method_type_p = env_p->find_symbol(in->get_value())) == NULL
                || method_type_p->value() == NULL)
    {
        in->log_error("Unknown class of Method");
        return;
    }
    in->skip_token();

    // get the method name
    if(in->next_token() != Name)
    {
        in->log_error("Missing Method name");
        return;
    }
    method_name = new char[strlen(in->get_value()) + 1];
    strcpy(method_name, in->get_value());
    in->skip_token();

    // if there is an arity...
    if(in->next_token() == Open_Paren)
    {
        in->skip_token();

        // get the arity
        if(in->next_token() == Integer)
        {
            method_arity = atoi(in->get_value());
            in->skip_token();
        }
```

```
        // make sure closing paren is there
        if(in->next_token() != Close_Paren)
        {
            in->log_error("Missing close parentheses in Method");
            return;
        }
        in->skip_token();
    }
}


// Method() constructs a method from nothing
Method::Method(Symbol *type_p, char *name, int arity)
{
    method_type_p = type_p;

    if(name == NULL)
        method_name = NULL;
    else
    {
        method_name = new char[strlen(name) + 1];
        strcpy(method_name, name);
    }

    method_arity = arity;
}


// ~Method() destructs a method
Method::~Method()
{
    if(method_name != NULL)
        delete[] method_name;
}


// set_name() sets the name of a method
void Method::set_name(char *in_name)
{
    if(method_name != NULL)
        delete[] method_name;

    method_name = new char[strlen(in_name) + 1];
    strcpy(method_name, in_name);
}


// set_arity() sets arity of the method
void Method::set_arity(int in_arity)
{
    method_arity = in_arity;
}


// set_type() sets the type of a method
void Method::set_type(Symbol *type_p)
{
    method_type_p = type_p;
```

```
}


// name() gets the name of the method
char *Method::name()
{
    return(method_name);
}


// arity() gets the arity of the method
int Method::arity()
{
    return(method_arity);
}


// type() gets the type of the method
Symbol *Method::type()
{
    return(method_type_p);
}
```

```
// object.cpp

// Contains code for Object, which is any Cyrano object.

#include <iostream.h>

#include "object.h"

// ~Object is a destructor for this
Object::~Object()
{
}



// << outputs an object
ostream &operator<< (ostream &out, Object &object)
{
    object.output(out);
    return(out);
}
```

```
// pdox_db.cpp

// Contains code for Paradox_Database, which provides the
// interface to the Paradox database.

#include <bdatabas.h>
#include <bengine.h>
#include <envdef.h>

#include "pdox_db.h"


// Paradox_Database() constructs the database
Paradox_Database::Paradox_Database()
{
    BEnv environment;                // the environment for this

    // get the engine
    engine_p = new BEngine;

    // get current environment
    engine_p->getDefaults(environment);

    // set environment as needed
    environment.engineType = pxWin;
    environment.winShare = pxSingleClient;
    engine_p->setDefaults(environment);

    // open the engine
    engine_p->open();

    // construct the database
    database_p = new BDatabase(engine_p);
}


// ~Paradox_Database() destructs the database
Paradox_Database::~Paradox_Database()
{
    delete database_p;
    engine_p->close();
    delete engine_p;
}



// get_engine() gets the database engine
BEngine *Paradox_Database::get_engine()
{
    return(engine_p);
}

// get_database() gets the database
BDatabase *Paradox_Database::get_database()
{
    return(database_p);
}
```

```
// pdoxclas.cpp

// Contains code for Paradox_Class, which is a gateway class
// that serves as a gateway to the Paradox database.

