MANAGING SCIENTIFIC COMPUTATIONS WITH A JAVA-BASED SCHEMA EXTENSION FACILITY

Name: Samuel Joseph Eaves, II
Department: Department of Computer Science
Major Professor: Gregory A. Riccardi
Degree: Doctor of Philosophy
Term Degree Awarded: Spring, 2000

Scientists using several computer programs to produce and/or analyze computational data face several challenges. They must organize the invocation of the programs. Also, they must keep track of the voluminous data being generated and/or analyzed. In addition, they must record the steps involved in the computational process for later examination.

We have produced a Java-based Schema Extension Facility (JSEF) as an aid to scientists involved in the process of computational discovery. We use Java's object-oriented data model as a basis for classifying computational entities. Also, JSEF's relationship mechanism allows us to create relationships between classes. In fact, JSEF allows a series of associated programs to be classified as a single entity called an analysis strategy.

In the computational process, each entity is referenced by a JSEF object. In order to drive the computation, a method is called on a JSEF object, which invokes the necessary processing. JSEF objects are stored in a database upon creation.

Thus, JSEF reduces the tasks of invoking and tracking a computation and the entities involved to a single method call and subsequent querying of the database and examining the state of the JSEF objects in the query result.

THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

MANAGING SCIENTIFIC COMPUTATIONS WITH A JAVA-BASED SCHEMA EXTENSION FACILITY

By

SAMUEL JOSEPH EAVES, II

A Dissertation submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Spring Semester, 2000

The members of the Committee approve the dissertation of Samuel Joseph Eaves, II defended on MARCH 13, 2000.

Gregory A. Riccardi
Professor Directing Thesis

Lawrence C. Dennis
Outside Committee Member

Theodore P. Baker
Committee Member

Kyle A. Gallivan
Committee Member

Lois W. Hawkes
Committee Member

Bryon K. Ehlmann
Committee Member

Approved:

Theodore P. Baker, Chair
Department of Computer Science

To my progeny yet unborn. I hope I can be an inspiration to you as my forefathers have been to me.

ACKNOWLEDGEMENTS

Praised be Thy name forever, O my King, my God and King; great and holy, in heaven and on earth. Unto Thee I offer blessings and thanksgiving from this time forth and forever more. Blessed art Thou, O Lord, exalted in praises, God of thanksgiving, Lord of wonders, Who has sustained me though this long process; my God and King, the life of the universe ...

I would like to thank Dr. Riccardi for guiding me through the research process. You have been concerned not only with my scholastic achievement, but with my well-being as a person. I really appreciate what you have done for me.

I must thank my family for their support and confidence in me throughout my life, and during this process in particular. At times, I did not know who wanted it more: them or me!

I must thank my second family, the McKnight Doctoral Fellows. While Dr. Israel Tribble and Dr. Bettye Parker-Smith offered advice and encouragement, we McKnights shared our good times and our "war stories" with each other. I am proud to be counted amoung you all.

Last, but not least, I would like to thank Cherié for her love, support and encouragement. She is perhaps the only one who really knows what challenges I have faced. She has been a stabilizing force through the tough times.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Scientists using several computer programs to produce and/or analyze computational data face several challenges. They must organize the invocation of the programs. Also, they must keep track of the voluminous data being generated and/or analyzed. In addition, they must record the steps involved in the computational process for later examination.

We have produced a Java-based Schema Extension Facility (JSEF) as an aid to scientists involved in the process of computational discovery. We use Java's object-oriented data model as a basis for classifying computational entities. Also, JSEF's relationship mechanism allows us to create relationships between classes. In fact, JSEF allows a series of associated programs to be classified as a single entity called an analysis strategy.

In the computational process, each entity is referenced by a JSEF object. In order to drive the computation, a method is called on a JSEF object, which invokes the necessary processing. JSEF objects are stored in a database upon creation. Thus, JSEF reduces the tasks of invoking and tracking a computation and the entities involved to a single method call and subsequent querying of the database and examining the state of the JSEF objects in the query result.

CHAPTER 1

INTRODUCTION

This dissertation research has been carried out to develop an application that manages the process of computational discovery. It will be used to support the management of very large scientific computations at the Thomas Jefferson National Accelerator Facility (TJNAF), a U. S. Department of Energy nuclear physics laboratory in Newport News, Virginia. Experiments at the lab collect one terabyte of raw data per day for 200 days per year. More than 400 hundred scientists around the world perform individual and shared analysis of this data.

In general, the process of *computational discovery* involves a scientist's use of computer programs to create and/or analyze datasets that are relevant to his/her research. Typically, computational discovery involves the production of massive amounts of data and the performance of many computations. Often, the data and the computations must be distributed among multiple computer systems.

Computational scientists face several challenges when working with scientific data. They must develop a data model which defines and identifies entities of interest and relationships between these entities. This model must be able to evolve, so as to remain current with evolutions in scientific models and theories. Scientists also need a mechanism to create and maintain metadata that adequately describes scientific data and the processing that occurs to create and/or analyze it. Reliable metadata, along with a persistent management facility will facilitate data reusability and sharing.

We have produced a general-purpose schema extension facility that scientists can use to define and manage computational analyses and scientific data. This facility is based on the Java programming language which provides us with several notable elements, including an object-oriented data model, functionality to manage database access with JDBC (see section 2.5.4), and compliance with industry standards (see section 2.2).

Using Java's object-oriented data model, scientists are able to define and extend classes to represent scientific data and computations. Objects instantiated from these classes serve as metadata for actual resources. These objects persist in a database, where they can be kept indefinitely, shared, and queried. Additionally, these objects are imparted with a certain degree of interoperability (as a result of Java's compliance with industry standards) which fosters resource sharing.

1.1 Related Work

Scientific data resulting from an experiment or simulation has been characterized as being volumous, infrequently updated, and stored indefinitely [Fre91]. Because of these traits, traditional database techniques are inadequate for managing large amounts of scientific data effectively [HQGW93]. Thus, scientific applications have relied on scientific databases to manage collections of scientific data. This includes maintaining metadata which identifies raw data and metadata which describes how raw data was derived (i.e. information that describes scientific experiments and/or simulations).

The services offered by database systems, such as persistence, concurrency, and recovery, are useful for scientific applications [RD94]. However, there is little similarity among data from different scientific areas. Different fields of scientific study may require different data models, query facilities, and storage techniques. Thus, federated database management systems (FDBMS) have been used to integrate autonomous and heterogenous data repositories (and other software systems). FDBMS provide a framework by which scientists can access and integrate various types of data and information from legacy systems into one federation [C^+97].

Maier and Hansen [MH94] maintain that it advantageous to use an object-based data model to express scientific data. Such a model is not constrained to a set of pre-defined data structures. Instead, researchers can create data types to directly model entities of interest. This extensibility allows researchers to maintain current data models, even as scientific models and theories evolve. Further, Maier and Hansen state that an object-based model can be implemented in an object-oriented database (OODB), a relational database, or a simple file paradigm. In fact, they found that an object-based model is an effective way to impose some management on scientific data which usually resides in files unmanaged by any system.

Gupta et al. [GHR97] also offer Virtual Database (VDB) technology as a mechanism to organize data scattered across various media. Organizing such data is vital, since as much at 90% of the world's data is not contained within relational database systems. One method of integrating this data is to use object wrappers. Chen et al. also discuss object wrappers as a component of their Object-Protocol Model [CKMS97].

Due to their flexibility, object models appear well-suited for capturing metadata. Metadata helps scientists ensure that they will properly interpret their data in the future. Metadata also facilitates data sharing, ensuring that others will interpret the data properly also.

1.2 Problems Addressed

Computationally-intensive scientific experiments often produce large amounts of data. To perform such experiments, a researcher may use several analysis programs and existing data over the course of an investigation to produce new data for future examination. However, there are several problems researchers must overcome in order to examine this data.

For instance, the output data from two given programs often suffer from a lack of interoperability [CMFD94]. This problem results when there are differences in the physical or logical structure of the data, or when there are platform incompatibilities between analysis programs and data files which reside on several different computers.

In addition to the possibility that data may be stored across multiple machines, the data may not be contained within a database. Thus, scientists trying to locate and obtain loosely-organized, heterogeneous scientific data face the *data scatter* problem [GHR97].

Also, according to Williams [Wil98], scientific data is generally useless without some descriptive mechanism (e.g. metadata) supporting it. Unfortunately, French [Fre91] suggests that scientists believe that most databases lack sufficient metadata. Consequently, there is a need for the development of a metadata infrastructure to adequately describe data, an infrastructure that is easily maintained, is capable of evolving as needed, and can provide “information required to identify data sets of interest, their content, validity, and sources” [Fre91, page 61]. Sufficient metadata is key in helping researchers understand data that is shared.

Generally speaking, scientists must schedule the execution of the programs they use. They must also keep track of the voluminous data being generated and/or analyzed. In addition, they must record the steps involved in and the results of the computational process for later examination. Each of these responsibilites becomes more difficult as the number of programs, datasets, computations, number of scientists involved, or the length of time the computation has been utilized increases.

Nevertheless, Karin and Graham [KG98] point out that the trend in high-performance computing, which includes scientific computing, has been to develop systems that compute faster, systems that are distributed across many sites and used by many researchers. In fact, “[s]upercomputers, large scientific instruments..., and multiterabyte scientific databases, data archives, and digital libraries are... becoming ubiquitous, continuous, and pervasive,” [KG98, p. 33]. Even in engineering, Hass [Has99] notes that although the increasing reliance on computation is not always embraced in industry, it is an irreversible trend.

1.3 Research Focus

This dissertation research has produced a general-purpose schema extension facility (i.e. one that is not tailored to any specific domain) and applied it to an actual computational experiment. This facility provides a framework by which computational entities can be described and managed. The Java programming language forms the basis of an object-oriented data model in which researchers can define classes to represent such resources as scientific data and data analysis programs. Objects instantiated from these classes serve as metadata for actual computational resources. In addition, JSEF objects can have relationships between them. A set of related JSEF objects can be used to describe an entire computation. Also, JSEF objects persist in a database where they can be kept indefinitely, shared, and queried.

In JSEF, new object types can be defined as needed, allowing a schema to evolve as necessary. In addition, object typing in JSEF as a result of inheritance allows JSEF data type constraints on relationships to maintain the integrity of computations by preventing incompatible objects from being related. Further, object types can be used as the basis of an OQL search in which sets of objects can be returned, new objects (e.g. sets) can be created, and methods can be executed on existing objects.

We have identified three classes of users of this facility. *Schema designers* are expert users of this facility who are responsible for developing a broad schema for classes of computational analyses and data sets. Next, *schema refiners* are users responsible for tailoring a broad schema to specific analysis programs and data sets. Finally, *data analyzers* are those users who instantiate and manipulate objects in order to perform a computation (e.g. analyzing a series of datasets). The tasks performed by these users are described in Section 4.2.

JSEF objects describe and manage computational actions without conflicting with techniques that optimize the execution environment itself. The instantiation of a JSEF object does not trigger the respective computational entity to begin its computational action. Instead, JSEF uses method calls to drive the computational process. Thus, a data analyzer may instantiate an arbitrarily large number of objects that describe several instances of the complex computation. When the data analyzer is ready to invoke the computation, he/she simply makes the appropriate method call(s).

1.4 Summary

This dissertation addresses the needs of researchers who perform scientific data computation and analysis. These needs include an expandable data model, sufficient metadata to describe scientific data and analyses, and computation management.

Thus, we are proposing a schema extension facility to manage scientific computations and data analysis. This facility will provide a general-purpose, Java-based object-oriented data model in which researchers will be able to design classes that model entities of interest and relationships between entities. Researchers will be able to evolve these classes as needed. In addition, researchers will be permitted to instantiate classes in order to create objects that will serve as metadata for actual scientific data sets and analyses. Finally, objects will be stored in a database where they can be stored indefinitely, shared, and queried.

The remainder of this dissertation is organized as follows. Chapter 2 discusses scientific databases, object oriented paradigm issues, computation management models, and metadata. Chapter 3 presents the Java-based schema extention facility (JSEF). Chapter 4 examines the application of JSEF to an example paradigm. Chapter 5 describes the impact of this research to the field.

CHAPTER 2

BACKGROUND

In this chapter, we discuss several areas of research and development which are pertinent to the development of a general-purpose Java-based schema extension facility. Among them are the object-oriented paradigm and object standards, existing models of schema development, distributed object computing, the Java programming language, and metadata.

2.1 The Object-Oriented Paradigm

According to Khoshafian and Abnous [KA90, p. 6], the object-oriented paradigm is appealing because it "provides better concepts and tools with which to model and represent the real world as closely as possible." Three concepts are fundamental to the object-oriented paradigm. *Abstract data typing* allows designers to encapsulate a user-defined data structure with operations that manipulate instances of that data structure. This encapsulated unit defines a *class*. *Inheritance* allows designers to create new classes based on existing ones by inheriting the structures and functions of the existing (base) classes. Khoshafian and Abnous regard this as the basis of "a powerful modeling ... strategy" [KA90, p. 9]. Finally, *object identity* is the distinguishing quality that makes two objects distinct, even if they are instances of the same class. This allows objects to refer to or contain other objects.

From a design point of view, the object-oriented paradigm has helped reduce the effort required to transform a system's requirements into a system specification. Designers can express problems more easily and naturally. These object-oriented techniques, designers can construct complex systems using reusable software modules. This promotes easier code maintenance and code reuse.

2.2 Object Standards

2.2.1 Object Management Architecture

The Object Management Group (OMG) is an international trade association formed in 1989 "... to help reduce the complexity, lower costs, and hasten the introduction of new software applications" [SS95, p. 2]. By doing so, the OMG hopes to move the industry toward portable, reusable, interoperable object-oriented software components. To achieve this goal, the OMG has set forth an Object Management Architecture (OMA) with detailed interface specifications supporting it. The major components of the OMA are described as follows.

Object Request Broker (ORB). The ORB is the component that manages communications within the OMA. Specifically, it provides an infrastructure that allows objects to communicate, regardless of specific platforms and addressing techniques. Thus, the ORB guarantees portability and interoperability of objects over a heterogeneous network (e.g. the internet). It is often referred to as the Common Object Request Broker Architecture (CORBA)[1] [OMG95].

Object Services. This component standardizes the management of an object's life cycle. In particular, it provides functionality that allows an object to be created and accessed. There is also functionality that tracks an object's location and maintains relationships between objects.

Common Facilities. This component provides a basic set of functions that are potentially common across applications and platforms . These functions can

[1] There is a slight difference between the ORB and CORBA definitions.

be tailored to the requirements of a specific configuration. Some examples include printing or electronic mail services.

Application Objects. This refers to those modules that are specific to a given application. These, combined with the Common Facilities, form a complete application system.

Core Object Model. This model is a set of commonly agreed upon object-oriented concepts (e.g. object, inheritance, etc.) that formally specifies features which all OMA-compliant object systems should support. It also allows for new components to extend this model. Each set of extensions forms a *profile*.

Strictly speaking, although ORB and CORBA are referred to synonymously (as mentioned above), CORBA is actually a profile of the OMA. Specifically, CORBA uses the term *interface* to express the notion of *type*. Also, CORBA has a more strict inheritance rule than that of the Core Object Model. While CORBA discusses the use of inheritance, it does not address the notion of *substitutability*. Finally, CORBA defines exceptions, which are not a part of the Core Object Model.

Reference Model. The reference model is intended to influence the high-level architectural and component designs of a proposed system by pointing out areas that need to be addressed in order for the design to be considered OMA compliant. Some of these considerations are objects that conform to the OMG Object Model, object location transparence, object naming, and object request performance.

The OMA is important because it addresses the current lack of interoperability among heterogeneous systems. Specifically, it provides a basis for the development of distributed, platform-independent applications and the exchange of data between machines in a distributed environment. In fact, Siegel [Sie98] points out several features of the OMA and CORBA which make it appealing to amost any computing enterprise where heterogeneity and the integration of legacy software are important considerations.

