Computational Proxies for Computational Chemistry: A Proof of Concept

Meenakshi Rao

A thesis submitted to the faculty of the
Oregon Graduate Institute of Science & Technology
in partial fulfillment of the requirement for the degree
Master of Science
in
Computer Science and Engineering
January 1996

The thesis "Computational Proxies for Computational Chemistry: A proof of Concept" by Meenakshi Rao has been examined and approved by the following Examination Committee:

DAVID MAIER
Professor
Thesis Advisor

STEVEN OTTO
Associate Professor

JONATHAN WALPOLE
Associate Professor

D. MICHAEL DEVANEY
Research Scientist
Batelle Pacific Northwest Laboratories

To -

Judy, who made the thesis fun,
Rahul, who made it necssary,
and Rohan, who made it (almost) impossible!

TABLE OF CONTENTS

LIST OF FIGURES v
ABSTRACT vi
1. INTRODUCTION 1
2. BACKGROUND 5
2.1 Object-Oriented Databases 5
2.2 Computational Chemistry 7
2.3 The Computational Chemistry Database Project 9
2.3.1 The Computational Chemistry Information Model 10
2.3.2 Populating the Database 15
3. COMPUTATIONAL PROXIES 17
3.1 Definition and Functional Specification 17
3.2 Functional Components of the Computational Proxy 20
3.3 Computational Proxies: The CCDB scenario 26
4. COMPUTATIONAL PROXY PROTOTYPE FOR GAMESS 28
4.1 GAMESS 28
4.2 Design of the GAMESS Proxy 33
4.3 Implementation of the GAMESS Proxy 36
4.3.1 The Class compProxy 37
4.3.2 The Data Control Functions 40
4.3.3 The Compute Monitor 43
4.4 Validation 46
4.4.1 Summary of Implementation 46
4.4.2 Validation 47
5. ANALYSIS 49
5.1 The Prototype Computational Proxy 49
5.1.1 Application Registration 50
5.1.2 The Data Control Functions 53
5.1.3 The Compute Monitor 54
5.2 The Computational Proxy and OODBMS 55
5.3 Extending the Prototype Proxy to other Computation Chemistry Applications 59
6. CONCLUSIONS 63
BIBLIOGRAPHY 67
APPENDIX A: THE CCDB SCHEMA 70
APPENDIX B: THE FORMATTING FUNCTIONS FOR GAMESS 82
APPENDIX C: CCDB QUERIES 85
APPENDIX D: SETTING UP THE COMPUTATIONAL ENVIRONMENT FOR GAMESS 93
APPENDIX E: A GENERAL INTRODUCTION TO BASIS SETS 98

List of Figures

FIGURE 1 FEATURES OF SCIENTIFIC DATABASES 4
FIGURE 2 THE COMPUTATIONAL CHEMISTRY INFORMATION MODEL 12
FIGURE 3.1 COMPUTATIONAL PROXY FUNCTIONS FOR MANAGING ONGOING EXPERIMENTS 19
FIGURE 3.2 APPLICATION REGISTRATION FUNCTIONS 21
FIGURE 3.3 APPLICATION INPUT FUNCTIONS NEEDED BY THE COMPUTATIONAL PROXY 22
FIGURE 3.4 MISSION CONTROL FUNCTIONS NEEDED BY THE COMPUTATIONAL PROXY 23
FIGURE 3.5 EXPERIMENT OUTPUT CAPTURE FUNCTIONS NEEDED BY THE COMPUTATIONAL PROXY 25
FIGURE 4.1 COMPUTATIONAL PROXY ARCHITECTURE 35
FIGURE 4.2 THE COMPUTE MONITOR COMMUNICATIONS 45
FIGURE 5 BASIS SET INSTANCE IMPLEMENTATION 56

ABSTRACT

Computational scientists typically generate data using a variety of experiments (application program executions) that model scientific phenomena. Additionally, accessory programs are often developed to analyze the data output from such experiments. Computational scientists are thus faced with the twin problems of data and experiment management. These problems are exacerbated by the fact that computational programs are developed independently over a long period of time, by different groups, on different platforms, and use different formats and data structures to represent the experimental data. A central data model developed for a specific scientific domain would mitigate the data management problem. However, a central data model does not directly address the problem of experiment management or the issue of migrating "legacy applications" to the central data model. Ideally, computational scientists are looking for an integrated solution to the data and experiment management problem that would also address the issues of interoperability and legacy applications.