#include <bcursor.h>
#include <bdatabase.h>
#include <stdio.h>

#include "pdoxclas.h"
#include "list.h"
#include "classpec.h"
#include "environ.h"
#include "gwclass.h"
#include "errorlog.h"
#include "relation.h"
#include "bin_obj.h"

// the following are lengths of field types
static const int short_field_length = 12;
static const int double_field_length = 12;
static const int date_field_length = 12;

// Paradox_Class() constructs a Paradox class
Paradox_Class::Paradox_Class(char  *name,  Class_Spec  *in,  Environment
*env_p)
        : Gateway_Class(name, in, env_p)
{
    // initialize fields
    cursor_p = NULL;
    record_p = NULL;

    // make sure this is a Paradox class
    if(in->next_token() != Paradox_Label)
    {
        in->log_error(
            "Missing Paradox label in Paradox class specification");
        return;
    }
    in->skip_token();

    // read in the gateway class body
    read_gateway(in, env_p);

    // check error status
    if(in->is_errors())
        return;

    // note the database
    database_p = env_p->paradox_database();

    // see if is possible to open the cursor
    open_cursor(in->get_errors());
    close_cursor();
}
```

```
// ~Paradox_Class() destructs a Paradox class
Paradox_Class::~Paradox_Class()
{
    close_cursor();
}



// run() finds all members of this class
List<Object> *Paradox_Class::run(Error_Log *log_p)
{
    // open the cursor
    open_cursor(log_p);
    if(log_p->is_errors())
        return(NULL);

    // construct an object list
    List<Object> *results_p = new List<Object>;

    // rewind the cursor
    cursor_p->gotoBegin();

    // do as long as there is stuff in the cursor
    while(cursor_p->gotoNext() == PXSUCCESS)
    {
        // read the field
        Relation *relation_p;                // the relation read in
        if((relation_p = read_record(log_p)) == NULL)
        {
            delete results_p;
            close_cursor();
            return(NULL);
        }

        results_p->add(relation_p);
    }

    close_cursor();

    return(results_p);
}



// read_record() reads in a single record
Relation *Paradox_Class::read_record(Error_Log *log_p)
{
    // construct an object for this
    Relation *relation_p = new Relation;

    int return_code;                        // return from function

    // get the record
    if((return_code = cursor_p->getRecord(record_p)) != PXSUCCESS)
    {
        delete relation_p;
        char error[200];
```

```
    sprintf(error,
            "Failure in Paradox getRecord with code %d: %s",
            return_code,
            PXOopErrMsg(return_code));
    log_p->log(error);
}

// do for all fields in this
int i;                          // loop index
for(i=0; i<methods.size(); ++i)
{
    // find the field
    int field_number = record_p->getFieldNumber(methods[i]->name());
    if(record_p->lastError != PXSUCCESS)
    {
        char error[200];                // error string
        sprintf(error,
            "Fail to find field %s with error code %d: %s",
            methods[i]->name(), record_p->lastError,
            PXOopErrMsg(record_p->lastError));
        log_p->log(error);
        delete relation_p;
        return(NULL);
    }

    // get the field type and length
    PXFieldType fld_type;               // type of field
    PXFieldSubtype fld_subtype;         // subtype of field
    int fld_len;                        // length of field

    record_p->getFieldDesc(field_number, fld_type, fld_subtype,
                    fld_len);

    char *data;                 // data buffer for this

    // if it is a blob...
    if(fld_type == fldBlob)
    {
        // handle appropriately
        if((data = get_blob(methods[i]->name(),
                    field_number, log_p)) == NULL)
        {
            delete relation_p;
            return(NULL);
        }
    }
    else
    {
        if((data = get_field(methods[i]->name(),
                            field_number,
                            fld_type,
                            fld_len,
                            log_p)) == NULL)
        {
            delete relation_p;
            return(NULL);
        }
    }
```

```
            relation_p->add_field(methods[i]->name(),
                new Builtin_Object(
                      (Builtin_Class *)methods[i]->type()->value(),
                                  data));

            delete data;
        }

        return(relation_p);
}



// read a Blob from the Paradox database
char *Paradox_Class::get_blob(char *name, int number, Error_Log *log_p)
{
        int return_code;                // return code from stuff

        // open the blob
        if((return_code = record_p->openBlobRead(number)) != PXSUCCESS)
        {
            char error[200];                // error string

            sprintf(error,
              "Failure to open Paradox blob %s with error code %d: %s",
              name, return_code, PXOopErrMsg(return_code));
            log_p->log(error);

            return(NULL);
        }

        // get its size
        long blob_size = record_p->getBlobSize(number);

        // allocate buffer for it
        char *data = new char[blob_size + 1];

        // read it in
        record_p->getBlob(number, blob_size, 0, data);