2.2.2 Object Database Management Group

The Object Database Management Group (ODMG) first met in the fall of 1991 to address the lack of progress toward Object Database Management Systems (ODBMSs) standards [CB97]. In March 1997, the ODMG released version 2.0 of its object database standard for object database management systems. This standard, referred to as ODMG 2.0, reflects the group's continuing effort to develop a standard for creating persistent, object-oriented programming language objects in databases. It applies to both Object Database Management Systems (ODBMSs) that store objects directly, and Object-to-Database Mappings (ODMs) that convert and store objects in a relational (or other alternative) representation.

The ODMG is striving to raise the level of integration between programming languages and databases. In fact, the ODMG hopes that the standard will foster the development of applications in which portability applies to entire applications and not just the portions where database access takes place. Ultimately, the group hopes that ODMG 2.0 will help promote interoperability between entire applications (e.g. distributed applications communicating via the OMG's ORB). Below, we describe the major components of ODMG 2.0.

Object Model. The ODMG Object Model is designed as a profile for the OMG architecture, adding components such as relationships and exceptions. The Object Model is important because it defines the entities supported by an ODBMS, and it specifies the kinds of semantics that can be defined explicitely to an an ODBMS. Among other things, the Object Model semantics determine the characteristics of objects, how one can create relationships between objects, and how one can name and identify objects.

An application developer would use the constructs of the ODMG Object Model when constructing an application's object model. The application's object model would then serve as the logical schema for some database.

Object Specification Languages. Object Specification Languages provide a means by which developers can define the schema, operations, and state of an object database so that the database will be portable across ODMG-compliant implementations.

The Object Definition Language (ODL) is a specification language used to express object type specifications that conform to the ODMG Object Model. While not a full programming language, using ODL makes the portability of database schemas across conforming ODBMSs possible. Additionally, an application designer can directly represent an ODL specification in and object-oriented programming language such as Java, C++, or Smalltalk.

The Object Interchange Format (OIF) is a specification language that provides constructs that allow for the dumping and loading of the current state of an object database to or from a file or set of files. OIF is useful for exchanging objects between databases, populating databases, providing database documentation, and testing databases.

Object Query Language. ODMG's Object Query Language (OQL) is a simple declarative query language which supports the ODMG data model, providing easy access to an ODBMS. It extends SQL–92 [CO92], adding object-oriented notions such as complex objects, object identity, polymorphism, and method invocation. Programmers can invoke OQL within an ODMG-bound programming language, and conversely, OQL can invoke methods programmed in that language.

Language Bindings. ODMG 2.0 defines language bindings for Java, C++, and SmallTalk. They describe constructs within the language that application designers can use to define ODMG-compatible objects, invoke OQL, and manage database transactions. Thus, the binding of ODMG to specific languages provides a seamless interface for programming and data manipulation.

ODMG 2.0 is valuable because it addresses persistence and interoperability, two issues which are critical to scientific data management. Specifically, researchers often need to keep scientific data indefinitely. Additionally, scientists working in a heterogeneous networking environment (e.g. the internet) often need or want to share resources.

2.3 Managing Scientific Information and Computation

Scientists in multiple disciplines (e.g. biologists and physicists) have collaborated with computer scientists to develop software that performs computations on their data. In addition, strategies have been developed to help scientists manage the components and results of a computation. We describe some of these strategies below.

2.3.1 ZOO System

Researchers at the University of Wisconsin have developed a Desktop Experiment Management Environment (DEME) called Zoo [ILGP96]. Zoo is a generic tool that can be enhanced to become a complete Customized Desktop Experiment Management System (CDEMS) using *agents* which are designed specifically to interact with and monitor external applications.

At the core of the Zoo system is its database server, *Horse* (Heavy-duty Object Repository for Scientific Experiments). The Horse server is based on the *Moose* (Modeling Objects Of Scientific Experiments) object-oriented data model and the *Fox* (Finding Objects of eXperiments) declarative query language [WI93]. The other Zoo modules include the following:

Opossum *and* Squid These two components make up the user interface. Opossum (Obtaining Presentations Of Semantic Schemas Using Metaphors) is a schema manager and Squid (System for Queries Updates Insertions Deletions) is a query/update manager.

Emu This component (Experimentation Management Unit) translates user requests to actions at external systems linked to Zoo by way of agents. Emu is responsible for making preparations for the actions when necessary.

Turtle This component (Translation Unit of Run Time of Large Experiments) translates Moose objects into ASCII files and ASCII files into Moose objects based on a mapping file.

Frog Frog (Files Related to Objects Graphically) is a visual tool used to generate the map files required by Turtle.

The Zoo system is pertinent because it provides experiment management that are natural and intuitive for the non-expert user, yet powerful enough to capture the complexity of the experiments. Specifically, Zoo was designed to be an integrated software package with a single user interface that supports all stages of an experimental study, manages all data generated by the study transparently, and hides the underlying software details. In fact, [AIL98] asserts that interrelated, persistent objects can be used to manage scientific work flow.

We have used the Zoo system as a basis for what we have developed. In particular, the JSEF system is also an object-oriented computation management system that uses persistent objects to manage scientific work flow. However, our approach differs from the Zoo system in the manner in which we utilize the object-oriented paradigm. Specifically, while Zoo uses triggers to drive a computation, JSEF uses method calls. Also, JSEF uses inheritance to allow relatively easy creation of new schemas based on an existing schema.

2.3.2 geoPOM System

Researchers at the UCLA Data Mining Laboratory have developed a geoscientific object system called geoPOM (geoscientific Persistent Object Manager) which concentrates on the integration of spatial data repositories [NMM97]. It was developed for use with the OASIS system [MMS$^+$96] to implement persistentent geoscientific CORBA objects in a data repository-independent manner.

GeoPOM facilitates access to data in repositories while the repositories remain autonomous and the data remains in the repositories. GeoPOM provides an object-oriented data model (based on ODMG 1.2 data model) by which users can create their own types in addition to the pre-defined object types relevant to geoscience that are already included. GeoPOM then maps object types to data in spatial data repositories, "wrapping" legacy data with geoPOM types. There is also an object-oriented query language with which users can query geoPOM objects.

Although geoPOM is targeted toward spatial data, it is notable because it endeavors to address the need of a distributed, heterogeneous data management system that provides uniform access to integrated, non-traditional (e.g. (geo)scientific) data. It also proposes a portable, object-oriented data model based on an industry standard. As [NMM97] point out, there are many independent databases and file systems that contain useful (legacy) data that are incompatible because of heterogeneous data models.

2.3.3 Computational Proxies

The computational proxy mechanism is an object-oriented database structure proposed to address the problems of interoperability and data management for computational science applications [CMFD94]. Computational proxies model applications and invocations of applications, encapsulating them as objects. A computational proxy, along with an experiment database, provides an infrastructure for managing computational experiments.

The proxy allows the user to start up and control ongoing computational processes. In addition, it maintains important information about the execution status of a given experiment. When an experiment is invoked, the proxy sets up input for an application, launches the application, monitors its progress, and notes its termination. Once completed, the proxy transforms the results into a common format and stores them in the database. While the experimental results persist, the proxy itself exists only for the duration of an experiment (unless it is archived).

Computational proxies were not developed to create computational schemas. However, the computational proxy mechanism is relevant because it manages the execution of the experiements themselves. Specifically, proxies provide an interface which links heterogeneous applications and data into a uniform environment. As a result, it contributes to the preservation of the scientific community's investment in legacy systems. In addition, it supports functionality that allows users to manage computational experiments and make pertinent information (e.g. experiment output data) persistent.

2.3.4 Object-Protocol Model

The Object-Protocol Model (OPM) is an object data model with constructs common to other prominent object models (e.g. ODMG) [CM95]. OPM was developed primarily to assist scientists developing scientific database schemas and applications. In particular, OPM allows scientists to construct OPM *views* that specify the structure of their database applications in terms of their own frame of reference (i.e. with objects and protocols), thus allowing these applications to become easier to use, manage, understand, and update. Additionally, OPM provides functionality for querying and manipulating scientific databases.

The OPM retrofitting tool discussed in [CKMS97] allows users to construct and maintain OPM views for scientific databases that were not originally developed with OPM tools. This approach includes notion of wrapping[2] scientific database systems stored in files or in relational databases with OPM objects. This allows scientists to manipulate these resources using the functionality provided by an OPM interface.

The OPM and its associated toolkit (including the retrofitting tool) are noteworthy because they support the development of object-oriented views of the life cycle of scientific experiments. These views allow scientists to specify, query, and manipulate scientific databases from their own point of reference. In addition, they outline a method that integrates data stored in flat files and relational databases into a single, object-oriented view, forming an OPM-based database federation.

2.4 Distributed Object Computing

Scientists have conducted research to address the challenges of and take advantage of the benefits of doing work across distributed systems. These benefits include increased computational power, more room for data storage, and greater access to data and computations.

Metasystems create the illusion of a giant desktop computational environment [GFLH98]. Metasystems are built from geographically distrubuted resources and linked by high-speed interconnections and a layer of software. A metasystem is a virtual machine that allows users to execute applications transparently, detects and handles faults, provides security to users and resource owners, and fosters collab-

[2]Wrappers are also mentioned in [GHR97].

oration and sharing. In a metasystem environment, multi-component applications whose components might normally execute as stand-alone programs can be merged into *meta-applications*.

2.4.1 Legion Metasystem

Scientists at the University of Virginia have developed the *Legion* metasystem [GW+97, GFH99]. Legion can be thought of as a wide-area operating system that allows multiple organizations with diverse computing platforms to share and combine their resources. It is based on an object-oriented core framework where all components of interest are objects and all objects are instances of classes. While committed to the object-oriented paradigm, Legion is capable of incorporating non-object-based legacy source codes.

Since they cannot satisfy the needs of all users, the designers of Legion have incorporated enough flexibility to let users make their own tradeoffs between the kind and level of functionality of some needed or desired service (e.g. security and distributed file systems). This flexibility extends to the core components. Legion specifies the functionality and not the implementation of these core components. Thus, users can extend or even replace core components.

2.4.2 Globus Project

Scientists working on the *Globus* project are also building on the powerful capabilities of metacomputing environments. Globus is a toolkit which provides the basic infrastructure for building portable, high-performance implementations of a wide range of metacomputing services [FK97]. Globus modules that provide services include the Nexus communication library, the Metacomputing Directory Service (MDS), the Generic Security System (GSS), the Remote I/O (RIO) interface for data access, and the Globus Heartbeat Monitor for fault detection. Ultimately, developers hope that Globus will be the basis for an Adaptive Wide Area Resource

Environment (AWARE), a set of higher-level services that will empower applications to adapt to heterogeneous, dynamic metacomputing environments.

2.4.3 Java Analysis Studio

The Java Analysis Studio (JAS) [Joh98] is a data analysis application for the desktop primarily focused on handling the analysis of high-energy physics data. JAS features a rich graphical user interface that makes the system easy to use and customize. Additionally, JAS provides several built-in Data Interface Modules (DIMs) and allows users to create their own DIMs so that it will be independent of any particular data format. Further, JAS allows scientists to perform arbitrarily complex data analysis tasks using the built-in set of analysis modules or by creating their own. These tasks are created by linking together one or more of the DIMs.

JAS operates as a stand-alone application for local data analysis or as a client to a remote Java Data Server to analyze data at a remote location. In particular, when operating as a client, JAS helps overcome the bandwidth problems associated with moving large amounts of data across a network by sending the analysis to the data and returning the results to the user.

2.4.4 Beowulf Project

The Beowulf Project [BSS+95, RBMS97] grew out of the success of the first Beowulf machine created by Sterling and Becker at NASA Goddard Space Flight Center. Beowulf was the result of their idea to provide commodity off the shelf (COTS) base systems to satisfy specific high performance computational requirements. In a Beowulf cluster, distributed Linux-based COTS systems are linked by an Ethernet network. Operating system parameters are tuned and drivers are re-written in order to achieve high performance.

2.5 The Java Programming Language

Java is an object-oriented programming language introduced by Sun Microsystems, Inc. in June of 1995 [GJS96, Web97]. Sun has described it as being "...simple, ...robust, secure, portable, high-performance, architecturally neutral, multi-threaded, [and] dynamic" [Web97, p. 7]. Below, we discuss some of the highlights of Java.

2.5.1 Unique Java Constructs

The Java programming language includes several noteworthy conventions that were designed to prevent complications. First, Java allows users to define completely abstact classes called *interfaces*. Although Java only allows single inheritance, a class may implement (i.e. specify the behavior of) multiple interfaces. The use of interfaces, coupled with Java's single inheritance restriction helps eliminate possible complications of multiple inheritance. Java also includes the concept of *references*. References are similar to pointers, but they eliminate common problems associated with pointers (e.g. memory leaks). Also, references do not allow users to perform pointer arithmetic or have much control over memory allocation and deallocation (although users can invoke limited memory allocation methods).

Java can be considered a completely object-oriented language. All Java classes inherit from the class `java.lang.Object`. This class ensures that at some level, all classes have a common ancestor. In addition, all variable declarations and method definitions must be contained within a class. Even the *main* method must be contained in a class. Furthermore, while Java defines primitive types (e.g. int, char, float, etc.), Java also provides wrapper classes which provide object representations of the primitive type. As a result, everything in Java is (essentially) a class.

2.5.2 "Write Once, Run Anywhere"

Java's portability is one of its more important features. Once the source code is written, it is compiled into Java bytecode. The bytecodes are binary and platform-independent. The bytecode is then interpreted by and executed by the Java Virtual Machine (JVM). Unlike bytecodes, the JVM is targeted for specific platforms. This two step process is intended to ensure that once a programmer compiles a Java source code into bytecodes, the program will work without alteration on any machine that has a JVM.

2.5.3 Java Application Programming Interface

Sun developed the Java Application Programming Interface (Java API) for use with the Java language. These classes, which are included in the Java Development Kit (JDK), assist developers in building classes and applications of their own. The Java API classes are grouped into *packages*. User-defined classes can be packaged as well.

Among other things, Java APIs support the development of applets (web-accessible programs), abstract data types (ADTs) (e.g. stack and hashtable), and network programming with sockets, Universal Resource Locators (URLs), CORBA, and remote method invocation (RMI). The Java abstract window toolkit (AWT) and "swing" packages give designers the ability to build graphical user interfaces which are platform-independent. Java also provides a way for users to incorporate legacy code into Java class definitions using the Java Native Interface (JNI).

Villacis [Vil99] suggests that Java's APIs make it a viable alternative for building integrated systems where Java serves as "glueware" that wraps legacy applications so that they fit into a component framework.

2.5.4 Java Database Connectivity (JDBC)

The JDBC API was developed as a mechanism for standardized database access [HCF97]. It provides a set of classes and interfaces that makes it possible for developers to write database applications using pure Java code. The combination of Java and JDBC eliminates the need to write several versions of an application that are tailored to specific platforms and databases. Java provides the portability, and JDBC provides an interface to almost any relational database (and some object databases). Specifically, Java and JDBC makes it possible for a developer to code a cross-platform program that connects to any database, sends SQL commands to that database, and processes the results of SQL commands.

2.5.5 Java for the Enterprise

Sun introduced a set of APIs called the Java Computing Platform for the Enterprise, which they hope will be influential in the development of enterprise-wide systems [Sun98b]. Using these APIs, developers will have the same flexibility and freedom to develop middle-tier and back-end applications that have already been demonstrated in client applications. A multi-tiered application architecture allows developers to move an application's logic into one or more server components, thereby increasing the scalability, performance, flexibility, and reliability of the application.

One of the enterprise APIs is JavaIDL [Suna], a library which adds CORBA functionality to Java. It includes a fully-compliant Java ORB that allows seamless interoperability and connectivity between Java programs and objects, and other distributed, heterogeneous resources. In addition, Java Remote Method Invocation (RMI) [Sunb] enables one to create distributed Java applications, in which the methods of remote Java objects can be invoked from a Java application executing on another Java virtual machine, possibly at a different host. Java RMI includes its own naming service.