Cushing and Maier have proposed a mechanism called a "computational proxy" to address the experiment management problem. A computational proxy is an encapsulated representation of the information required to run an experiment, such as data inputs, formats, platform dependencies and environmental settings. A computational proxy, in conjunction with a central data model, has the information required to manage data and experiments, as well as the flow of data between experiments, thus addressing the issue of interoperability. Because of the information it encapsulates, a proxy can bind either legacy or new applications to the central data model.

This thesis presents the results of a feasibility analysis of the proxy mechanism. A prototype proxy mechanism was implemented for an application program in the domain of computational chemistry, in a SUN/Unix networked environment, using the object-oriented database management system ObjectStore. We demonstrated that the prototype proxy adequately

performs the tasks of data input, mission control and data capture. The prototype implementation also provided feedback that helped in further refining the proxy architecture.

CHAPTER 1

Introduction

Computational scientists develop models that are computational in nature to represent the physical phenomena of interest to them. Computational models have been developed for a wide range of phenomena, such as stellar atmospheres, chemical reactions, weather systems and molecular structures. Computational models are generally complex mathematical representations that are not amenable to analytic solutions and need to be solved numerically. These numerical solutions are coded as computer programs and are used to make predictions, which generally trigger experiments that, in turn, provide feedback to refine the models. The suite of programs available to a computational scientist represents his laboratory, each computational program in the suite is analogous to an instrument in a lab and each run of a computational program constitutes an experiment. The inputs to and outputs of such computational programs are usually data in the form of formatted, ASCII files.

Frequently, a number of computational programs exist to model the same phenomenon within a given scientific domain. Each program may make slightly different assumptions, use different approximation techniques or be optimized for some subset of input parameter ranges. Additionally, accessory programs are often developed that essentially act as a front-end or back-end to the computational program(s), helping the scientist to either prepare the input parameters for the experiment or analyze the output from an experimental run. Since the computational programs and accessories are developed independently and at different times, by different research groups, they are often written in different languages and have different input and output formats. Consequently, an accessory program will invariably run with only a (small) subset of

computational programs and vice versa. It is the syntactic disparity of the input and output that inhibits the interoperability of these programs.

Computational scientists, in their research efforts, have to manage large quantities of complex data, while keeping track of which experiment generated the data, which data can be used in conjunction with which accessory programs and so forth, while simultaneously keeping track of the evolving input and output file formats of the suite of programs they use! A database implementing a central data model that encapsulates the relevant semantic information for the scientific domain would alleviate some of the computational scientists' interoperability and data-management problems, while providing the basic advantages of a database management system [13]:

- Data sharing and concurrency control: Output from semantically equivalent applications would be centrally available and concurrent updates controlled.
- Data abstraction: A scientist would need to be familiar only with the central data model, and would not have to be acquainted with the details of the physical representation of the data for each program.
- Data independence: A scientist would interact with the central data model, and be shielded from format changes in the database implementation. Programs developed with one version of the model would not have to change as the database evolves.
- Query and retrieval: The query facility generally provided with a database would allow a scientist to browse the database, without writing programs.

Scientists have not been tempted to use existing database technology despite the benefits above mainly because prevalent record-based database management systems do not address the needs of the scientific community [17]. Specifically, record-oriented formats are not well suited to represent commonly used scientific data types such as matrices or time-series, or the complex relationships that usually exist within scientific data. Object-oriented systems may be better suited to represent scientific data, since they provide a richer data model and features such as extensibility, encapsulation and inheritance. Early research efforts at the Scientific Database Group Laboratory investigated using object-oriented database technology to address the data management problems of the scientific community. As part of this effort, we built a data

repository using an object-oriented database management system to store the results of computational experiments. However, we soon realized that populating such a database was a non-trivial task. One solution to the problem of populating the database was to automate the process of data capture. Automatically capturing the data requires the database to be able to communicate with the computational programs generating the data. This raises the issue of legacy applications – how can the wealth of legacy application programs and analysis tools be interfaced with a such central database?

Cushing proposed [6,8] an abstract data type, dubbed a "computational proxy" to interface a scientific database to a computational program. A "computational proxy" would be the locus of experiment control in the experiment management infrastructure. A computational proxy would stand-in, within the database, for an active process that is running a scientific application. The proxy would provide the database user with a consistent view of different applications that may be executing on distributed processors. The proxy could be used for specifying computational experiments, generating input to experiments from the database, launching and monitoring experiments, and loading output from experiments into the database. In effect, a proxy and its related objects would model scientific programs and processes within a database.