        // close the blob
        record_p->closeBlob(number);

        return(data);
}


// get a non-Blob field from Paradox
char *Paradox_Class::get_field(char *name,
                               int number,
                               PXFieldType type,
                               int fld_len,
                               Error_Log *log_p)
{
        // set the field length based on its type

        // do based on the field type
```

```
    switch(type)
    {
        case fldChar:
            fld_len++;
            break;

        case fldShort:
            fld_len = short_field_length;
            break;

        case fldDouble:
            fld_len = double_field_length;
            break;

        case fldDate:
            fld_len = date_field_length;
            break;
    }

    // get data buffer
    char *data = new char[fld_len + 1];

    // read it in
    BOOL fld_null;                  // is the field null?
    if(record_p->getField(number, data, fld_len, fld_null)
                != PXSUCCESS)
    {
        char error[200];                // error string

        sprintf(error, "Failure to get field %s with code %d: %s",
                name, record_p->lastError,
               PXOopErrMsg(record_p->lastError));
        log_p->log(error);
        delete data;
        return(NULL);
    }

    // null terminate
    if(fld_null)
        data[0] = '\0';
    else
        data[fld_len] = '\0';

    return(data);
}



// open_cursor() opens the cursor and record
void Paradox_Class::open_cursor(Error_Log *log_p)
{
    // get a cursor for this
    cursor_p = new BCursor(database_p->get_database(), name());
    if(!cursor_p->isOpen)
    {
        char error[200];                        // to hold error string

        sprintf(error,
```

```
                "Failure to open cursor with error %d: %s",
                cursor_p->lastError,
                PXOopErrMsg(cursor_p->lastError));
        log_p->log(error);
    }

    // get a record
    record_p = new BRecord(cursor_p);
}


// close_cursor() closes the cursor
void Paradox_Class::close_cursor()
{
    if(cursor_p != NULL)
    {
        delete cursor_p;
        cursor_p = NULL;
    }
    if(record_p != NULL)
    {
        delete record_p;
        record_p = NULL;
    }
}
```

```
// quanval.cpp

// Contains code for Quantified_Value, which is a value that
// consists of a quantifier (e.g., FORALL, EXISTS) and another
// value.

#include <stddef.h>

#include "quanval.h"
#include "value.h"
#include "list.h"
#include "variable.h"
#include "errorlog.h"
#include "classpec.h"
#include "environ.h"
#include "symbol.h"
#include "baseimps.h"
#include "object.h"
#include "bin_obj.h"

// Quantified_Value() is a constructor
Quantified_Value::Quantified_Value(Class_Spec *in_p, Environment *env_p)
{
    var_p = NULL;
    body_p = NULL;

    // find the boolean class
    if((boolean_class_p = env_p->find_symbol(boolean_class_name))
                        == NULL)
    {
        in_p->log_error("Missing BOOLEAN class");
    }
    else
    {
        // pick out the quantifier
        switch(in_p->next_token())
        {
            case Forall_Label:
                quantifier = Q_Forall;
                break;

            case Exists_Label:
                quantifier = Q_Exists;
                break;

            default:
                in_p->log_error("Unknown quantifier");
                break;
        }

        // if all okay...
        if(!in_p->is_errors())
        {
            // skip the quantifier
            in_p->next_token();

            // get the variable of this
            var_p = new Variable(in_p, env_p);
```

```
            // if okay so far...
            if(!in_p->is_errors())
            {
                body_p = read_value(in_p, env_p);
            }
        }
    }
}


// ~Quantified_Value() destructs this
Quantified_Value::~Quantified_Value()
{
    if(var_p != NULL)
        delete var_p;
    if(body_p != NULL)
        delete body_p;
}


// generate_value() generates the value for this
Object *Quantified_Value::generate_value(List<Variable> *vars_p,
                                         Error_Log *log_p)
{
    // if the variable of this has an undefined class...
    if(var_p->type()->value() == NULL)
    {
        log_p->log("Attempt to quantify on undefined variable type");
        return(NULL);
    }

    // get possible values of the quantified variable
    List<Object> *objects_p = var_p->type()->value()->run(log_p);

    // if failed...
    if(objects_p == NULL || log_p->is_errors())
        return(NULL);