Thomas [Tho97] suggests that the recent development of Enterprise JavaBeans API will also boost Java's use in complex commercial applications. Enterprise JavaBeans (EJB) extends Sun's JavaBeans component model, defining a component model that supports the development and deployment of client/server applications based on a multi-tiered, distributed object architecture. Thus, EJB fosters the development of thin-client, multi-tiered systems that are highly capable of supporting high-volume business transactions.

Another enterprise product, Java Blend [Sun98a], is a powerful development tool and runtime environment that employs JDBC and enables enterprises to simplify database application development. Specifically, it allows developers to create database applications that are 100% Java by automatically and transparently mapping the Java objects to database records and database records to Java objects. As a result, developers can avoid translating Java classes to database tables manually. In fact, Java Blend eliminates the need for developers to know SQL or understand database schemas.

2.6 Metadata

Metadata has been loosely defined as "data about data." It provides information to information seekers that describes the source and characteristics of a resource. It also enhances a resource's meaning and context, and provides information on how it can be accessed. Metadata is especially helpful when a resource is shared among users unfamiliar with its meaning. In particular, it is often the case that different scientists use the same data in different ways. Williams [Wil98] suggests that scientific data, in particular, is often useless without some relatively sophisticated explanation facility.

There are several metadata standards, most of which are geared to specific genres of information. For example, the Federal Geographic Data Committee (FGDC) has recently endorsed version 2.0 of the Content Standard for Digital Geospatial Metadata (CSDGM), a comprehensive standard used to describe spatial data [FGD98]. This standard is particularly important to federal agencies providing spatial data and metadata to the National Geospatial Data Clearinghouse [FGD94] as a result of Executive Order 12906 [Pre94].

The United States Machine Readable Cataloging (USMARC) [Cra89, LOC] standard was first developed in the mid-1960s by the Library of Congress. It is a very comprehensive, heavily-documented standard that has become *the* cataloging standard used by libraries.

The ISO 10303 is the Standard for the Exchange of Product Model Data (STEP) [Owe93]. It is an effort that attempts to overcome proprietary data management issues in engineering. Included in this standard are EXPRESS, an object-oriented modeling language, and the Standard Data Access Interface (SDAI), which is used to access STEP data.

Over the past few years, the Dublin Core Initiative has been working to define a core set of elements sufficient for effective electronic resource discovery on the internet [WH98]. It have been instrumental in providing information providers with a way to embed metadata into HTML documents [W3C].

Unfortunately, [Eav97] has shown that many standards are overly general and too comprehensive, making it difficult for data producers and managers to create new metadata records and to maintain those large records adequately and efficiently. On the other hand, some metadata standards do not provide *enough* information to be useful.

2.7 Summary

Research has shown that object-orientation is a very intuitive paradigm that allows designers to more directly represent the real world in their designs and code. Just a few years ago, a lack of standards, especially regarding ODBMSs, was a major limitation to more widespread use of the object-oriented paradigm. Now, however, at least two standards bodies, namely the OMG and ODMG, have been actively developing standards to help promote the use of the object-oriented paradigm.

Several researchers have shown viability of the object-oriented paradigm in schema development and computation management. For example, the Zoo system uses persistent objects to model computational entities and manage computational processes. Also, the computational proxy mechanism is used to manage the execution of computations. In addition, distributed object-based strategies have been devised to improve the efficiency of computation processing. Finally, Java is a powerful, flexible language well-suited to object-oriented modeling and database and computation management.

We have taken advantage of the intuitiveness of object-orientation, the examples of previously-developed object-oriented computation management tools, and the utility of Java to develop the general-purpose Java-based schema extension facility (JSEF). In the JSEF paradigm, JSEF objects are used to describe computational entities. Each object contains attributes that describe a specific computational entity (e.g. a scientific dataset or an execution of a data analysis program), and thus, serves as metadata. Of course, JSEF objects persist in a database, where they are kept indefinitely, shared, and queried. The functionality of JBDC allows us to manage database activity. Also, as a schema changes, designers can create new classes to represent new entities or extend classes to reflect entity modifications.

Objects can be linked together to form a schema that describes a computational process. Methods calls are used to drive the computational process. As a result, the creation of objects to describe computational processes is independent of and is complementary to techniques being used to optimize the execution of actual computations. Consequently, a data analyzer is free to create as many JSEF objects as he/she needs at any given time. Essentially, the data analyzer can “lay out” several instances of a computation before any of them are actually processed.

The capabilities described in the JSEF paradigm, including the management of databases and facilitation of relationships are supported by the JSEF API. The JSEF API also includes functionality with which a JSEF graphical user interface can be built, JSEF objects can be manipulated remotely, and the execution of computations can be invoked and monitored.

In the next chapter, we present the components of the Java-based schema extension facility. In Chapter 4 we will examine how we have applied this Java-based schema extension facility to an example computation. Chapter 5 summarizes the work and discusses our contributions to the field.

CHAPTER 3

JAVA-BASED SCHEMA EXTENSION FACILITY

3.1 Introduction

The object-oriented paradigm provides a framework by which scientists can specify and manage simulation programs effectively and efficiently. This is feasible, since the object-oriented paradigm has been shown to be a powerful modeling tool capable of representing real-world enties naturally and intuitively. In addition, the flexibility and power of the Java programming language allows us to manage databases and create OMG/CORBA- and ODMG-compatible objects, among other things.

Thus, we present the Java-based Schema Extension Facility (JSEF), a generic modeling environment in which scientists can manage scientific experiments. The development of this facility extends the computational modeling work of Ehlmann et al. [EDR93] and Ford [For95].

In Java, a set of related classes can be grouped together in a library called a *package*. The JSEF API consists of a set of class packages that are used to define a subtle, yet powerful object-based mapping from abstract schemas to the physical entities required to carry out the process of computational discovery.

JSEF objects are used to represent computational entities, which include computer files that contain scientific datasets and computational programs that analyze and process those files. These JSEF objects persist in a database. In addition, a collection of objects can be interrelated to describe a computational schema. For example, given two objects representing two datasets and one object representing an analysis program, one dataset object can be linked as input to the analysis object while the other dataset object would be linked as its output. The functionality of these objects can be used to get information about the actual computational entities or invoke the computation itself. Figure 3.1 illustrates the abstract-to-physical mapping.

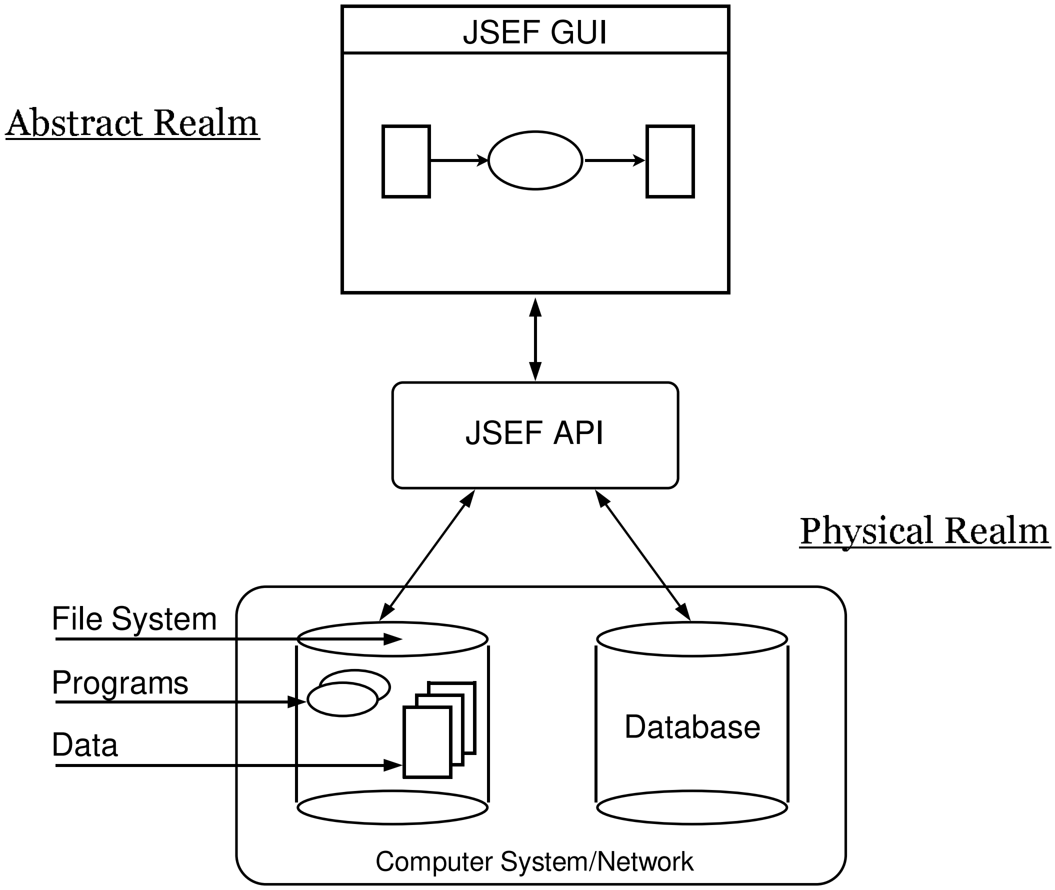

Figure 3.1. JSEF mapping abstract schemas to physical entities.

In the JSEF paradigm, schema designers translate compuational schema entities into Java classes such that each class extends from the JSEF base class, `JSEFObject`. As a result, these classes inherit metadata capability and functionality to drive a computational discovery process without referring to the computational entites directly. Examples of such classes would be *Analysis* and *Dataset*. These new classes can be understood as base classes of the schema.

Next, schema refiners derive new classes from the schema base classes to represent specific computational resources depicted in the schema. For example, from the base classes *Analysis* and *Dataset*, a schema refiner would define classes that refer to specific programs (e.g. calibration, visualizer) and specific types of data (e.g. raw data, calibrated data).

Finally, data analyzers instantiate the classes in order to carry out their analyses. Specifically, a new JSEF object is instantiated for each distinct resource. For example, a data analyzer may instantiate an object to represent a file containing raw data, an object to represent a (yet to be created) file containing calibrated data, and an object to represent an (uninvoked) execution of the calibration program.

In order to fully emulate a schema, auxiliary packages manage relationships between entities and enforce constraints on those relationships. Thus, a raw data object (representing an existing scientific dataset) can be related to a calibration program object (input), which represents a yet-to-be-invoked execution of the calibration program. The calibration object, in turn, can be related to a calibrated data object (output), which represents a yet-to-be-created scientific dataset. The functionality inherited from class `JSEFObject` and supported by auxilliary packages would allow the data analyzer to carry out the computational discovery process using these JSEF objects instead of having to delve into the specifics of setting up the computation manually.

There are a few additional auxiliary packages. For example, JSEF includes a database package (`JSEF.db`) that facilitates the storage of JSEF objects to a database, thus allowing researchers to preserve metadata about the computational process. The `JSEF.extent` package contains classes which define a database-independent extent mechanism for managing persistent objects. The functionality of the remote method invocation package (`JSEF.rmi`) allows users to develop client-server applications which allows them to distinquish between JSEF objects' *existence* as persistent metadata entities and their *functionality* which is used to drive a computation and to obtain state information about computational entities. In addition, while a schema designer can translate an existing schema into JSEF classes manually, he/she might find it useful to employ a (GUI-based) tool for schema development. Thus, the `JSEF.autocode` package can be used to translate a (GUI) schema definition into Java code that conforms to the JSEF paradigm.

We describe the `JSEFObject` class and the auxiliary packages in the following sections.

3.2 The Base Class: `JSEFObject`

All objects created in the JSEF paradigm to represent computational entities extend the base class, `JSEFObject`, either directly or indirectly. This class is part of the `JSEF` package and contains the definition of attributes and methods that are common to all classes defined in JSEF. In particular, it defines attributes which maintain basic descriptive information about JSEF objects, such as

- a name for the object
- a description of the computational resource it represents
- the name of person creating the object
- the date/time of object creation
- the date/time of the beginning and completion of the resource's computational activity
- information denoting the resource's computational status

In addition to this "created by–when" information, designers deriving new classes from `JSEFObject` may specify new attributes. These attributes are used to store information specific to the represented entity. Thus, as JSEF objects are instantiated, the attribute values of the objects (i.e. their states) serve as metadata for the entities they represent. As the state of the object is updated, the metadata is updated, too. This automates the (potentially) burdensome metadata maintenance process.

The methods defined in `JSEFObject` provide the following functionality:

capacity for persistence As mentioned earlier, JSEF objects also serve as up-to-date metadata records for a computational resource. This information is preserved indefinitely by storing the object in a database. By doing so, other scientists who may be interested in this information can simply query the database. Consequently, we have given JSEF objects the capacity for persistence by directing the `JSEFObject` constructor to store objects to the database upon creation. This operation is supported by the `JSEF.db` package (see Section 3.4) and the `JSEF.extent` package (see Section 3.5).

invocation of computational action At some point, the actual resource represented by a JSEF object must either exist, be created, or be invoked to carry out some processing. Thus, we have included (the specification of) functionality that invokes a computational action associated with the computational resource with the `process` method. Examples of computational actions include the creation of a dataset or the invocation of a computational program. Also, the `isReady` method specifies functionality that allows a JSEF object to check its *readiness*. Readiness indicates whether or not the pre-requisite(s) for the computational action to take place are satisfied. A JSEF object can also specify the option to have its computational action performed in an external environment (see Section 3.6.3).

maintaining and sharing object status Since JSEF objects represent computational entities, it is useful for JSEF objects to have the capacity to keep track of the computation status of its corresponding entity. Consequently, there is a `getStatus` method that returns an integer status code, and `setStatus(int)` that allows users to designate that code. There are also methods that allow the specification of status messages.

In addition, it may be that the status of one entity is influenced by the status of another entity. Thus, the `JSEFObject` class defines functionality that allows objects to share their status information. In particular, the `checkDependencies` method allows an object to check the status of another object it may depend on for status information or for something else. Conversely, the `notifyDependents` method allows an object to inform other "interested" objects about itself.

restricted access to attributes Access to attributes of the `JSEFObject` class is protected by *accessor* and *mutator* methods. These methods enforce the object-oriented concept known as *information hiding*. Specifically, they provide a uniform mechanism to access attribute values and help ensure the validity of attribute values. The code generator (see Section 3.7) perpetuates this paradigm to user-defined JSEF classes extending from `JSEFObject`.

JSEF object attributes can be updated freely until the object becomes *locked*. The capacity of JSEF objects to enter a locked state is intended to preserve the final state of the resource after its computational action has begun. When locked, a user cannot alter the value of the object's attributes. In the case of pre-existing resources, the user should lock the corresponding JSEF object once the proper attribute values have been set. Users can check for a locked state with the `isLocked` method.

It is important to note that in addition to having a single, distinct JSEF object for each resource, restricting access to attribute values by locking JSEF objects helps us to maintain the distinctions between different resources (e.g. different dataset files, distinct invocations of a computational program), regardless of how similar they may be.

Much of this functionality has been defined abstractly, since we cannot pre-define the actual functional specification for all entities. For example, a specification for an object sharing its status information depends on the computational resource being represented, and it may also depend on the relationships that object has with other objects, among other things. Of course, designers and refiners can define functionality as needed for JSEF classes in addition to that mentioned above.

3.3 Relationship package

In order for us to model real-world schemas, there must be a mechanism by which we can specify a relationship between two entities. The `JSEF.relationship` package, which is anchored by the `Relationship` class, defines classes that manage *one side* of a binary relationship between two entities[1]. The attributes of the `Relationship` class include references to the entities that a JSEF object is related to. Its methods provide access to the related entities, add and remove entities to and from the relationship, and determine if the relationship is *complete*. A relationship is considered to be complete if some condition is met (e.g. there are a sufficient number of entities referenced in the relationship object) such that the parent JSEF object (i.e. the one managing the relationship) can invoke the computational action associated with the resource it represents. Often, a relationship's completeness is a determining factor in a JSEF object's readiness.