The aim of this thesis is a feasibility analysis of the computational proxy. A prototype computational proxy has been implemented for an application program in the domain of computational chemistry, interfacing it to a computational chemistry database. We chose the domain of computational chemistry for the prototype implementation because it is representative of the scientific computational domains, exhibiting many of the features that characterize scientific databases. Figure 1 illustrates the characteristic features of scientific databases, and notes, on a scale of 1 to 3, (3 being most important) the importance of the feature to the domain of computational chemistry. [6, 17].

The implementation and feasibility analysis of the prototype proxy provided important feedback on the applicability of the proxy mechanism and object-oriented database systems to the domain of computational chemistry, as well as other scientific domains. Cushing, as a result

of this work, went on to identify components of the proxy mechanism that needed refinement and to determine the system, language and database services required to support the proxy concept.

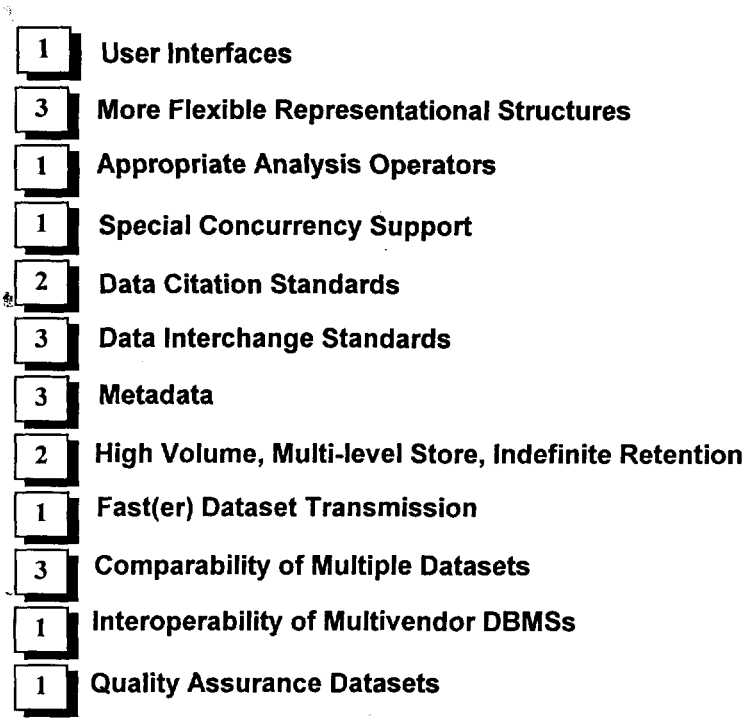

Figure 1: Features of Scientific Databases

The thesis is organized as follows: Chapter 2 presents a brief overview of object-oriented databases, the domain of computational chemistry and the computational chemistry database project. Chapter 3 presents the formal definition and functional requirements of the computational proxy. Chapter 4 discusses design decisions and implementation of the prototype computational proxy for a typical computational chemistry application program within the Computational Chemistry Database (CCDB). Chapters 5 and 6 present an analysis of the implementation process and our conclusions.

CHAPTER 2

Background

The goal of this thesis is a proof of concept of the computational proxy mechanism. A computational proxy would interface a domain database to a stand-alone domain application. The prototype computational proxy was implemented for the domain of *ab initio* computational chemistry, interfacing the Computational Chemistry Database (CCDB) to the ab initio computational chemistry application GAMESS. The CCDB was implemented using the object-oriented database management system ObjectStore, and the proxy implementation itself drew on the features of object-oriented databases. In Section 2.1 we present the main features of object-oriented databases. Section 2.2 offers an overview of the domain of computational chemistry, and in Section 2.3 we discuss the CCDB.

2.1 Object-Oriented Databases

Object-oriented databases constitute a new generation of databases, drawing on developments from the fields of artificial intelligence, programming languages and software engineering. They provide the traditional database features such as concurrency control, persistent storage management, transaction control and a query facility. It has been proposed that object-oriented database systems support the following features [2]:

- Complex objects: Complex objects can be constructed from simpler objects by using the appropriate constructors. The minimum set of constructors provided includes the set, list and

array constructors. The set of constructors is orthogonal to type and can be used on any user-defined types.