    // add to variable list
    vars_p->add(var_p);

    int status;                         // processing status

    // do based on quantifier type
    switch(quantifier)
    {
        case Q_Forall:
            status = run_forall(vars_p, log_p, objects_p);
            break;

        case Q_Exists:
            status = run_exists(vars_p, log_p, objects_p);
            break;
    }

    vars_p->remove(vars_p->size() - 1);
```

```
    delete objects_p;

    // create the object in question
    Object *obj_p = NULL;
    if(!log_p->is_errors())
    {
        if(status)
        {
            obj_p = new Builtin_Object(
                  (Builtin_Class *)boolean_class_p->value(),
                                   true_value);
        }
        else
        {
            obj_p = new Builtin_Object(
                  (Builtin_Class *)boolean_class_p->value(),
                                   false_value);
        }
    }

    return(obj_p);
}


// get_quantifiers() gets all quantifiers of the symbol
void Quantified_Value::get_quantifiers(List<Symbol> *symbols_p)
{
    int found = 0;                          // is it found?
    int i;                                  // loop index

    // see if this type is already here
    for(i=0; i<symbols_p->size(); ++i)
    {
        // if this is the one...
        if((*symbols_p)[i] == var_p->type())
        {
            found = 1;
            break;
        }
    }

    // if not found...
    if(!found)
    {
        symbols_p->add(var_p->type());
    }

    body_p->get_quantifiers(symbols_p);
}


// run_forall() runs a FORALL value
int Quantified_Value::run_forall(List<Variable> *vars_p, Error_Log *log_p,
                List<Object> *objects_p)
{
    int status = 1;                // processing status
```

```
    int i;                          // loop index

    // do for all values of the object...
    for(i=0; status && i < objects_p->size(); ++i)
    {
        Object *obj_p;                  // value returned

        var_p->set_value((*objects_p)[i]);
        if((obj_p = body_p->generate_value(vars_p, log_p)) == NULL)
        {
            // fail to get value - return
            var_p->remove_value();
            return(0);
        }

        var_p->remove_value();

        status = obj_p->true();
        delete obj_p;
    }

    var_p->remove_value();

    return(status);
}


// run_exists() runs an EXISTS quantified value
int Quantified_Value::run_exists(List<Variable> *vars_p, Error_Log *log_p,
                        List<Object> *objects_p)
{
    int status = 0;                 // processing status
    int i;                          // loop index

    // do for all values of the object...
    for(i=0; !status && i < objects_p->size(); ++i)
    {
        Object *obj_p;                  // value returned

        var_p->set_value((*objects_p)[i]);
        if((obj_p = body_p->generate_value(vars_p, log_p)) == NULL)
        {
            // fail to get value - return
            var_p->remove_value();
            return(0);
        }

        var_p->remove_value();

        status = obj_p->true();
        delete obj_p;
    }

    var_p->remove_value();

    return(status);
}
```

```
// relation.cpp

// Contains code for Relation, which is a type of object that
// maps to a relation (e.g., a tuple).

#include <iostream.h>
#include <stddef.h>
#include <string.h>

#include "relation.h"
#include "object.h"
#include "list.h"
#include "variable.h"
#include "utils.h"
#include "errorlog.h"

// output() outputs a relation
void Relation::output(ostream &out, int indentation)
{
    int i;                  // loop index

    indent(out, indentation);

    out << "RELATION:\n";

    // do for all fields
    for(i=0; i<fields.size(); ++i)
    {
        // output the field
        indent(out, indentation + 3);
        out << *(fields[i]) << "\n";
    }

    out << "\n";
}



// clone() duplicates this
Object *Relation::clone()
{
    Relation *new_p = new Relation;         // the new thing
    int i;                                  // loop index

    // do for all fields
    for(i=0; i<fields.size(); ++i)
    {
        // add the field
        new_p->add_field(fields[i]->name(), fields[i]->value()->clone());
    }

    return(new_p);
}


// add_field() adds a field to a relation
void Relation::add_field(char *name, Object *value)
{
```

```
    fields.add(new Variable(name, value));
}


// true() tells that relation is always true
int Relation::true()
{
    return(1);
}