[1]When we use the term *Relationship*, we are most often referring to instances of the `Relationship` classes. However, we should emphasize that there is a difference between the actual relationship between entities and instances of the `Relationship` classes that work in pairs to implement the relationship mechanism in JSEF.

3.3.1 Example relationship

Let us consider how relationship objects function to manifest the actual relationship by using the *input* relationship between class `Analysis` and class `Dataset` relationship illustrated in Figure 3.2. The `Analysis` class defines a `Relationship` attribute to manage its *input* relationship to the `Dataset` class. Similarly, the `Dataset` class defines a complementary `Relationship` attribute to manage its relationship, *consumedBy*, to the `Analysis` class.

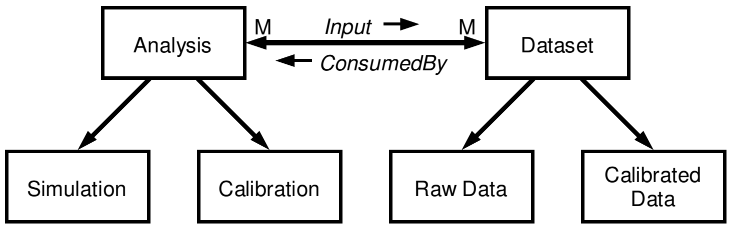

Figure 3.2. Example Entity Relationship and Hierarchy.

3.3.2 Constraining relationships

JSEF requires that all elements bonded to an entity by the same relationship have the same data type. In other words, `Analysis` objects can only be related to `Dataset` objects through its *input* (side of the) relationship (note that instances of `RawData` and `CalibratedData` are `Dataset` objects). Likewise, `Dataset` objects can only be related to `Analysis` objects through its *is input to* (side of the) relationship. These restrictions are imposed transparently by defining *data type constraints* on the appropriate relationship objects.

Designers can also impose *cardinality constraints* on relationships. There are two types of cardinality constraints: maximum cardinality and minimum cardinality. A cardinality constraint on a relationship imposes a limit on the number of elements that can be bonded to an entity through that relationship. For instance, by designating a maximum cardinality constraint for the *input* relationship, a designer can specify that an analysis can accept at most a particular maximum number of datasets as input. Conversely, imposing a minimum cardinality constraint means that the analysis would need at least a particular minimum number of datasets as input.

In order to enforce the behavior of a new relationship, the designer creates a Java *package class* within the same source code file as the class defining the relationship. A package class is a class that is only available for use by other classes in the same package. A package class for a relationship extends directly from class `Relationship`. For example, in the case of the `Analysis` class, the package class `AnalysisInput` would be defined within source file containing the `Analysis` class to implement its *input* relationship. It contains code in its `addElement(JSEFObject newElem)` method to ensure that the object being added conforms to the data type, `Dataset`. There is also code in its `isComplete` method to validate the cardinality constraints.

Table 3.1 shows a pseudo-code definition of the `Analysis` class implementing the *input* relationship with a maximum of three `Dataset` objects.

Relationships are defined as instance attributes so that each class instance will support its own instance of the relationship. Consequently, for three instances of the *Analysis* class, each would manage its own distinct instance of the *input* relationship. Of course, a single JSEF class can define multiple relationships, and it can be related to several other classes.

3.3.3 Relationship inheritance

Since relationships are defined as instance attributes, JSEF subclasses inherit the relationships defined in the parent class. Thus, the `Simulation` class maintains the relationships with the `Dataset` class it inherited from the `Analysis` class.

In addition, a subclass can further constrain its parents' relationships by defining additional data type and/or cardinality constraints. For instance, we can define a

Table 3.1. Pseudo-code illustrating *input* relationship of the `Analysis` class.

```
public class Analysis extends JSEFObject {

  public Relationship input;
  ...

  protected void initialize() {
    input = new AnalysisInput();
    ...
  }
  ...
}

class AnalysisInput extends Relationship {
  ...

  public boolean addElement(JSEFObject newElem) {

    if (!(newElem instanceof Dataset))
      return (false);
    } else {
      return (super.addElement(newElem));
    }
  }

  ...

  public boolean isComplete() {

    if (getCount() > 3) {     // Ensure that there are a maximum
      return (false);         // of three related elements.
    } else {
      return (true);
    }
  }
}
```

new data type constraint to allow `Calibration` objects to only be related to `RawData` objects through its *input* relationship. Figure 3.3 shows this refinement.

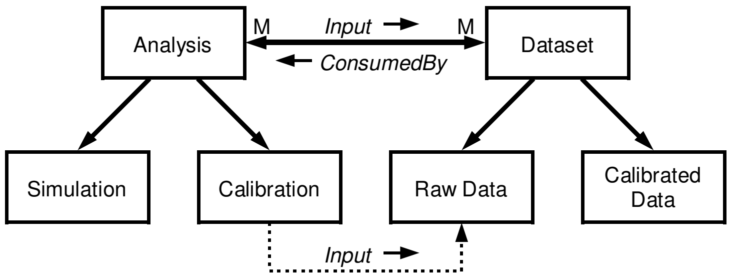

Figure 3.3. Refinement of Entity Relationship.

This refinement is implemented by creating a new package class in the same source file as the `Calibration` class to extend from the parent's implementation of the same relationship. Specifically, we extend the `CalibrationInput` class from the `AnalysisInput` class. Then, we override the `addElement(JSEFObject newElem)` method to impose the additional data type constraint.

As we see in Table 3.2, the package class `CalibrationInput` has been refined so that only `RawData`, and not just any `Dataset`, will be accepted into the *input* relationship. At the same time (unless later specified), a `RawData` object could still be used as input by any `Analysis` object and not just `Calibration` objects. Also, note that the constraints imposed by the parent are still enforced by the call to `super.addElement()`.

The rules for refining a relationship constraint are:

- A data type constraint $\mathcal{A}$ can be refined to data type $\mathcal{B}$ if and only if $\mathcal{B}$ is a subclass of $\mathcal{A}$ or $\mathcal{A}$ is a Java interface and class $\mathcal{B}$ implements that interface.

- A maximum cardinality constraint value $\mathcal{X}$ can be refined to $\mathcal{Y}$ if and only if $\mathcal{X}$ is greater than $\mathcal{Y}$.

- A minimum cardinality constraint value $\mathcal{X}$ can be refined to $\mathcal{Y}$ if and only if $\mathcal{X}$ is less than $\mathcal{Y}$.

Table 3.2. Pseudo-code illustrating entity relationship refinement.

```
public class Calibration extends Analysis {

  // input attribute inherited from parent.
  ...

  protected void initialize() {
    input = new CalibrationInput();
    ...
  }
  ...
}

class CalibrationInput extends AnalysisInput {

  public boolean addElement(JSEFObject newElem) {
    if (!(newElem instanceof RawData))
      return (false);
    } else {
      return (super.addElement(newElem));
    }
  }
  ...

  // isComplete() inherited from the parent
}
```

By using package classes to implement relationships, we are able to keep the relationship declaration (or refinement) and its implementation together (i.e. within the same source file), emphasizing its context-specific meaning within the class hierarchy. In addition, since constraints are enforced on the relationship, and since the relationship is a member of the class, all class instances will have its own relationship member and enforce the same constraints on a given relationship. Thus, all (sub-)instances of the `Analysis` class would have its own relationship member, *input*, and impose the constraints on that relationship encoded in its corresponding package class.

3.3.4 Role of `JSEF.relationship` package

Package `JSEF.relationship` is helpful for several other reasons. First, it helps us distinguish between a relationship between two entities and a reference-type attribute of the class. Also, it provides a framework by which we can enforce the behavior of a relationship without cluttering a JSEF class with methods to support the relationship, methods which may be unrelated to the primary functionality of that class. In addition, there was no pre-defined binding in the ODMG 2.0 release for relationships in Java [CB97].

3.4 `JSEF.db`: The database package

The `JSEF.db` package is anchored by the base class `DBManager`. This class defines (abstract) functionality to handle interactions between JSEF objects representing computational entities and a database that is used to store those objects. This functionality includes:

- opening and closing a connection to a database
- loading and storing JSEF objects
- querying the database, and
- managing database transactions
- loading Java database connectivity (JDBC) drivers for a specific database engine

3.4.1 The JSEF persistence model

JSEF employs a basic persistence model that relies on classes defined in the `JSEF.extent` package (see Section 3.5). Specifically, when a JSEF object is created, it is stored in the extent corresponding to its class and in the extents corresponding to each of its parent classes. All JSEF extents are contained within a single set of extents. The database manager maintains a reference to this `SetOfExtent` object. Finally, only one object (albeit a compound object), the `SetOfExtent` object, is ever explicitly stored to or loaded from the database.

In JSEF, as in most (if not all) persistent object paradigms, a database connection must be open and a transaction must be in progress while JSEF objects are being created and/or modified in order for changes to become persistent. Thus, in order to work with persistent JSEF objects, the data analyzer should first create a `DBManager` object to correspond to the database to be used. Next, the (`create` or) `open` method should be called on that object to (create and) open the corresponding database. Then, the data analyzer should begin a database transaction.

When a transaction is begun, the `DBManager` object reads the `SetOfExtent` object from the database into memory. At this point, data analyzers may either obtain existing JSEF objects by name, query, or some other means, or create new JSEF objects which are stored in the `SetOfExtent` object on construction. When all the processing has been completed, the transaction must be committed. Committing the transaction causes all the changes made to the desired objects to become persistent, since modifying those objects modifies the persistent `SetOfExtent` object.

3.4.2 Creating a `DBManager` class for a specific database

The `DBManager` class defines basic and abstract functionality needed to provide persistence for JSEF objects. In order to support a given database, a schema designer has to design a class that extends `DBManager` and implements its (abstract) functionality according to that database's interface. Database opening, closing, querying, and transaction management are examples of functionality that tends to vary across database implementations. For example, we have defined the `FileManager` class, which extends class `DBManager`, and is tailored to provide primitive persistence management with binary files using Java's *object serialization* feature. We also defined the `PoetDBManager` class that is tailored to interact with the Poet object database.

3.5 `JSEF.extent`: Database-independent extent mechanism

Extents serve as a grouping mechanism for objects stored in object databases in a similar fashion that tables group tuples in relational databases. As mentioned previously, when a JSEF object is created, it is automatically stored in the extent corresponding to its class and in the extents corresponding to each of its parent classes. Further, all JSEF extents are contained within a single set of extents.

We have defined a few classes in the `JSEF.extent` package to support the extent mechanism. The `Extent` class defines an object that maintains a platform-independent extent mechanism. It maintains a reference to the elements of the extent, and it implements functionality from the `java.util.Set` interface for (future) compatibility with object database systems that (will) support the ODMG Java binding. This functionality includes adding and removing elements and returning the number of elements in the extent.

`Extent` objects are named by the *primary class* whose objects will be maintained by it. In addition, the `ExtentElement` interface is used to denote primary JSEF classes. For example, class `Analysis` implements the `ExtentElement` interface. Consequently, there is an `Extent` object named *JSEF.Analysis* that maintains a reference to every (sub-)instance of the `Analysis` class. Thus, while `Simulation` objects would be referenced by an `Extent` object named *JSEF.Simulation*, those objects would also be referenced by the *JSEF.Analysis* `Extent` object, too, since class `Simulation` extends from class `Analysis`.

It is important to note that only primary JSEF objects require an extent for their persistence. For example, descendents of the `JSEFObject` class require an extent, while `Relationship` instances do not. This is because primary objects, such as

`JSEFObject` instances, should be available directly, while `Relationship` instances need not be, since they are only meaningful in the context of `JSEFObject` instances.

In addition, there is a special *pending extent* which monitors JSEF objects whose computational action (i.e. processing) has been deferred. This extent is discussed further in Section 3.6.3.

All `Extent` objects are maintained by a `SetOfExtent` object. While `SetOfExtent` class does not define a name attribute, it, too, implements the `java.util.Set` interface for the sake of compatibility. A `SetOfExtent` object and a `DBManager` object work together to provide JSEF object persistence. Since every persistent JSEF object is reachable from a `SetOfExtent` object, persistence is achieved by storing the `SetOfExtent` object in a database in the context of a `DBManager` object transaction.

In JSEF, there is only one `SetOfExtent` object per database. A `SetOfExtent` object is created by a `DBManager` object for a database whenever the `create` method is called on a `DBManager` object. When `open()` is called, an existing `SetOfExtent` object is retrieved from the database. Whenever a new JSEF object needs to be stored, the `SetOfExtent` object stores the new object in the appropriate extent hierarchy. If one or more `Extent` objects corresponding to the class hierarchy of the new object do not exist, the `SetOfExtent` object creates them.

3.6 `JSEF.rmi`: Using JSEF objects remotely

In the computational discovery process, data analyzers create and manipulate JSEF objects using a *JSEF client program*, a program that creates and/or utilizes JSEF objects to perform a computation. However, it is not essential that a JSEF client program access JSEF objects directly. The `JSEF.rmi` package effectively bundles JSEF so that data analyzers can create and manipulate JSEF objects remotely.

In order to implement Java RMI, a designer must first create *remote interfaces* that define functionality that can be accessed remotely. These interfaces must extend either directly or indirectly from Java's `java.rmi.Remote` interface. Each remote method must declare that an exception may occur while Java's remote mechanism is handling the method execution. Also, there must be at least one class that implements the remote interface. This class should directly or indirectly extend Java's `java.rmi.server.UnicastRemoteObject` class.

3.6.1 JSEF remote interfaces and wrapper classes

In JSEF, there is a remote interface for class `JSEFObject` called `rJSEFObject`. It mirrors the functionality of the `JSEFObject` as described in Section 3.2. Essentially, all public methods in the class are made available remotely by including them in the remote interface. In fact, all JSEF classes that a schema designer wants to make available to a remote client should have a corresponding remote interface. Specifically, those methods of a class that will be available to the remote client should be included in the remote interface definition corresponding to that class.

Wrapper classes implement the remote interfaces in the JSEF paradigm. These classes simply map the functionality guaranteed by the remote interfaces to the existing functionality of the corresponding JSEF objects. JSEF wrapper classes extend the wrapper base class, `JSEFWrapperObject`, which extends `java.rmi.server.UnicastRemoteObject`. This class has one member attribute, a reference to the JSEF object being wrapped. The methods are implementations of the remote interface methods that simply call the corresponding method on the actual JSEF object. Table 3.3 shows an example of how a wrapper class wraps a `JSEFObject` instance.

Table 3.3. Pseudo-code of wrapper class implementation.

```
public class JSEFObject ... {
  ...

  public String getName() { return (name); }
  ...
}

public interface rJSEFObject extends java.rmi.Remote {
  // Remote interface for class JSEFObject
  ...

  public String getName() throws java.rmi.RemoteException;
  ...
}

protected class JSEFObjectWrapper extends JSEFWrapperObject
                                  implements rJSEFObject {
  // Wrapper class for class JSEFObject
  // Member attribute "objRef" is inherited from parent.

  protected JSEFObjectWrapper(JSEFObject toBeWrapped)
            throws java.rmi.RemoteException {
    objRef = toBeWrapped;
  }
  ...

  public String getName() throws java.rmi.RemoteException {
    return (objRef.getName());
  }
  ...
}
```

3.6.2 Obtaining remote JSEF objects

Java remote objects may be made available through a server object that is registered by name to the RMI registry with an execution of Java's *rmiregistry* program. Remote objects are bound to the RMI registry. The rmiregistry then listens for requests for named objects. When a request for a given name is received, the RMI registry returns a remote reference to the object bound by that name to the calling (client) process.

In the JSEF paradigm, there is a single server object that is bound to the rmiregistry. This server object implements the `JSEFServer` interface, which defines a `create` method for each type of object that should be directly available to the client. The `create` method creates a JSEF object on the server and returns a remote reference to that object to the client. When a remote client retrieves a `JSEFServer` implementation, it is essentially retrieving a "JSEF remote object factory" with which it can create new JSEF objects on the server (or identify existing JSEF objects) and obtain a remote handle to them.