- Object Identity: Every object in the database has an identity independent of its value. Thus, it is possible for two objects in the database to have the same value but distinct identities.
- Encapsulation: Encapsulation is a mechanism for separating an interface for a data type from its implementation. Encapsulation promotes modularity and permits changing the implementation of a data type without affecting its behavior or any other object in the system.
- Types or Classes: Types or classes provide the data structuring mechanism in an object-oriented database. The database schema includes a set of types or classes. An object belonging to a particular class or type responds to all methods defined for that class or type.
- Class or Type hierarchies: Classes or types defining a database schema may be arranged in a hierarchy. A type or class inherits behavior from all types or classes above it in its type or class hierarchy. Inheritance provides both a useful modeling tool and a mechanism for filtering out shared specifications and implementations.
- Overriding, overloading and late-binding: It is sometimes convenient to use the same name for different operations. For example, we may call a set of operations "rotate", where the actual rotate operation used may vary based on the co-ordinate system used or the type of the object to be rotated. Distinguishing functions based on argument types is called overloading. Similarly, the operation "rotate" may be defined for a base class and subsequently modified by a derived class. Distinguishing functions based on the class they are invoked on is called overriding. Since the type of an object to be rotated may not be known until runtime, the correct "rotate" operation cannot be bound to the object until runtime. This runtime support for binding the appropriate function is called late-binding.
- Computational Completeness: Any computable function can be represented in the database manipulation language (DML).
- Extensibility: The user can add user-defined types to the system whose instances are subsequently treated as first class objects.

Some of the features of an object-oriented database are relevant to scientific applications. The features of extensibility, encapsulation, types and inheritance together provide a powerful and flexible modeling tool. Using these features scientists can represent scientific

data types such as matrices or time series, specialized for each scientific domain. These features also facilitate code reuse, thus encouraging the use of class libraries. Since implementation and behavior can be separated for a class, scientists are shielded (at least to some extent) from implementation changes, including changes in class libraries as they evolve. These features also provide the flexibility to store metadata information, that is information about the experiment, such as the *kind of data* – raw, calibrated or validated, and *source* of the data [17]. It is also possible to enforce a wider range of constraints and checks on the data in the database, by attaching the appropriate behavior to the database classes. The complex object constructors, such as the set constructor, permits a scientist to aggregate data in useful ways. For example, all experiments performed on methane could be aggregated into one set. The list constructor enables capture of order-related information, critical for representing scientific data such as time-series or DNA sequences.

Currently, several object-oriented database systems are available. These include commercial products such as GemStone from Servio Corp., ObjectStore from Object Design. Inc., Versant form Versant Technology and ITASCA from MCI. Also available are many research databases such as Observer/Encore [22], Trellis/Owl[1] [30] and Postgres [34]. These databases provide the features that characterize object-oriented database systems, but differ in their implementations, interface languages, platforms, querying facilities, level of user support and robustness. Many of them provide additional features such as multiple inheritance, configuration management and versioning, and database utilities such as schema evolution and database back-ups.

2.2 Computational Chemistry

Chemists working in the domain of computational chemistry try to predict the values of molecular properties based on computational models of the molecules. There are three main methodologies in computational chemistry – molecular mechanics, semi-empirical molecular orbital theory and ab initio quantum chemistry [9]. The first two are empirical methods, requiring

[1] Trellis/Owl is no longer available

some experimentally determined values as part of their input; the third methodology predicts values of molecular properties from first principles. These three methodologies overlap in their range of applicability. The Scientific Database Group has focused on the field of ab initio computational chemistry.

There exists a rich legacy of ab initio application programs – called "codes" by the computational chemists – such as GAMESS [11], GAUSSIAN [18] and HONDO [12]. Computational chemists also use a range of tools such as graphical visualizers and molecular editors. Input to ab initio calculations for a molecule includes the constituent atoms, the coordinates of the atomic nuclei and a first guess at the wave function representing the electronic distribution. The wave function is the most complex input, and is generally represented in terms of basis functions. The basis functions may be chosen from one of many basis sets. The basis set is one of the principal adjustable parameters to any ab initio calculation. Choosing the wrong basis set can lead to time-consuming but meaningless results. Furthermore, CPU time and memory required for parts of the ab initio calculations vary as the number of basis functions (which depends on the choice of basis set and size of molecule), ranging from n^5 to $n!$, depending on the algorithm used. Many basis sets, e.g., STO-3G [21] and Dunning DZP [10], have been (and are being) developed, each optimized for a set of atoms or for a different type of calculation. Thus, the choice of basis set is a highly technical endeavor. A computational chemist familiar with the field, but not an expert on basis sets, who wishes to run a computational chemistry experiment, would have to:

- Determine what codes and basis sets are available, most probably by talking to the system administrator and other computational chemists.
- Determine the combination of codes and basis sets that are optimum for the particular molecule and properties of interest.
- Determine the input parameters required, by reading the available manuals.
- Determine the input and output parameter formats, maybe by looking at the examples in the manuals or by examining the input and output files of similar experiments performed by him or other computational chemists.