// run_message() runs a message against this
Object *Relation::run_message(char *message, List<Object> *parms_p,
                        Error_Log *log_p)
{
    // make sure there are no parameters - relation methods are arity 0
    if(parms_p->size() > 0)
    {
        log_p->log("There should be no parameters to relational message");
        return(NULL);
    }

    // find the field
    int i;                // loop index

    for(i=fields.size() - 1; i >= 0; --i)
    {
        if(strcmp(message, fields[i]->name()) == 0)
            return(fields[i]->value()->clone());
    }

    log_p->log("Relational field not found");

    return(NULL);
}
```

```
// reslist.cpp

// Contains code for Result_List, which is a list of query
// results.

#include "reslist.h"
#include "list.h"
#include "result.h"
#include "class.h"
#include "object.h"
#include "errorlog.h"


// add_class() adds a class to the result list
void Result_List::add_class(Class *class_p, Error_Log *log_p)
{
    // if no result for this already...
    if(find_result(class_p) == NULL)
    {
        results.add(new Result(class_p));
        class_p->get_bases(this, log_p);
    }
}



// extract() extracts objects for a specific class
List<Object> *Result_List::extract(Class *class_p)
{
    Result *result_p;                // the result in question

    // find the result
    if((result_p = find_result(class_p)) != NULL)
        return(result_p->extract());
    else
        return(NULL);
}



// run() runs the results until all are done
void Result_List::run(Error_Log *log_p)
{
    int done = 0;                // are we done?

    // do until done
    while(!done)
    {
        // mark as tentatively done
        done = 1;

        int i;                        // loop index

        // do for all results...
        for(i=0; i<results.size(); ++i)
        {
            // if this one not done...
            if(!results[i]->complete())
```

```
            {
                // run on this loop
                if(results[i]->get_class()->run_for_result(results[i],
                                                           this, log_p))
                    done = 0;
            }
        }

        // checkpoint all results
        for(i=0; i<results.size(); ++i)
            results[i]->end_cycle();

        // if failed on this loop, terminate
        if(log_p->is_errors())
            done = 1;
    }
}


// find_result() finds a specific result
Result *Result_List::find_result(Class *class_p)
{
    Result *result_p = NULL;        // the answer

    int i;                          // loop index

    // search for it
    for(i=0; i<results.size(); ++i)
    {
        if(results[i]->get_class() == class_p)
        {
            result_p = results[i];
            break;
        }
    }

    return(result_p);
}
```

```
// result.cpp

// Contains code for Result, which is a single result found as
// part of a query.

#include "class.h"
#include "list.h"
#include "object.h"
#include "result.h"


// Result() constructs a result
Result::Result(Class *class_p)
{
    result_class_p = class_p;
    is_complete = 0;
    new_object_offset = 0;
}

// extract() extracts the current object list
List<Object> *Result::extract(void)
{
    List<Object> *out_p = new List<Object>;        // results list

    out_p->merge(&objects);
    new_object_offset = 0;

    return(out_p);
}


// get_class() gets the class of the result
Class *Result::get_class()
{
    return(result_class_p);
}


// get_objects() gets the object list of this
List<Object> *Result::get_objects()
{
    return(&objects);
}


// complete() tells if result is completely filled in
int Result::complete()
{
    return(is_complete);
}


// end_cycle() checkpoints this
void Result::end_cycle()
{
    new_object_offset = objects.size();
    objects.merge(&pending_objects);
}
```

```
// add() adds an object to this list
void Result::add(Object *obj_p)
{
    pending_objects.add(obj_p);
}


// add() adds an entire list of objects
void Result::add(List<Object> *obj_list_p)
{
    pending_objects.merge(obj_list_p);
}


// mark_complete() notes this is complete
void Result::mark_complete()
{
    is_complete = 1;
}



// is_new() tests if an object is new (since the last checkpoint)
int Result::is_new(int index)
{
    return(index >= new_object_offset);
}
```

```
// roxanne.cpp

// Contains code for Roxanne, which is the application-level
// object for Roxanne.