Table 3.4 shows pseudo-code of a JSEF client acquiring and using JSEF objects remotely. It makes reference to classes depicted in Figure 3.2.

While, JSEF allows for the remote manipulation of all JSEF objects, some JSEF objects are not available through the `JSEFServer` interface. For example, `Relationship` objects cannot be retrieved through a `JSEFServer` object because they are meaningless when they stand alone. However, a remote reference to the *input* relationship is available from the `Analysis` that manages it via the call to `anal.getInput()`.

3.6.3 Remote execution management

JSEF also includes a remotely-available `ExecutionManager` class that, along with the special *pending extent*, allows schema refiners to separate the invocation of the computational action from the actual processing involved with that action. JSEF classes may define a method that allows an instance to make itself available for external processing. In fact, the `process` method can be defined to perform this external submission. When this submission method is called on an object, the

Table 3.4. Pseudo-code for a client using remote JSEF objects.

```
public class TestClient {

  // Get the server object.
  JSEFServer server = Naming.lookup("JSEFServer");

  // Get a DBManager for a file-based database.
  DBManager dbm = server.createDBManager("FileManager","myDatabase");

  dbm.open();                        // Open the database connection.
  dbm.beginTransaction();            // Begin a transaction.

  // create JSEF objects
  rAnalysis anal = server.createAnalysis("Simulation", "sim1");
  rDataset  dset = server.createDataset("RawData", "rawData1");

  // assign the dataset as output from the analysis.
  anal.getOutput().addElement(dset);

  // Run a simulation.  dset references the output of the simulation.
  anal.process();

  dbm.commitTransaction();           // Save the results of processing.
  dbm.close();                       // Close the database.
  }
}
```

object is placed in the pending extent where it can be made available to an external execution environment through an `ExecutionManager` object.

An *execution client* is a program that links pending JSEF objects with some execution environment (e.g. DQS, Legion [GW+97]). An execution client obtains a reference to a `ExecutionManager` object from a `JSEFServer` object. The `ExecutionManager` class defines methods that return references to objects that have been submitted to the pending extent. Next, the execution client calls methods on an `ExecutionManager` objects to obtain references to objects in the pending extent that are ready to carry out their computational action. For example, the `getNext` method returns a remote reference to the next available (i.e. ready) JSEF object in the pending extent. When a JSEF object's computational action is completed, the object is removed from the pending extent.

3.7 `JSEF.autocode`: Automating JSEF code generation

The `JSEF.autocode` package includes classes whose functionality is used to encode a computational schema specification and translate it into JSEF classes. Any (GUI) development tool designed to help designers specify JSEF schemas should utilize the functionality provided by this package. As the schema is being specified, the development tool calls methods to capture

- package names, class names, access modifiers, and interfaces to be implemented
- class attribute details, including attribute name, data type, access modifiers, and a default value, which are stored in `JSEF.autocode.Attribute` objects
- relationships between entities/classes, including relationship name, data type constraint, maximum and minumum cardinality constrants, and information about the inverse relationship, which are stored in `JSEF.autocode.Relationship` objects

Class specification information is captured in a `JSEFClassDef` object and persists in a database. Table 3.5 shows a partial definition of the `JSEFClassDef` class.

A `JSEFClassDef` object stores the name of the package that the new class will belong to in the `packageName` member. By default, JSEF classes belong to the `JSEF` package. It also stores a name for the class in the `className` attribute. Access modifiers[2] for the new class are stored in the `modifiers` attribute. The `superclassName` attribute stores the name of the class being extended. By default,

[2]Java access modifier codes can be found in the Javadoc documentation of the `java.lang.reflect.Modifier` class.

Table 3.5. Partial definition of the `JSEFClassDef` class.

```
public class JSEFClassDef implements ExtentElement {

  protected String packageName = "JSEF";
  protected int    modifiers   = 0;
  protected String className;
  protected String superclassName;
  protected List   interfaces;
  protected List   attributes;
  protected List   relationships;

  public JSEFClassDef(String cName, String pName, boolean isCopy)
         throws JSEFException { ... }

  public void addModifier(int mod) { ... }
  public boolean addInterface(String i) { ... }
  public boolean addAttribute(Attribute a) { ... }
  public boolean addRelationship(Relationship r) { ... }
  public boolean isComplete() { ... }

  // other class methods ...
}
```

JSEF classes extend `JSEFObject`. The `interfaces` attribute maintains a list of interfaces to be implemented by the new class.

In addition, a `JSEFClassDef` object two additional member lists, one of `Attribute` objects named `attributes` that specify the attributes of the new class, and one of `Relationship` objects named `relationships` that specify the relationships (or their new constraints) of the class.

`JSEFClassDef` objects must be *complete* in order to be useful. For example, only complete `JSEFClassDef` objects can be stored in a database. Additionally, a `JSEFClassDef` object must be complete before it can be used by the `Encoder` class to generate the source code. A `JSEFClassDef` object is complete if all the information necessary to describe a JSEF class has been included in the `JSEFClassDef` object. Particularly, the class name must be a valid Java identifier, and any `Attribute` and

Relationship objects must also be complete. An Attribute object is complete if it contains a valid Java identifier as the attribute name and if that identifier does not duplicate any other identifier associated with the new class, and if a data type for the attribute has been defined. A Relationship object is complete if it contains a valid Java identifier as the relationship name and if that identifier does not duplicate any other identifier associated with the new class. In addition, a Relationship object must contain a valid identifier indicating the name of the member attribute of inverse relationship and the name of the class implementing the inverse relationship.

In order to generate Java code, the user calls the generateCode method of the Encoder class with a complete JSEFClassDef object as the parameter. The generateCode method produces source code for a new Java/JSEF class by reading in a template file and replacing template placeholders with information from the JSEFClassDef object. Table 3.6 shows a portion of a template file.

The generated class includes the specified class access modifiers, superclass, and interface list, along with the declaration of attributes. An accessor and mutator method is generated for each attribute. Special attention is given when handling the inclusion of class attributes representing relationships. Specifically, when a class defines a new Relationship attribute or refines an inherited one, information about the inverse relationship is hard-coded into the class. We also incorporate code to enforce relationship constraints by defining classes for each new (or modified) relationship.

The JSEFServer interface provides remote access to JSEFClassDef objects and the Encoder object. Thus, schema designers and refiners can develop applications that allow JSEF classes to be defined remotely.

We have implemented a limited, yet functional, text-based Code Generator prototype for creating JSEF classes. When the Encoder class is executed as an application, it allows users to create brand new classes or extend existing classes. Users are prompted to specify a name for the new class and other details, such as

Table 3.6. Portion of a template file for a JSEF automatic code generator.

```
  // _NAME_     stands for the attribute's name
  // _CAPNAME_  stands for the attribute's name with first letter in caps
  // _TYPE_     stands for the attribute's data type
  // _ACCESSOR_ indicates the code up to _END_ is a template accessor
  //            method
  // _MUTATOR_  indicates the code up to _END_ is a template mutator
  //            method

_ACCESSOR_
  // Javadocs with template markups here...
  public _TYPE_ get_CAPNAME_() { return (_NAME_); }
_END_

_MUTATOR_
  // Javadocs with template markups here...
  public boolean set_CAPNAME_(_TYPE_ newValue) {

    if (isLocked())
      return (false);

    _NAME_ = newValue;
    return (true);
    }
_END_
```

attribute declarations and relationship declarations and/or refinements. Section 4.3 describes how we used the Code Generator to extend existing base classes. In fact, the Code Generator could be used to generate much of the code for the base classes themselves.

If no interface is used (i.e. the classes are designed by hand), designers should incorporate the JSEF principles addressed previously when specifying their classes. These include the extension of the `JSEFObject` class, the use of accessor and mutator methods, and the proper specification of relationships and their inverses in class definitions and the implementation of relationship, along with constraint enforcement defined in a package class.

3.8 Summary

This section describes the Java-based Schema Extension Facility. It is a modeling environment that allows scientists to effectively manage computational experiments. This environment is based on a set of core classes that facilitate the describing of computational entities, the relationships between entities, the persistence of objects representing computational entities, the remote use of JSEF objects, and the ability to automate the design of JSEF classes to represent computational entities.

Schema designers translate (either by hand or via an interface) existing scientific analysis schemas into Java code such that each entity in the schema extends the `JSEFObject` class. Auxiliary classes manage relationships between entities and constraints on those relationships, control the object persistence mechanism, and facilitate the remote availability of JSEF objects. Schema refiners will tweak the classes so that new subclasses encapsulate the state and function of specific computational resources. Finally, data analyzers will instantiate the classes in order to carry out their analyses.

In the next chapter, we introduce a computational paradigm and describe how we have applied JSEF to it. In Chapter 5 we summarize the work and offer suggestions for future work.

CHAPTER 4

USING JSEF TO MANAGE COMPUTATIONS

When scientists apply the JSEF paradigm to their computational discovery processes, they are able to manage and carry out their experiments using JSEF objects and do not have to be concerned with the specifics of performing the computation manually. When using JSEF, the following tasks are either eliminated or simplified:

- locating and organizing datasets,
- locating program binaries and scheduling their execution, and
- keeping a log of the execution and results of computations.

Without JSEF, each of these tasks becomes more difficult as the number of programs used, the volume of data analyzed or produced, the number of scientists involved, or the length of time the computation has been utilized increases.

In this chapter, we introduce a computational physics experiment. Then we describe how we apply the JSEF paradigm to assist in this computational process. Specifically, we explore how the schema designer defines basic classes for computational strategies that extend `JSEFObject`. Next, we show how the schema refiner extends those classes to create new classes specific to a particular computational strategy. Finally, we describe how the data analyzer instantiates objects of the refined classes to carry out the process of computational discovery.

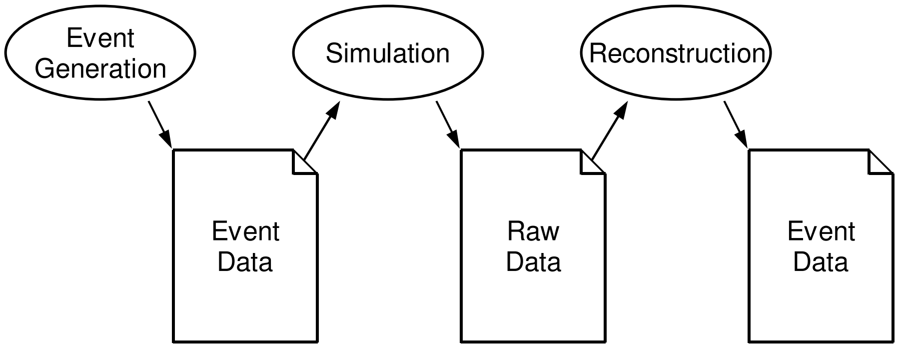

Figure 4.1. Simulation of detector response.

4.1 Simulating Atomic Particle Collisions

The Continuous Electron Beam Accelerator Facility (CEBAF) at Jefferson Lab is used to conduct experiments in nuclear physics. It features a continuous beam of electrons that is directed at fixed targets in each of three experimental halls. Interactions between the electron beam and target occur within complex detector systems that collect data about each interaction event. An event is typically the interaction between an electron and a proton that results in several particles of different types and energies flying from the target area into the detector.

The major computational activities in experimental physics are the simulation of detector response and the analysis of experimental data. Many times, the amount of simulated data far exceeds the experimental data. Thus, the simulation must be carefully managed to avoid repetition, to ensure accuracy and to ensure that all necessary simulations are carried out.

A typical computation for a Jefferson Lab experiment is illustrated in Figure 4.1. In this figure, the ovals represent computations, the pages represent datasets, and the arrows represent the flow of the computational process.

We use this example to explain the capabilities of the JSEF system. Before describing the system, we give a brief description of this important, and relatively simple, computation.

There are three computational steps. The first step is the generation of physical events to simulate the interaction between an electron and a proton. The result of this is an event dataset. The second step is the simulation of the response of the detector to each event. Specifically, for each event in the event dataset, there is an output event that mimics the data that would be produced by the experimental equipment. This dataset is in the same format as raw data collected from the detector during an experiment. It is labeled "Raw Data" in Figure 4.1.

The third step in the computation is to *reconstruct* event data from raw data. In a real experiment, the raw data collected from the detector is analyzed to determine what physical event triggered the detector response. For instance, the reconstruction might determine that an electron and a proton with specific momenta were emitted from the target area. The output of reconstruction is event data, in the same format as the data produced by the event generator.

Essentially, the third step in the computation takes the simulated detector responses to the simulated events and reproduces the original events. The purpose of this third step is to determine what fraction of real events are likely to be correctly recognized by the combination of detector hardware and reconstruction software.

This computation scheme will be applied thousands of times to generate hundreds of millions of simulated events for use by a few hundred scientists.

4.2 Schema design

In this section, we highlight the role of the schema designer. In particular, we describe how JSEF schema classes were created to represent the basic entities in this computation and how relationships were defined between those entities.

4.2.1 Base classes and relationships

The most basic entity-relationships in this experiment are the ones between computational programs and scientific datasets. A computational program uses zero or more datasets as input, performs some processing and/or analysis on that data, and produces zero or more output datasets. In contrast, a dataset is produced by a computational program and can be consumed as input by many computational programs. It is important to note that these entities and the relationships between them are the basis of a large class of computational strategies and are not unique to this atomic particle collision simulation experiment.

We created the `Analysis` and `Dataset` classes to model the computational entities mentioned above. Specifically, each instance of the `Analysis` class represents a distinct execution of a computational program, while instances of the `Dataset` class represent files containing scientific data. It is important to note that while we have defined JSEF classes to represent these computational entities, they are external to JSEF. In general, though, entities represented by JSEF do not have to be external to JSEF.

Classes `Analysis` and `Dataset` both extend `JSEFObject` and inherit the functionality described in Section 3.2. Figure 4.2 illustrates the `Analysis` and `Dataset` classes and the relationships between them.

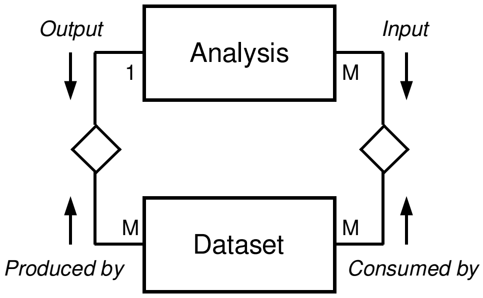

Figure 4.2. Relationship between `Analysis` and `Dataset` objects.

The relationships between `Analysis` and `Dataset` objects are implemented by `Relationship` objects that inherit the functionality described in Section 3.3. These `Relationship` objects are referenced as attributes in the `Analysis` and `Dataset` classes and are instances of classes that extend `Relationship`. Lines 2–3 and lines 18–19 of Table 4.1 shows these `Relationship` attribute definitions.

Table 4.1. Partial definitions of the `Analysis` and `Dataset` classes.

```
1 public class Analysis extends JSEFObject {

2   public Relationship input;
3   public Relationship output;
4   public Relationship schema;
5   public ParameterSet params;
    // other attribute definitions

6   protected void initialize() {
7     input  = new AnalysisInput(this);
8     output = new AnalysisOutput(this);
9     schema = new AnalysisSchema(this);
      ...
    }

10   protected void checkDependencies() { ... }
11   protected void notifyDependents() { ... }

12  public String getExecutableName() { ... }

13  public boolean isReady() { ... }
14  public Object process() { ... }
15  public void submit() { ... }
16  public void dequeue() { ... }
    // other class methods
  }
```

(continued in Table 4.1-B)

The `Analysis` class defines a `Relationship` attribute named `input` to reference `Dataset` objects to be used as input. It defines another `Relationship` attribute named *output* to reference those `Dataset` objects to be generated as output. Finally, it defines a `Relationship` attribute named `schema` to reference the analysis strategy in which an `Analysis` object participates. We discuss the `AnalysisStrategy` class in

Table 4.1-B.

```
17 public class Dataset extends JSEFObject {

18   public Relationship producedBy;
19   public Relationship consumedBy;
     // other attribute definitions

20   protected void initialize() {
21     input  = new DatasetProducedBy(this);
22     output = new DatasetConsumedBy(this);
       ...
     }

23   public boolean setExists(boolean b) { ... }
24   public boolean exists() { ... }

25   public String getLocation() { ... }
26   public boolean setLocation() { ... }

27  protected void checkDependencies() { ... }
28  protected void notifyDependents() { ... }

29  public boolean isReady() { ... }
30  public Object process() { ... }
31  public void create() { ... }
     // other class methods
   }
```

Section 4.2.4. The classes `AnalysisInput`, `AnalysisOutput`, and `AnalysisSchema` used in lines 7–9 of Table 4.1 are instantiated to implement the `Analysis` side of these relationships. See Section 3.3.2 for a discussion of classes defined to implement relationships. Also, Table 3.1 shows a sample definition of the `AnalysisInput` class.