- Determine how to run the experiment, that is, find, and possibly modify, the appropriate Job Control Language script required to start the experiment on an IBM VMS system or the shell commands required on a UNIX machine.
- Write or find translators that would enable the output to be used as input to accessory programs such as molecular editors and visualizers.

A bench chemist who is looking for a theoretical basis to explain an experimental result currently has two options. He can do the molecular modeling himself, which means going through a steep learning curve to familiarize himself with the range of ab initio codes, basis sets and input and output formats. A lot of this information, especially the detail of input and output formats, is not useful to him in the long run. Alternatively, he can look for existing results on the molecular property of interest to him by undertaking a literature search for these properties or by approaching computational chemists likely to have performed the computational experiment(s) of interest. The lack of a central data repository for the results of computational chemistry experiments makes access to results of previous experiments extremely inefficient.

2.3 The Computational Chemistry Database Project

The Scientific Database Group at the Oregon Graduate Institute, in collaboration with computational chemist Dr. David Feller at Batelle Pacific Northwest Laboratories (PNL), began by investigating the applicability of object-oriented databases in addressing the data access problems of the computational and bench chemists. We envisioned the following data access scenario for chemists interested in using computational codes:

- Browse a data repository to see what codes and basis sets are available.
- Use the data repository's query facility to retrieve and browse existing experiments performed on molecules similar to the molecule under investigation, or to determine the properties of interest. Alternatively, use an expert system tool that interfaces with the data repository (such as the Computational Chemistry Input Advisor [15]

developed by Feller at PNL) to determine the optimum choice of code and basis set for the problem at hand.

- Start the experiment.
- If the experiment is successful, load the results into the database.

Such a data access facility would aid both the novice computational chemist and the bench chemist. Chemists could have access to a repository of past experiments with means to query and browse the data. Expert systems could provide an intelligent interface to the data repository, allowing a novice computational chemist to perform such complex tasks as setting up the input parameters for an experiment based on inputs to similar experiments performed in the past.

2.3.1 The Computational Chemistry Information Model

As a first step towards achieving an efficient data access scenario for computational chemists, Cushing developed an information model for a computational chemistry database, formulated independently of any specific database system [5, 6]. The main aim of the initial information model was to capture sufficient information about past runs of successful computational experiments to be able to support queries such as:

List the computational experiments performed on methane.
List all experiments performed by A. Chemist.
List all basis sets supported by the database.
Describe the STO-3G basis set.

Figure 2 is a diagrammatic representation of the information model. The model consists of entities and relationships between entities. An entity in the information model is represented as a box with rounded corners. Aggregations of entities are depicted as multiple entity boxes, and labeled with the keyword "grouped into". Superclass-subclass relationships are denoted by thick lines. Relationships between entities are represented as labeled lines between the entity boxes; a

dashed line represents a renaming of a superclass-class relationship by a subclass. A cardinality of greater than one is indicated by a black dot. Logical entry points are denoted by dotted line arrows, and are labeled with the name of the entity attribute that serves as the entry point. Below is a brief description of the entities in the information model and the relationships between them.

Experiment: An experiment entity represents a computational or laboratory experiment that may be performed collaboratively by several chemists. An experiment has a molecule as subject and produces zero or more observable properties. The experiment entity also holds information such as the date the experiment was started, date completed and the title of the experiment. It is related to the chemist, molecule and observable property entities in the model. An experiment is one of the two types:

> **Computational Experiment:** A computational experiment entity represents a type of experiment that is run on a computer, using one of the ab initio codes. It uses a basis set to represent the subject molecule and produces molecular orbitals in addition to observable properties as output. The computational experiment entity is linked to the basis set and program package (representing ab initio codes) entities.
>
> **Laboratory Experiment:** A laboratory experiment entity represents a traditional experiment performed in a laboratory. A laboratory experiment is performed using a specific apparatus. A laboratory experiment serves to confirm a computational experiment if the value of an observable property determined by a laboratory experiment is consistent with the value determined by a computational experiment. Conversely, a computational experiment is said to be confirmed by a laboratory experiment when the values of their observable properties are consistent.