#include <owl\applicat.h>
#include <owl\framewin.h>
#include <owl\dialog.h>
#include <dir.h>

#include "cllstwin.h"
#include "roxwin.h"

// Create the application
class Roxanne : public TApplication
{
    public:
        Roxanne(char *title) : TApplication(title) {}
        void InitMainWindow();
};

// Initialize the main window
void Roxanne::InitMainWindow()
{
    SetMainWindow(new TFrameWindow(0,
                        "Roxanne",
                        new Class_List_Window(NULL, 1),
                        TRUE));
}


// main function for this
int
OwlMain(int, char **)
{
    chdir("d:\\thesis\\roxanne\\data");
    return Roxanne("Roxanne").Run();
}
```

```
// statetbl.cpp

// Contains code for State_Table, which implements a finite
// state machine's state table.

#include <stddef.h>

#include "statetbl.h"

const int no_state = -1;                // no state found
const int no_type = -1;                 // no type found

static const int state_width = 128;     // count of transitions in state
static const int state_quantum = 100;   // number of states to allocate

// State_Table() constructs a state table
State_Table::State_Table()
{
    // allocate a bunch of states and types
    states_p = new int *[state_quantum];
    types_p = new int[state_quantum];

    int i;                  // loop index

    for(i=0; i<state_quantum; ++i)
    {
        states_p[i] = NULL;
        types_p[i] = no_type;
    }

    // set values as needed
    state_ptrs_allocated = state_quantum;
}


// ~State_Table() destructs a state table
State_Table::~State_Table()
{
    int i;                  // loop index

    for(i=0; i<state_ptrs_allocated; ++i)
        if(states_p[i] != NULL)
            delete states_p[i];

    delete states_p;
    delete types_p;
}


// add_transition() adds a new transition to another state
void State_Table::add_transition(int from, char on, int to)
{
    // allocate the state if necessary
    if(from >= state_ptrs_allocated || states_p[from] == NULL)
    {
        create_state(from);
    }
```

```
    // add the transition
    *(states_p[from] + (on & 0x7f)) = to;
}


// set_type() sets the type of a state
void State_Table::set_type(int state, int type)
{
    if(state >= state_ptrs_allocated)
        expand_states_p(state);

    types_p[state] = type;
}

// get_next_state() gets the next state given current state and char
int State_Table::get_next_state(int from, char on)
{
    int next = no_state;                // next state

    if(from < state_ptrs_allocated && states_p[from] != NULL)
        next = *(states_p[from] + on);

    return(next);
}


// get_type() gets type of a state
int State_Table::get_type(int state)
{
    if(state >= state_ptrs_allocated)
        return(no_type);
    else
        return(types_p[state]);
}



// expand_states_p() expands states to be at least a given size
void State_Table::expand_states_p(int minimum)
{
    int new_state_ptrs_allocated;       // new amount to allocate
    int **new_states_p;                   // new states
    int *new_types_p;                    // new types

    new_state_ptrs_allocated = state_ptrs_allocated;
    while(new_state_ptrs_allocated < minimum)
        new_state_ptrs_allocated += state_quantum;
    new_states_p = new int *[new_state_ptrs_allocated];
    new_types_p = new int [new_state_ptrs_allocated];

    int i;                          // loop index */

    // move in current states
    for(i=0; i<state_ptrs_allocated; ++i)
    {
        new_states_p[i] = states_p[i];
        new_types_p[i] = types_p[i];
```

```
    }

    // null fill new states
    for(i=state_ptrs_allocated; i<new_state_ptrs_allocated; ++i)
    {
        new_states_p[i] = NULL;
        new_types_p[i] = no_type;
    }

    // move new stuff into place
    delete states_p;
    delete types_p;
    states_p = new_states_p;
    types_p = new_types_p;
    state_ptrs_allocated = new_state_ptrs_allocated;
}



// create_state() creates a new state
void State_Table::create_state(int state)
{
    // expand states if needed
    if(state > state_ptrs_allocated)
        expand_states_p(state);

    // if must create the state...
    if(states_p[state] == NULL)
    {
        // allocate it
        states_p[state] = new int[state_width];

        // fill it in with no-state transitions
        int i;                  // loop index

        for(i=0; i<state_width; ++i)
            *(states_p[state] + i) = no_state;
    }
}
```

```
// symbol.cpp

// Contains code for Symbol, which is anything with a name
// (e.g., a class or variable).