Similarly, the `Dataset` class defines a `Relationship` attribute named `producedBy` (line 18) to reference the `Analysis` object used to produce the dataset. It also defines a `Relationship` attribute named `consumedBy` (line 19) to reference the `Analysis` objects using it as input. Consequently, `Dataset` class source file includes definitions of the DatasetProducedBy and DatasetConsumedBy classes to implement the Dataset site of these relationships. Lines 21–22 show the instantiation of these relationship classes.

4.2.2 Functionality of class Analysis

The Analysis class implements the basic functionality inherited from class JSEFObject in the context of a computational program. It also defines new attributes and methods specific to computational programs. Table 4.1 includes a listing of the key attributes and methods of the Analysis class.

The Analysis class defines a ParameterSet attribute named params (line 5) to manage parameters for the computational program. The ParameterSet class supports the Analysis class by specifying attributes and methods to maintain a series of name–value pairs that serve as parameters to the computational program represented by the Analysis object.

The Analysis class implements the checkDependencies method (line 10) so that an Analysis object can check on the status of the Dataset objects upon which it depends to carry out its processing. In order for an Analysis object to carry out its processing, all input Dataset objects must exist, i.e. the input relationship must be complete. The input relationship is complete if it has the proper cardinality and if all those Dataset objects represent existing dataset files.

Conversely, the Analysis class implements the notifyDependents method (line 11) so that an Analysis object can share its processing state with related (and dependent) Dataset objects. Essentially, the actual datasets exist when the Analysis object that produces them completes its processing successfully. When the notifyDependents method executes, the Analysis object prompts each of its dependent Dataset objects to check its processing state.

There is also the getExecutable method (line 12) that returns the name of the executable program associated with this class of objects. This method is defined abstractly; we leave its definition to the schema refiner, since this base class is not (and should not be) tailored to any particular computational program.

An `Analysis` object's readiness can be checked by calling the `isReady` method (line 13). An `Analysis` object is considered ready if the `input` relationship is complete and if all the necessary parameters have been specified. Also, the *output* relationship must be complete. The `output` relationship is complete if it has the proper cardinality and if none of the datasets represented by `Dataset` objects of the relationship exist. Note that `Analysis` object must be ready in order to carry out its processing.

A call to the `process` method (line 14) on an `Analysis` object implies that the caller wants to invoke the computational program represented by that object. We have defined the `process` method to gather and return the object's state information required to invoke the computational program. An `Analysis` object's state includes the name, location, and other information pertaining to the input and output datasets represented by `Dataset` objects in the `input` and `output` relationships, the parameters obtained from the `params` attribute, and the name and location of the executable file associated with this `Analysis` object.

The `Analysis` class defines the `submit` method (line 15) that allows an `Analysis` object to submit itself to the pending extent, as described in Section 3.6.3. From the pending extent, the object can be retrieved for external processing at a later time by the `ExecutionManager`, or it can removed from the pending extent with a call to the `dequeue` method (line 16).

4.2.3 Functionality of class `Dataset`

The `Dataset` class also implements the functionality it inherits from `JSEFObject` in the context of a dataset file. In addition, it defines functionality to monitor the location of datasets. Table 4.1 lists key methods of the `Dataset` class.

The methods `exists` and `setExists` (lines 23–24) allow users to determine and specify whether or not the dataset represented by this `Dataset` object exists. Note that a `Dataset` object can represent a yet-to-be-created dataset, or it can also represent a previously created dataset, even if JSEF objects were not used to generate that dataset.

Also, the methods `setLocation` and `getLocation` (lines 25–26) allow users to specify and retrieve of the (future) location of the (yet-to-be-created) dataset file represented by this `Dataset` object, respectively. The location string can be a filename, a URL, or some other text that identifies how the actual dataset can be obtained.

The `Dataset` class implements the `checkDependencies` method (line 27) so that a `Dataset` object can check on the status of the `Analysis` objects designated to be its producer. In order for a `Dataset` object to invoke the generation of the dataset it represents, the `producedBy` relationship must be complete, i.e. exactly one `Analysis` must be in the `producedBy` relationship, and it must be ready for processing. Note that if the producing `Analysis` object has already carried out its processing, then the dataset represented by this `Dataset` object would already exist.

Similarly, the `Dataset` class implements the `notifyDependents` method (line 28) so that a `Dataset` object can share its processing state with dependent `Analysis` objects linked via the `consumedBy Relationship` attribute. For example, an `Analysis` object that uses this `Dataset` object as input needs to know if the actual dataset exists. When the `notifyDependents` method executes, the `Dataset` object prompts each `Analysis` object consuming it to check its processing state.

A `Dataset` object's readiness can be checked by calling the `isReady` method (line 29). A `Dataset` object is considered ready if the *producedBy* relationship is complete. The `producedBy` relationship is complete if there is exactly one `Analysis` object referenced by the `producedBy` attribute and that object has not begun its processing.

In class `Dataset`, the `process` method (line 30) calls a new method, `create` (line 31). This aliasing is done because it is more semantically correct to speak of *creating* a dataset (instead of *processing* one) when it is being generated from scratch. A call to the `create` method invokes the dataset's producer to generate it. This is done by calling the `process` (or `submit`) method of the `Analyis` object referenced by its `producedBy` relationship attribute.

4.2.4 Class `AnalysisStrategy`

Often, multiple computational programs are used together to perform a computational experiment. The set of executions of these programs and any other details, such as rules that govern the order in which these executions are carried out, can be viewed as one entity. In essence, the programs that are used and the order in which they are executed define an *analysis strategy*. As a result, there is an implied relationship between an analysis strategy and the programs used to perform the computation. One analysis strategy involves multiple component program executions. Alternatively, each program execution may be a component in (at most) one analysis strategy. Figure 4.3 illustrates the relationship between `Analysis` objects and `AnalysisStrategy` objects.

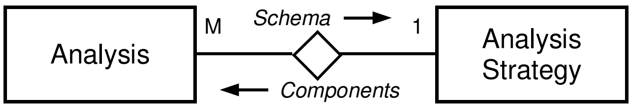

Figure 4.3. Relationship between `Analysis` and `AnalysisStrategy` objects.

As a result, the `Analysis` class includes a third `Relationship` member named *schema* (Table 4.1, line 4) to reference the analysis strategy it participates in (if any). Of course, the `AnalysisSchema` class implements the `Analysis` side of this relationship.

In addition, the `AnalysisStrategy` was defined as the third base class for this application. `AnalysisStrategy` class instances represent the co-dependent instances of program executions in a single, complete computational process. The `AnalysisStrategy` class defines a `Relationship` attribute named `components` (see Table 4.2 line 2) that references the `Analysis` objects that are the components of the analysis strategy. The `AnalysisStrategyComponents` class implements the `AnalysisStrategy` side the the relationship and is instantiated on line 4.

Table 4.2. Partial definitions of the `Analysis` and `Dataset` classes.

```
1  public class AnalysisStrategy extends JSEFObject {

2    public Relationship components;
      // other attribute definitions

3    protected void initialize() {
4      components = new AnalysisStrategyComponents(this);
        ...
     }

5    protected abstract void componentInit() { ... }

6    public boolean isReady() { ... }
7    public Object process() { ... }
8    public void submit() { ... }

9    public Relationship getInput() { ... }
10   public Relationship getOutput() { ... }
11   public Relationship getComponents() { ... }
  }
```

The purpose of the `componentInit` method (line 5) is to create new component `Analysis` objects, new `Dataset` objects (which are not components), and relate these objects to each other in the appropriate manner to correspond to the flow of the computation encapsulated by the `AnalysisStrategy` object. The `componentInit` method is called by the `AnalysisStrategy` constructor. As a result, when an `AnalysisStrategy` object is created, most, if not all, of the entire experiment is set up and ready to be carried out.

The readiness of an AnalysisStrategy object can be checked by calling the isReady method (line 6). As mentioned previously, an AnalysisStrategy object may be ready as soon as its construction is complete. In particular, the completeness of the *components* Relationship attribute is all but guaranteed by the *componentInit* method since this method ensures that the correct number of component Analysis objects are created, that they are of the correct type, and that they have not been processed (since they are new objects).

However, an AnalysisStrategy object may not be ready if, for example, the user needs to specify one or more Dataset objects to be used as input to the first component of the strategy. Consequently, we defined the getInput method (line 9) to return a reference to the input Relationship attribute of the first Analysis component. Similarly, the getOutput method (line 10) returns a reference to the output Relationship attribute of the final Analysis component. In addition, the user can access the components of the AnalysisStrategy with the getComponents method (line 11).

When it is time for the computation to take place, users call the process method (line 7). The default behavior for invoking a computation via an AnalysisStrategy object is to submit its components to an external execution engine for processing. Like the Analysis class, the AnalysisStrategy class defines the submit method (line 8) as an alias to the process method. In other words, a call to process simply invokes the submit method.

When the submit method is called, the AnalysisStrategy object calls the submit method of each component Analysis object. In short, making an AnalysisStrategy object available for external processing is equivalent to making its components available for external processing. This illustrates the fact that an AnalysisStrategy object acts as a passive organizer object whose computational action is defined by the actions of its components.

4.3 Schema Refinement

From the perspective of the computation, the `Analysis`, `Dataset`, and `AnalysisStrategy` classes are basic since they do not refer to specific resources. As a result, they are defined as abstract classes. However, they have been defined with a large amount of functionality. Consequently, during schema refinement, while these classes are extended to derive new classes to represent specific computational resources depicted in the schema, much of the refiner's work has already been done. The refiner's job is often as simple as defining new constraints on the relationships between objects of the newly-extended classes.

4.3.1 Base class extensions

We extended several classes from the previously-created base classes using functionality supported by the `JSEF.autocode` package described in Section 3.7. In particular, we have extended three subclasses from the `Analysis` class to correspond to our computation: `EventGeneration`, `Simulation`, and `Reconstruction`. Instances of the `EventGeneration` class represent distinct executions of the program that generates simulated electron-target collision events. `Simulation` objects represent distinct executions of the program that takes event data and generates "raw" data that simulates detector responses. Objects of the `Reconstruction` class represent executions of the program that accept raw data and generates event data that describes the (simulated) event that caused the (simulated) detector response stored in the raw data file.

We show an example of how we extended the `Analysis` class by describing the steps involved in creating class `EventGeneration` with the prototype code generator included in class `JSEF.autocode.Encoder` discussed in Section 3.7. First, we indicate that we want to create a new class. Table 4.3 shows the interface interaction involved to create the `EventGeneration` class.

Table 4.3. Creating a new class with the text-based code generator.

```
Encoder Main Menu

[1] Create New Class
[2] Quit
Your choice: 1
Enter a name for the new class: EventGeneration
Enter the class being extended/copied (Default: JSEFObject): Analysis
(E)xtend or (C)opy "JSEF.Analysis"?: e
Is class EventGeneration "public" (y/n)?: y
Is class EventGeneration (a)bstract, (f)inal, or (n)either?: n

EventGeneration Class Definition
// Menu options here...
```

At this point, a new `JSEFClassDef` object is created for the new class, `EventGeneration`. Next, we specify other class details. Options 1–4 listed in Table 4.4 shows which class details can be specified. The order in which one specifies class details does not matter. For example, a new attribute (Option 2) can be added before an interface is added. The `EventGeneration` class requires no additional attributes, so all that is left for us to do is specify relationship constraints. We illustrate this process in Section 4.3.2.

Table 4.4. Menu options for specifying class details.

```
EventGeneration Class Definition

[1] Add an Interface
[2] Add a New Attribute
[3] Add a New Relationship
[4] Constraint an Inherited Relationship
[5] List Class Elements
[6] Generate Code
[7] Return to Main
Your choice:
```

We have also extended two `Dataset` subclasses classes: `EventData` and `RawData`. `EventData` objects represent files that contain event data. Similarly, `RawData` objects represent files that contain raw data. The process of specifying these new classes is similar to that shown in Table 4.3 with appropriate changes to the new class name and the name of the class being extended.

Finally, we have extended one `AnalysisStrategy` subclass, `BarrowStrategy`. This class is named for Dr. Stephen Barrow, a researcher at Jefferson Lab who routinely performs this experiment. `BarrowStrategy` objects represent one execution of each of the following programs in succession: the event generator program, the detector response simulation program, and the event reconstruction program. Again, the process of specifying this class is similar to the process shown in Table 4.3. Figure 4.4 shows the hierarchy of these newly-extended classes.

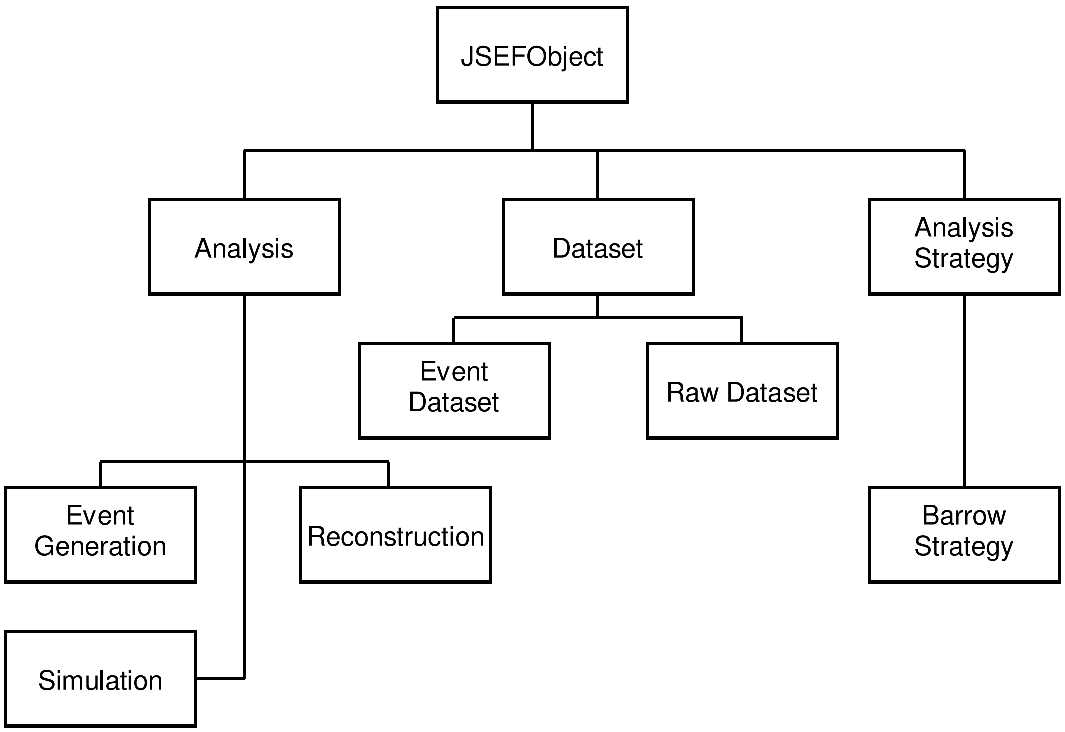

Figure 4.4. Hierarchy of base and refined classes.