Chemist: The chemist entity represents the chemist, that is, the agent, performing experiments. Many scientists can collaborate on an experiment, and a scientist is likely to perform more than one experiment A chemist may also author basis sets, alone, or in collaboration with other chemists. The chemist entity includes identifying information about a chemist, such as name, address and email address.

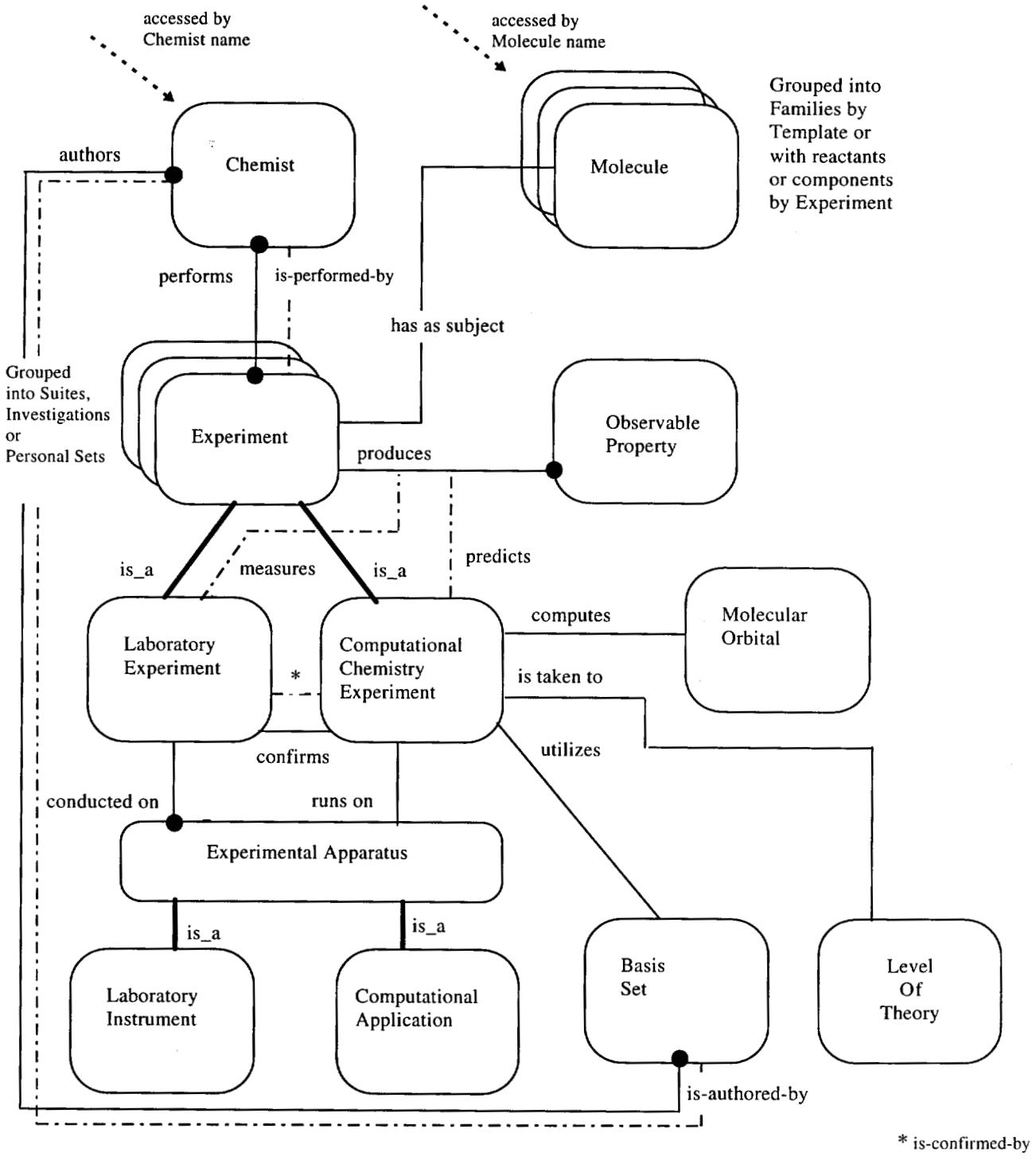

Figure 2: The Computational Chemistry Information Model

Molecule: A molecule entity is the subject of an experiment, either computational or laboratory. A molecule is made of atoms that have a location in three-dimensional space. The molecule entity holds the name and formula of the molecule. One molecule is the subject of an experiment, and each experiment has one molecule as subject.