#include <stddef.h>
#include <string.h>

#include "environ.h"
#include "symbol.h"
#include "class.h"


// Symbol() constructs this
Symbol::Symbol(char *in_name, Class *class_p)
{
    symbol_name = new char[strlen(in_name) + 1];
    strcpy(symbol_name, in_name);

    value_p = class_p;
}


// ~Symbol is the destructor of this
Symbol::~Symbol(void)
{
    if(symbol_name != NULL)
        delete[] symbol_name;

    if(value_p != NULL)
        delete value_p;
}


// set_value() sets the value of the symbol
void Symbol::set_value(Class *class_p)
{
    if(value_p != NULL)
        delete value_p;

    value_p = class_p;
}

// value() gets the value of this symbol
Class *Symbol::value(void)
{
    return(value_p);
}


// clear_value() clears out the value of the symbol
void Symbol::clear_value()
{
    set_value(NULL);
}


// name() gets the name of the symbol
```

```
char *Symbol::name()
{
    return(symbol_name);
}
```

```
// utils.cpp

// Contains various utility functions.


#include <iostream.h>

#include "utils.h"

// indent() indents n spaces
void indent(ostream &out, int n)
{
    // do for all the spaces...
    for(/* no init */; n > 0; --n)
        out << ' ';
}
```

```
// value.cpp

// Contains the code for Value, which is a virtual class that
// maps to anything that can serve as a value.

#include <stddef.h>

#include "value.h"
#include "object.h"
#include "list.h"
#include "variable.h"
#include "errorlog.h"
#include "symbol.h"
#include "classpec.h"
#include "environ.h"
#include "quanval.h"
#include "constval.h"
#include "varval.h"
#include "cplxval.h"



// ~Value() is the destructor for this
Value::~Value()
{
}



// get_quantifiers() gets quantifiers for this
void Value::get_quantifiers(List<Symbol> *)
{
    // This is default for virtual function: default does nothing
}



// read_value() is related: it reads in any type of value
Value *read_value(Class_Spec *in_p, Environment *env_p)
{
    Value *val_p;                // the value

    // do based on next token
    switch(in_p->next_token())
    {
        case Open_Paren:
            // get a (<VALUE>)
            in_p->skip_token();
            if((val_p = read_value(in_p, env_p)) == NULL)
                return(NULL);
            if(in_p->next_token() != Close_Paren)
                in_p->log_error("Missing close parenthesis");
            else
                in_p->skip_token();
            break;

        case String:
        case Integer:
```

```
        case True_Label:
        case False_Label:
            val_p = new Constant_Value(in_p, env_p);
            break;

        case Name:
            val_p = new Variable_Value(in_p, env_p);
            break;

        case Forall_Label:
        case Exists_Label:
            val_p = new Quantified_Value(in_p, env_p);
            break;

        case Not_Label:
            val_p = new Complex_Value(NULL, in_p, env_p);
            break;

        default:
            in_p->log_error("Invalid value");
            val_p = NULL;
            break;
    }

    // check value
    if(val_p == NULL || in_p->is_errors())
    {
        if(val_p != NULL)
            delete val_p;
        val_p = NULL;
    }
    else
    {
        while(in_p->next_token() == Period
                || in_p->next_token() == Plus
                || in_p->next_token() == Minus
                || in_p->next_token() == Equals
                || in_p->next_token() == Slash
                || in_p->next_token() == Asterisk
                || in_p->next_token() == Less_Than
                || in_p->next_token() == Greater_Than
                || in_p->next_token() == Not_Equals
                || in_p->next_token() == Less_or_Equals
                || in_p->next_token() == Greater_or_Equals
                || in_p->next_token() == And_Label
                || in_p->next_token() == Or_Label)
        {
            val_p = new Complex_Value(val_p, in_p, env_p);
        }
    }

    return(val_p);
}
```

```
// variable.cpp

// Contains the code for Variable, which stores a variable,
// its name, its type, and its value (if any is yet assigned).