4.3.2 Relationship refinements

The new classes refine the relationships they inherited from their parent classes to correspond to the flow of the experiment as depicted in Figure 4.1. Specifically, an `EventGeneration` object has no input and produces one `EventData` object as output. We specify these relationship constraints by selecting Option 4 of the menu shown in Table 4.4. Table 4.5 illustrates the interaction involved in refining the `input` and `output` relationships of the new `EventGeneration` class using the `Encoder` code generator.

Table 4.5. Steps involved in relationship refinement.

```
// EventGeneration Class Definition menu here...
// Option 4--Constrain an Inherited Relationship
Your choice: 4
Enter existing relationship name: input
Enter name of the related class: Dataset
Enter attribute name in class Dataset that
        implements the inverse relationship: consumedBy
Enter the maximum cardinality: 0
Enter the minimum cardinality: 0

// EventGeneration Class Definition menu here...
// Option 4--Constrain an Inherited Relationship
Your choice: 4
Enter existing relationship name: output
Enter name of the related class: EventData
Enter attribute name in class EventData that
        implements the inverse relationship: producedBy
Enter the maximum cardinality: 1
Enter the minimum cardinality: 1
```

The relationships of the other `Analysis` subclasses are also refined. A `Simulation` object uses one `EventData` object as input and produces one `RawData` object as output. Finally, a `Reconstruction` object accepts one `RawData` object as input and produce one `EventData` object as output. These refinements are specified using the `Encoder` code generator in the manner shown in Table 4.5.

The refined `Dataset` objects also include relationship refinements. In particular, an `EventData` object is produced by one `Analysis` object and can be used by any number of `Analysis` objects as input. On the other hand, while a `RawData` object, like an `EventData` object, can be used as input by any `Analysis` object, it can only be produced by one `Simulation` object. Again, the relationship refinements for `Dataset` subclasses are specified using the `Encoder` code generator according to the process shown in Table 4.5.

Of course, there are relationship refinements between `BarrowStrategy` objects and its component `Analysis` objects. Particularly, a `BarrowStrategy` object must have exactly three components: one `EventGeneration` object, one `Simulation` object, and one `Reconstruction` object. However, these `Analysis` objects can be involved in any type of analysis strategy. In the context of `AnalysisStrategy` subclasses, these data type specifications are not constraints. Instead, they indicate the type of components to be created when the components of an `AnalysisStrategy` object are initialized. The the functionality provided by the `componentInit` method is discussed in Section 4.3.3. Figure 4.5 illustrates the relationship refinements of the all the refined subclasses.

When a class definition is completed, we invoke the generation of Java code that corresponds to the class definition by selecting Option 6 from the menu in Table 4.4. The Java code includes class definitions to implement relationship refinements. For example, class `EventGenerationInput` enforces a zero-element relationship with `Dataset` objects since `EventGeneration` objects take no input. At the same time, class `EventGenerationOutput` enforces an `EventData` class data type constraint on the `EventGeneration` output. Table 4.6 shows the automatically-generated code for the `EventGeneration` class.

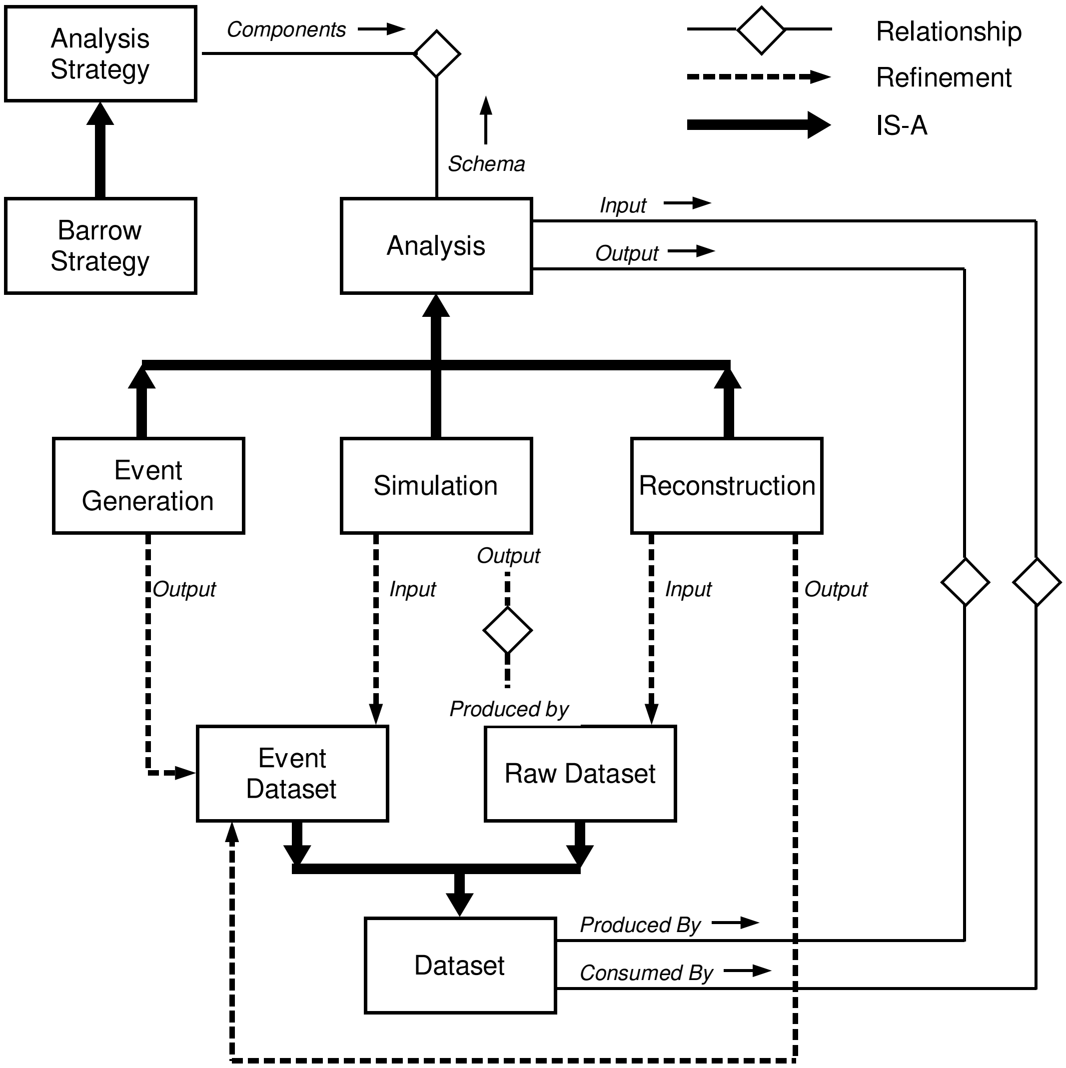

Figure 4.5. Relationships between refined JSEF classes.

4.3.3 Additional functionality of refined classes

As we mentioned previously, much of the work of the schema refiner is done by the schema designer and by an automatic code generator. Case in point, `Dataset` subclasses do not require any additional functionality since datasets simply exist or can be created. The name of a `Dataset` subclass is used to identify the type of data stored in a file referenced by a `Dataset` object.

Table 4.6. Partial listing of automatically-generated code.

```
public class EventGeneration extends JSEF.Analysis {

  protected void initialize() {
    super.initialize();
    input  = new EventGenerationInput(this);
    output = new EventGenerationOutput(this);
  }
}

class EventGenerationInput extends JSEF.AnalysisInput {
  public EventGenerationInput(EventGeneration theParent) {
    super(theParent);
  }

  public boolean addElement(JSEFObject newElem) {
    return (super.addElement(newElem));  // Enforce unch. inherited
  }                                      // data type constraint.

  public boolean isComplete() {
    // Enforce cardinality constraints here!
    if (super.getCount > 0)           // No inputs required!
      return (false);
    else
      return (super.isComplete());    // Consider parent constraints.
  }
}
```

However, some effort on the part of the refiner is required. For example, the `getExecutable` method must be defined in each `Analysis` subclass to return the name of the executable program that corresponds to it. For example, all `EventGeneration` objects return the name of the event-generating program, all `Simulation` objects return the name of the detector simulation program, and all `Reconstructor` objects return the name of the event-reconstruction program. Also, the constructor of each `Analysis` subclass also includes the designation of any default parameter–value pairs used by its corresponding executable program to its `ParameterSet` attribute.

In the role of schema refiner, we manually coded the implementation of the `componentInit` method of class `BarrowStrategy`. When this method is called, the flow of this computation (as illustrated in Figure 4.1) is set up in the state of the `BarrowStrategy` object. In particular, the components are created: an `EventGeneration` object, a `Simulation` object, and a `Reconstruction` object. In addition, two `EventData` objects and one `RawData` object are created.

These objects are then related to one another. One `EventData` object is referenced both as the output from the `EventGeneration` component and as input to the `Simulation` component. The `RawData` object is referenced both as output from the `Simulation` component and as input to the `Reconstruction` component. Finally, the remaining `EventData` object is referenced as output from the `Reconstruction` component.

The `componentInit` method also employs a rudimentary naming scheme among the components and their corresponding `Dataset` objects. For instance, if a `BarrowStrategy` object was named "run1," each object would have "run1" designated as part of its name. Table 4.7 shows the code for the `componentInit` method.

4.4 Data Analysis

Data analyzers instantiate the classes defined during schema refinement in order to carry out the this physics experiment. As objects are instantiated, they are stored in a database. Thus, information about the computation such as the time it took place and which computational entities were involved, can be obtained by querying the database for those objects and examining their state information.

Table 4.7. Code for the `componentInit` method.

```
protected void componentInit() throws JSEFException {
  String runNumr = getName();                 // Get run number.

  System.out.println("Preparing components for BS #" + runNumber);

  // Create the schema components and pass
  // the run number to those components.
  EventGen      eventGen = new EventGen(runNum);
  GSim          gSim     = new GSim(runNum);
  Reconstructor re       = new Reconstructor(runNum);

  System.out.println("Constructed components for BS #" + runNum);

  components.addElement(eventGen);            // Add components to
  components.addElement(gSim);                // the components
  components.addElement(re);                  // relationship.

  // Create input and output objects to relate to
  // the Analysis components.
  EventData eventGenOutput = new EventData("eventGenOutput" + runNum);
  RawData   gSimOutput     = new RawData("gSim output" + runNum);
  EventData reOutput       = new EventData("recon output" + runNum);

  if ((! eventGen.output.addElement(eventGenOutput))  // Now, relate
   || (! gSim.input.addElement(eventGenOutput))       // components
   || (! gSim.output.addElement(gSimOutput))          // together.
   || (! re.input.addElement(gSimOutput))
   || (! re.output.addElement(reOutput))) {
    throw new JSEFException("component relationship failure");
  }
}
```

4.4.1 Creating remote JSEF objects

In Section 3.6, we discussed how remote interfaces, wrapper objects and a remote server function to support remote access to JSEF objects. We created several interfaces and wrapper classes to provide remote access to the objects developed to support this computational process. In particular, class `Analysis` is supported by the `rAnalysis` interface and the `AnalysisWrapper` class. The `rDataset` interface and the `DatasetWrapper` class provide remote access to `Dataset` objects. Finally, remote access to `AnalysisStrategy` objects is made possible through the `rAnalysisStrategy` interface and the `AnalysisStrategyWrapper` class.

Although we derived several subclasses from the base classes, all of the functionality required by the computation is defined in the base classes. This is also true in the case of our implementations of the `Relationship` and `DBManager` classes. Consequently, it was not necessary to create an interface-wrapper class pair for every class we created for this computation.

We defined the `JSEFServer` interface to have methods to create instances of the base classes in this computation. In fact, since the base classes are abstract, the implementation of the *create* methods must actually instantiate the subclasses. Thus, the *create* methods require at least one parameter, the name of the subclass to instantiate. Methods that create `JSEFObject` subclass objects also require a parameter value to use as the object's name. The method that creates `DBManager` subclass objects requires a parameter that specifies the database URL.

The `JSEFServiceObject` class implements the `JSEFServer` interface. The method calls `createDataset` and `createAnalysis` depicted in Table 3.4 show how the *create* methods are used. For example, when the `createDataset` method is called with parameters "RawData" and "rawData1," the `JSEFServiceObject` instantiates a `RawData` object and names it *rawData1*. Next, that object is wrapped

Table 4.8. Pseudo-code of a JSEF client creating new objects for processing.

```
class BarrowClient {
  public static void main(String args[]) {
    JSEFServer server = Naming.lookup("//www.myhost.edu/JSEFServer");
    rDBManager dbm = server.createDBManager("FileManager", "JSEF.db");

    dbm.open();
    dbm.beginTransaction();

    for (int i = 1; i < 10; i++) {
      rAnalysisStrategy as = server.createAnalysisStrategy(
                                    "BarrowStrategy", "run" + i);
      as.submit();
    }

    dbm.commitTransaction();
    dbm.close();
  }
}
```

by a `DatasetWrapper` object (which implements the `rDataset` interface), and this wrapper object is returned to the remote client.

4.4.2 Creation of `BarrowStrategy` objects

The first step in using JSEF to carry out this experiment is to create a remote client that will create and submit multiple `BarrowStrategy` objects. A simple client would look up and acquire the remote JSEF server, then create and submit `BarrowStrategy` objects within a loop. Note that when one `BarrowStrategy` object is created, multiple component `Analysis` and `Dataset` objects and the `Relationship` objects linking these objects together are also created. Table 4.8 shows a pseudo-code example of a simple client.

As we can see from the pseudo-code, the `BarrowStrategy` objects are named according to a simple naming strategy. In addition, since the objects are created in the context of a database transaction, they persist in the database after the client terminates.

4.4.3 External processing of JSEF objects

Once the computational objects have been created and are pending, an *execution client* submits them to an execution environment. More precisely, The execution client obtains a reference to the `ExecutionManager` object with which it will retrieve `Analysis` objects that are pending. Each pending `Analysis` object contains information that is used to invoke its corresponding computational program. The execution client formats this information and submits it to the execution environment. Table 4.9 shows an example of an execution client that processes the next available `Analysis` object.

The execution environment and/or other thin clients update the status of the objects as their processing takes places. Data analyzers can query the database to examine the status of these objects while they are processing and when the processing terminates.

4.5 Summary

We have discussed a multi-step computational physics experiement and shown how we have applied the JSEF approach. In the *schema design* stage, we developed classes that represented basic entities of the experiment and created relationships between them. Next, we *refined* those basic classes to represent specific computational entities. We also discussed the development of classes to support remote access to JSEF objects and showed how *data analyzers* use those classes to carry out the computational process.

Table 4.9. Pseudo-code of a JSEF execution client.

```
class ExecutionClient {
  public static void main(String args[]) {
    JSEFServer server = Naming.lookup("//www.myhost.edu/JSEFServer");
    rDBManager dbm = server.createDBManager("FileManager", "JSEF.db");

    dbm.open();
    dbm.beginTransaction();

    ExecutionManager execMgr = server.getExecutionManager();
    rAnalysis anal = execMgr.getNext();

    // Organize the state info from analysis object into
    // a format acceptable to the execution environment.
    String execCmd = formatInfo(anal.process());

    // Invoke the computational action
    // in the execution environment.
    submitCmd(execCmd);

    dbm.commitTransaction();
    dbm.close();
  }
}
```

In the next chapter, we summarize this work, illustrating its usefulness and its contributions to the fields of computer science and computational science. We also suggest next steps with regard to work in this area.

CHAPTER 5

DISCUSSION

5.1 Summary

Scientists who use several computer programs to produce and/or analyze scientific data face several challenges. They must schedule the execution of the programs. Also, they must keep track of the voluminous data being generated and/or analyzed. In addition, they must record the steps involved in and the results of the computational process for later examination. Each of these responsibilites becomes more difficult as the number of programs, datasets, computations, scientists involved, or the length of time the computation has been utilized increases. However, the current trend has been toward developing faster, higher capacity, distributed systems to be used my many scientists.