Molecular Orbital: The molecular orbital entity represents an output of a computational experiment. The molecular orbital is a mathematical representation of the electronic distribution of the subject molecule, as optimized by the computational experiment.

Program Package: A program package entity represents an ab initio code on a specific platform, and a specific compiler version. It holds information about the platform and the compiler. Many computational experiments may be run with the same program package.

Basis Set: The basis set entity represents the basis functions used to construct the molecular orbital for molecules under investigation. Ab initio codes take as input a molecular orbital (wave function), which is generally represented in terms of the atomic orbitals of its constituent atoms. The atomic orbitals themselves are constructed from basis functions. These basis functions are authored by computational scientists, and are optimized for different types of calculations and atomic shell configurations. A computational experiment utilizes a single basis set, while a basis set may be utilized by many computation experiments.

Observable Property: An observable property is the output of either a computational or a laboratory experiment. It is a property name, value and unit triple. The value can be a simple value like an integer or a complex value such as a matrix. The representation of the value depends on the property name. An experiment can produce many observable properties, while an observable property is produced by one experiment.

This data model was implemented in four different database systems, by four teams of three people each. The aim was to determine the feasibility of implementing a scientific data model in an object-oriented database, and subsequently choosing a database for further work on

the computational chemistry database (CCDB) project. The four database systems are described below:

Encore/Observer is an object-oriented database under development at Brown University. The implementation was done using the first release of Encore. Documentation was inadequate, and some of the basic functionality had not yet been implemented. The consensus of the implementation team was that the system was not yet mature enough to use for the feasibility study [26].

Postgres is an extended relational system, under development at UC Berkeley. Adding new types in this system was not straightforward, since input and output functions and all operators for a type had to be provided when creating the new type. Only data members were inherited by subclasses, not methods. Object identifiers had to be used explicitly to establish relationships. It also suffered from lack of adequate documentation [23].

GemStone is one of the earliest commercially available object-oriented database systems, from Servio, Corp. Being an established commercial product, it came with good documentation and user support. The implementation team encountered no problems in creating new types, methods or in implementing the sample queries. The only major hurdle was populating the database [20].

ObjectStore is a commercial object-oriented database system from Object Design Inc. It has good documentation and user support. The implementation team had no problems in creating user-defined types, methods, or in implementing the sample queries. The problem faced here, too, was populating the database [7, 32].

Based on the implementation experiences described above, the choice of a database for further implementation was narrowed to ObjectStore and GemStone. We opted for ObjectStore because it was available to us on the SUN. At that time, we had access to GemStone only on the Tektronix workstations. Most of the computational chemistry codes run in the SUN/Unix

environment, but not on the Tektronix workstations. Also, computational chemists at PNL generally work on SUN/Unix systems.

2.3.2 Populating the Database

The experiences of the computational chemistry database implementation teams indicate that one of the most time-consuming tasks was populating the database with the results of past experiments. There were two problems. The first problem was correctly establishing relationships between new objects being loaded into the database and objects already in the database. For example, when loading an experiment into the database, information about the chemists who performed that experiment also has to be loaded in the database. Some of the chemists performing that experiment may already exist in the database, and not others. The question then is: how does one specify chemists so that the experiment is correctly related to the chemists already in that database, and new chemist objects are created only for those chemists not already in the database? More generally, the issue is how to ensure that object(s) being loaded into the database get correctly related to objects already in the database and trigger the creation of new objects only when appropriate. This problem is common to loading all databases. There are two approaches to solving this problem:

1. Assign an identifier to each object as it is loaded into the database. Subsequent objects being loaded into the database refer to database objects by identifiers. The onus of assigning identifiers and mapping from load objects to identifiers falls on the user, programmer or programming tool.
2. Define methods that determine "equality" based on values in "key" fields for every class of object in the database [31]. In this approach, we effectively embody in a method what we mean for two objects to be semantically the same, in the context of the database. This approach entails writing an "equality" method for every type of object as part of the database schema.

We chose the second approach in the CCDB implementation, as it is more general. We provide "equality" methods for every class in the database schema. Writing an "equality" method

for a class allows us to more closely model the real world. For example, an equality method on class "Person" could identify a person object with name "Tom Thumb" with another person object with the name "Thomas Thumb" by looking in an alias list.