#include <iostream.h>
#include <string.h>

#include "variable.h"
#include "environ.h"
#include "object.h"
#include "symbol.h"
#include "classpec.h"
#include "utils.h"


// Variable() constructs this from its pieces
Variable::Variable(char *name, Object *value_p, Symbol *in_type_p)
{
    // store the name
    if(name != NULL)
    {
        var_name = new char[strlen(name) + 1];
        strcpy(var_name, name);
    }

    // store the value
    var_value_p = value_p;

    // store the type
    var_type_p = in_type_p;
}


// Variable() constructs this from input
Variable::Variable(Class_Spec *in, Environment *env_p)
{
    // initialize everything to NULL
    var_type_p = NULL;
    var_value_p = NULL;
    var_name = NULL;

    // get type
    if(in->next_token() != Name)
    {
        in->log_error("Missing type of variable");
    }
    if((var_type_p = env_p->find_symbol(in->get_value())) == NULL)
    {
        in->log_error("Unknown variable type");
    }
    in->skip_token();

    // get name
    if(in->next_token() != Name)
    {
        in->log_error("Missing name of variable");
    }
```

```
    var_name = new char[strlen(in->get_value()) + 1];
    strcpy(var_name, in->get_value());
    in->skip_token();
}


// ~Variable() destroys this
Variable::~Variable()
{
    if(var_name != NULL)
        delete[] var_name;
    if(var_value_p != NULL)
        delete var_value_p;
}


// set_value() sets the value of this
void Variable::set_value(Object *new_value_p)
{
    if(var_value_p != NULL)
        delete var_value_p;

    var_value_p = new_value_p;
}


// clear_value() clears a value
void Variable::clear_value()
{
    set_value(NULL);
}


// remove_value() clears a value without destructing it
void Variable::remove_value()
{
    var_value_p = NULL;
}


// type() gets type of the variable
Symbol *Variable::type()
{
    return(var_type_p);
}


// name() gets name of this
char *Variable::name()
{
    return(var_name);
}
```

```
// value() gets value of this
Object *Variable::value()
{
    return(var_value_p);
}


// clone() duplicates this
Variable *Variable::clone()
{
    Object *new_value_p = NULL;                 // the new object value

    if(var_value_p != NULL)
        new_value_p = var_value_p->clone();

    return(new Variable(var_name, new_value_p, var_type_p));
}



// output() outputs a variable
void Variable::output(ostream &out, int indentation)
{
    indent(out, indentation);
    out << name() << " = ";
    if(value() == NULL)
        out << "<UNDEFINED>";
    else
    {
        value()->output(out, indentation + 2);
    }
}


// operator<< outputs this
ostream &operator<< (ostream &out, Variable &var)
{
    var.output(out);

    return(out);
}
```

```
// varval.cpp


// Contains the code for Variable_Value, which is a type of
// value that references a variable.


#include <string.h>
#include <stddef.h>
#include <stdio.h>

#include "varval.h"
#include "classpec.h"
#include "environ.h"
#include "list.h"
#include "errorlog.h"

// Variable_Value() is the constructor
Variable_Value::Variable_Value(Class_Spec *in_p, Environment *)
{
    // initialize variable name
    variable_name = NULL;

    // make sure this is a name
    if(in_p->next_token() != Name)
    {
        in_p->log_error("Missing variable name");
    }
    else
    {
        variable_name = new char[strlen(in_p->get_value()) + 1];

        strcpy(variable_name, in_p->get_value());
        in_p->skip_token();
    }
}


// ~Variable_Value is the destructor
Variable_Value::~Variable_Value()
{
    if(variable_name != NULL)
        delete[] variable_name;
}



// generate_value() gets the value of this
Object *Variable_Value::generate_value(List<Variable> *vars_p,
                                       Error_Log *log_p)
{
    int i;                          // loop index

    // find the variable in question
    for(i=0; i<vars_p->size(); ++i)
    {
        // if this is the desired variable
        if(strcmp(variable_name, (*vars_p)[i]->name()) == 0)
```

```
        {
            if((*vars_p)[i]->value() == NULL)
                return NULL;
            else
                    return((*vars_p)[i]->value()->clone());
        }
    }

    // log an error - variable unfound
    char error[100];                // to hold error string

    sprintf(error, "Unknown variable %s", variable_name);
    log_p->log(error);

    return(NULL);
}
```