We have produced a Java-based Schema Extension Facility (JSEF) to assist scientists involved in the process of computational discovery. We have used Java's object-oriented data model as a basis for classifying computational entities. Also, JSEF's relationship mechanism allows us to create relationships between classes. Scientists can use JSEF objects to manage their computational processes by specifying JSEF classes to represent each entity in the computation and defining relationships between each class. In fact, JSEF allows a series of objects representing co-dependent programs to be classified as a single entity called an *analysis strategy.* In addition, JSEF object persistence allows them to be queried and examined, and it allows them to serve as metadata references to entities of a computation.

In order to drive the computation, the *processing* method is called on a JSEF object, which invokes the computational action(s) associated with that object, or the object may be *submitted* to an execution manager for external processing. Thus, JSEF reduces the tasks involved in invoking and tracking a computation and the entities involved to a single method call, subsequent querying of the database, and an examination of the state of the JSEF objects in the query result.

We have shown the viability of the JSEF approach by applying it to a computational physics experiment that detects and analyzes the results of sub-atomic particle collisions. For this computation, we developed several JSEF classes to correspond to each of the computational programs and types of datasets involved. We also defined classes to implement the relationships between those entities and classes that made it possible to access class instances remotely. In addition, we developed remote clients; one client submitted JSEF objects for external processing, and the other client used information from these objects to perform the external processing of individual objects.

5.2 Contributions

5.2.1 Computer Science

As a result of this research, we have a tool for performing schema extention that readily extends to rising technologies, such as distributed computing using CORBA, web-based computing, and metacomputing. In fact, it employs a distributed computing technology, Java Remote Method Invocation (RMI). Additionally, the JSEF paradigm, which includes relationships and constraints, offers an extension to the ODBC standard which has yet to include specification for modeling relationships in Java.

We have also developed a persistent relationship implementation for Java objects. In addition, we have advanced the concept of relationship object having functionality, including support for the notions of relationship constraints and completeness.

5.2.2 Computational Science

In the computational science community, JSEF allows scientists to capitalize on the intuitiveness of the object-oriented paradigm. They are able to set up and manage experiments and share information easily. For example, (JSEF) object references to real-world entities allow the flexibility of having multiple objects of varying types refer to the same entity. Also, while serving as metadata, JSEF objects add context information to an entity. This context may vary from researcher to researcher, allowing scientists to view the same information from different perspectives. Further, employing an object-based paradigm to describe real-world entities lends itself to other high-level organization techniques, such as high-dimensional indexing and cluster analysis.

Object typing in JSEF as a result of inheritance allows JSEF data type constraints on relationships to maintain the integrity of computations by preventing incompatible objects from being related. Further, object types can be used as the basis of an OQL search in which sets of objects can be returned, new objects (e.g. sets) can be created. In fact, OQL queries can be used to execute methods on existing objects.

Another significant feature of JSEF is that it relies on method calls to manage computational entities. In addition, while JSEF objects describe computational processes, the JSEF paradigm separates the description from the processing itself. As a result, a data analyzer can create an arbitrarily large number of objects that describe several instances of the complex computation at one time. For example, a JSEF client might be programmed to instantiate and submit 1000 `AnalysisStrategy` objects at one time (of course, each `AnalysisStrategy` might contain several components). The time required to instantiate and submit those objects is minimal, and it would remain relatively constant compared to the time required to perform the actual computations, as the number of objects/computations to be submitted increased.

The execution of each instance of the computation can be performed at another time, even by another researcher. In fact, an execution client can be used to automate the execution process, and the execution engine may process several activities simultaneously, making the process even more efficient.

Another concept that arises from JSEF is *computational equivalence*. Although scientists run several programs and analyze several datasets, it is the result of the last step that is important. In other words, a scientist might ideally want to ask the system, "Give me the dataset that results from the last step of the computation," without necessarily having to worry about the details of setting up the entire computation. The `AnalysisStrategy` class makes this ideal feasible. In short, the creation of an `AnalysisStrategy` instance takes care of setting up the computational details. When the scientist calls the `submit()` method on an `AnalysisStrategy` instance, the ultimate result of that call (e.g. some dataset(s)) is precisely what the scientist is looking for.

5.3 Future Work

For the purposes of this dissertation, development of the necessary JSEF classes was done manually. However, in order to make JSEF an enterprise application, graphical user interfaces (GUIs) must be designed. Schema designers and schema refiners would benefit from using GUIs to define JSEF classes, create relationships, and invoke computational actions in a more intuitive manner. In fact, we have already developed the `JSEF.autocode` package with GUIs in mind.

For instance, a GUI tool for JSEF class development would use the `JSEF.autocode` classes to capture JSEF class definitions specified by the user and turn those definitions into source code that adheres to JSEF principles. Also, a GUI tool could be developed to act as a dynamic JSEF submission and execution client, eliminating the need for hard-coded clients like the examples shown in Tables 4.8 and 4.9. In addition, servlet- and applet-based tools would allow scientists to develop JSEF classes and manage their computations with JSEF objects using web browsers.

Also, it may be a worthwhile endeavor to improve the server-side of JSEF by including enterprise-level support for JSEF objects. For example, incorporating Java IDL would allow JSEF objects to be integrated into a distributed, CORBA-based system. Similarly, the `JSEFServer` interface and the `JSEF.rmi` package could be implemented using Enterprise Java Beans for more complex concurrency control, among other things. A JSEF object query engine might also be useful, allowing OQL queries to be performed on a relational database.

In addition, we may want to revisit the `AnalysisStrategy` class in order to improve its ability to handle complex computational strategies. Unfortunately, in its current form, the `AnalysisStrategy` class is not well-suited to be easily extended with the automatic code generator. Consequently, schema refiners are required to write code to extend `AnalysisStrategy` subclasses. However, by redefining the `AnalysisStrategy` class and defining a complementary set of semantics, the extension of the `AnalysisStrategy` class could be automated. It might be possible, for example, for an `AnalysisStrategy` object to re-create an existing computational strategy at runtime by supplying appropriate command arguments.

APPENDIX A

JAVADOCS FOR JSEF PACKAGES AND SELECTED CLASSES

The Java programming language includes a special commenting style called *javadocs*. These comments are delimited by `/**` and `*/` and may include tags that are preceded by the @ sign. The contents of javadoc comments may be used in automatically generated HTML-based documentation by the `javadoc` tool. The `javadoc` tool is included in the JDK bundle support HTML-based source code documentation.

The javadocs generated from the JSEF packages and selected classes are included in this appendix.

REFERENCES

[AIL98] Anastassia Ailamaki, Yannis Ioannidis, and Miron Livny. Scientific workflow management by database management. In *Proceedings of the 10th International Conference on Scientific and Statistical Database Management*, Capri, Italy, July 1998.

[BSS+95] Donald J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak, and Charles V. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings of the 1995 International Conference on Parallel Processing*, August 1995. Available at http://www.beowulf.org/.

[C+97] S. Conrad et al. Research issues in federated database systems. *SIGMOD Record*, 26(4):54–56, December 1997.

[CB97] Rick G. G. Cattell and Douglas K. Barry, editors. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, San Francisco, 1997.

[CKMS97] I-Min A. Chen, Anthony S. Kosky, Victor M. Markowitz, and Ernest Szeto. Contructing and maintaining scientific database views in the framework of the Object-Protocol Model. In *Ninth International Conference on Scientific and Statistical Database Management*, pages 237–248, August 1997.

[CM95] I-Min A. Chen and Victor M. Markowitz. An overview of the object protocol model (OPM) and OPM data management tools. *Information Systems*, 20(5):393–418, July 1995.

[CMFD94] Judith B. Cushing, David Maier, David Feller, and Michael DeVaney. Computational proxies: Modeling scientific applications in object databases. In *Seventh International Working Conference on Scientific and Statistical Database Management*, pages 196–206, Charlottesville, September 1994.

[CO92] Stephen Cannan and Gerard Otten. *SQL—The Standard Handbook.* McGraw Hill, Berkshire, November 1992.

[Cra89] Walt Crawford. *MARC for Library Use: Understanding integrated USMARC*. G. K. Hall and Company, Boston, second edition, 1989.

[Eav97] Samuel J. Eaves, II. A "lite" paradigm for spatial information access. Master's thesis, The Florida State University, Tallahassee, Florida, 1997.

[EDR93] Bryon K. Ehlmann, Lawrence C. Dennis, and Gregory A. Riccardi. An object-based conceptual model of a nuclear physics experiments database. *Nuclear Instruments and Methods in Physics Research*, pages 294–308, 1993.

[FGD94] Federal Geographic Data Committee FGDC. Guidelines for implementing the National Geospatial Data Clearinghouse (guidelines for federal agencies). Available at http://www.fgdc.gov/clearover2.html, 1994.

[FGD98] Federal Geographic Data Committee FGDC. Metadata version 2.0 released. Federal Geographic Data Committee Newsletter, Summer 1998. Available at http://www.fgdc.com.

[FK97] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal for Supercomputing Applications*, 11(2):115–128, 1997. Available at http://www-fp.globus.org/-documentation/papers.html.

[For95] Charles Wesley Ford, Jr. *Modelling Experimental Data Analysis*. PhD thesis, The Florida State University, Tallahassee, Florida, 1995.

[Fre91] James C. French. Support for scientific database management. In Zbigniew Machalewicz, editor, *Statistical and Scientific Databases*, pages 51–82. Ellis Horwood Limited, New York, 1991.

[GFH99] Andrew Grimshaw, Adam Ferrari, and Marty Humphrey. Wide-area computing: Resource sharing on a large scale. *Computer*, 32(5):29–37, May 1999.

[GFLH98] Andrew Grimshaw, Adam Ferrari, Greg Lindahl, and Katherine Holcomb. Metasystems. *Communications of the ACM*, 41(11):46–55, November 1998.

[GHR97] Ashish Gupta, Venky Harinarayan, and Anand Rajaraman. Virtual database technology. *SIGMOD Record*, 26(4):57–61, December 1997.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1996.

[GW+97] Andrew Grimshaw, William A. Wulf, et al. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1), January 1997. Available at http://www.cs.virginia.edu/~legion.

[Has99] Ken Hass. Where the rubber meets the road. *Computing in Science and Engineering*, 1(1):4–6, January–February 1999.

[HCF97] Graham Hamilton, Rick Cattell, and Maydene Fisher. *JDBC Database Access with Java.* The Java Series. Addison-Wesley, Reading, MA, 1997.

[HQGW93] Nabil I. Hachem, Ke Qiu, Michael Gennert, and Matthew Ward. Managing derived data in the Gaea scientific DBMS. In *Proceedings of the 19th VLDB Conference*, pages 1–12, Dublin, August 1993.

[ILGP96] Yannis E. Ioannidis, Miron Livny, Shivani Gupta, and Nagavamsi Ponnekanti. ZOO: A desktop experiment management environment. In *Proceedings of the 22nd VLDB Conference*, Bombay, 1996.

[Joh98] A. S. Johnson. Java analysis studio. In *Proceedings of the International Conference on Computing in High-Energy Physics*, Chicago, 1998.

[KA90] Setrag Khoshafian and Razmik Abnous. *Object Orientation: Concepts, Languages, Databases, User Interfaces.* John Wiley and Sons, New York, 1990.

[KG98] Sidney Karin and Susan Graham. The high-performance computing continuum. *Communications of the ACM*, 41(11):32–35, November 1998.

[LOC] Library of Congress LOC. MARC standards. Available at http://www.loc.gov/marc/marc.html.

[MH94] David Maier and David Hansen. Bambi meets Godzilla: Object databases for scientific computing. In *Seventh International Working Conference on Scientific and Statistical Database Management*, pages 176–184, Charlottesville, September 1994.

[MMS+96] Edmond Mesrobian, Richard Muntz, Eddie Shek, Silvia Nittel, Marc Larouche, Mark Kriguer, and Frank Fabbrocino. OASIS: An EODSIS science computing facility. In *International Symposium on Optical Science, Engineering, and Instrumentation (SPIE), Conference on Earth Observing System*, pages 284–298, November 1996.

[NMM97] Sylvia Nittel, Richard R. Muntz, and Edmond Mesrobian. geoPOM: A heterogenous geoscientific persistent object system. In *Ninth International Conference on Scientific and Statistical Database Management*, pages 252–263, August 1997.

[OMG95] Object Management Group OMG. *CORBA: Common Object Request Broker Architecture and Specification.* Object Management Group, Framingham, 1995.

[Owe93] J. Owen. *STEP – An Introduction.* Information Geometers, Winchester, 1993.

[Pre94] U. S. Office of the President. Coordinating geographic data acquisition and access: The National Spatial Data Infrastructure. Executive Order 12906, April 1994.

[RBMS97] Daniel Ridge, Donald Becker, Phillip Merkey, and Thomas Sterling. Beowulf: Harnessing the power of parallelism in a Pile-Of-PCs. In *IEEE Aerospace Conference*, February 1997. Available at http://www.beowulf.org/.

[RD94] Barbara Rieche and Klaus R. Dittrich. A federated DBMS-based environment for molecular biology. In *Seventh International Conference on Scientific and Statistical Database Management*, pages 118–127, September 1994.

[Sie98] Jon Siegel. OMG overview: CORBA and the OMA in enterprise computing. *Communications of the ACM*, 41(10), October 1998.

[SS95] Richard M. Soley and Christopher M. Stone, editors. *Object Management Architecture Guide.* John Wiley and Sons, New York, third edition, 1995.

[Suna] Java IDL. Available at http://java.sun.com/products/jdk/1.2/docs/-guide/idl/index.html.

[Sunb] Java Remote Method Invocation. Available at http://java.sun.com/-products/jdk/1.2/rmi/index.html.

[Sun98a] Java Blend: Integrating Java objects with enterprise data, 1998. Available at http://java.sun.com/marketing/collateral/javablend.html.

[Sun98b] Java for the enterprise, 1998. Available at http://java.sun.com/-marketing/enterprise/enterprise.html.

[Tho97] Anne Thomas. Enterprise JavaBeans: Server component model for Java. Technical report, Sun Microsystems, Inc., December 1997. Available at http://java.sun.com/products/ejb/white_paper.html.

[Vil99] Juan Villacis. A note on the use of Java in scientific computing. *Applied Computing Review*, 7(1):14–17, 1999.

[W3C] World Wide Web Consortium W3C. HTML 4.0 specification. Available at http://www.w3.org/TR/REC-html40.

[Web97] Joe Weber. *Using Java 1.1.* Que, third edition, 1997. Special Edition.

[WH98] Stuart Weibel and Juha Hakala. DC-5: The Helsinki metadata workshop: A report on the workshop and subsequent developments. *d-Lib Magazine*, February 1998. Available at http://www.dlib.org/-dlib/february98/02weibel.html.

[WI93] Janet L. Weiner and Yannis E. Ioannidis. A moose and a fox can aid scientists with data management problems. In *Proceedings of the 4th International Workshop on Database Programming Languages*, pages 376–398, New York, August 1993.

[Wil98] Roy Williams. Workshop on interfaces to scientific data archives. Available at http://www.cacr.caltech.edu/isda, March 1998.

BIOGRAPHICAL SKETCH

Samuel Joseph Eaves, II

Samuel Joseph Eaves, II was born to Samuel J. and Linda J. Eaves on December 18, 1972 in Jacksonville, Florida. Samuel attended the public schools of Duval County, and graduated as salutatorian of the William M. Raines High School class of 1990.

After high school, Samuel continued his education at the prestigious Morehouse College in Atlanta, Georgia, where he majored in computer science. As a Ronald E. NcNair Scholar, Samuel had the opportunity to spend the summers interning at the Kennedy Space Center in Florida. Also, while at Morehouse, Samuel received several honors, including Phi Beta Kappa (1993), Who's Who In America's Universities and Colleges (1992-1994), and the National Dean's List (1992-1994).

Presently, Samuel is a McKnight Doctoral Fellow attenting The Florida State University. While at Florida State, he has been honored by Upsilon Pi Epsilon Honor Society and been named an Outstanding Young Man of America for 1996. He earned the Master of Science degree in computer science in 1997. He is currently pursuing the Ph.D. degree in computer science.