The second problem was the amount of time and tedious effort required to (i) abstract the relevant information from the output files, and (ii) format it for a load program. It is not realistic to expect a scientist to go through the tedious steps of abstracting the relevant data from the output file, reformatting it for the database, and then loading the data into the database. Even if parsing and formatting programs are available, the chemist still has to go through an extra step to load the database. Moreover, a scientist will probably have to load the input data into the database as well, since it is necessary for him to know the exact input that generated the output. Additionally, a chemist may also have to explicitly enter metadata information that appears in neither the input nor output files, such as name of chemist performing the experiment or type of machine the experiment ran on. Given the amount of information that has to be loaded into the database along with output file, it is likely, that the data would not be loaded into the database, or would be loaded either incompletely or incorrectly. Ideally, the metadata, the input data and the output data would be captured automatically into the database. In practice, automatic data capture implies the database has the ability to communicate with the computational experiment generating the data, to capture the inputs, outputs and metadata associated with the experiment, and to convert the data into database objects.

Cushing proposed that the database system be extended to solve the second problem by providing computation services to the chemists. Computation services include the ability to generate input from database objects, launch an experiment, and capture the results into the database at experiment termination. Extending the database to provide computation services alleviates the problem of loading a database with inputs, outputs and metadata as a separate step from actually running the experiment. In Chapter 3, we outline Cushing's proposal for extending the database to provide computation services and describe the mechanism we have dubbed "computational proxies" that implements computation services.

CHAPTER 3

Computational Proxies

In Chapter 2 we discussed our efforts to build a database of past computational experiments and our realization that loading such a database was a non-trivial task. Cushing proposed extending the database to provide computational services. She proposed a database construct called a "computational proxy" to interface a computational experiment to the computational database, automating data capture, and thus addressing the problem of populating the database. In this chapter, we present the definition and functional specification of the computational proxy and discuss its functions.

3.1 Computational Proxies

A database extended to provide computation services enables a computational scientist to generate input from the database, to launch an experiment using the database input, and to capture the results of the experiment at its termination. Computation services can be provided in several ways. Cushing [5, 6] explored and rejected two traditional approaches to interfacing applications to a database: modifying the computational application to read and write directly to the database is impractical, while encapsulating the computational application as a database object does not provide adequate flexibility in controlling the running application. Cushing proposed that we model computational experiments as database objects, called computational proxies. The purpose of the computational proxy is to serve as the focal point of all communication and information about the ongoing experiment it represents. The tasks performed by the computational proxy include:

1. managing the data input for the launched experiment, accessing the database for the input parameters, and creating an appropriately formatted input file,
2. controlling and querying an ongoing experiment about intermediate states, and
3. capturing output data on termination of the experiment.

The computational proxy can be defined as an object that "stands-in" within the database for each computational experiment about to be scheduled, scheduled or recently completed. It encapsulates data and functionality needed by a user to start up, control and capture the results into the database from a computational experiment. This approach is "data-centered" rather than procedural, because the computational proxy is a database object, with encapsulated behavior, that relies on other database objects for information about the application and for input and output parameters.

Functionally, a computational proxy generates the application input file(s) for a computational experiment using information stored in the database. The proxy launches or starts a computational process, and controls that process as long as it is active. When the process has terminated, the proxy parses the output files and places the results into the database. The proxy marks the experiment as being completed, but the proxy is not yet deleted. The proxy is deleted at the logical completion of an experiment, which occurs when the chemist looks at the results and determines whether they should be discarded or made persistent. Figure 3.1 gives a simplified functional view of the proxy. The sequence of proxy events as depicted in Fig 3.1 are as follows:

1. The proxy automatically converts input parameters to particular program formats and creates an input file. The proxy may need to transfer the input file to the target machine of the computation.
2. The proxy launches (starts up) the computational process on the target machine.
3. The proxy periodically monitors the computational process during the course of its execution, and maintains a record that the chemist can access, independent of network or operating system details.
4. When the experiment has finished, the proxy automatically parses the output file and updates the database with the results.

5. Once the scientist has analyzed the results, the proxy object is deleted. The results can be either discarded or made "public" by the scientist. We call this step validation.

The scientist can now launch an entirely new experiment, or replicate an existing experiment object, modify its input as needed, and launch the modified experiment. In either case, a new proxy is generated.

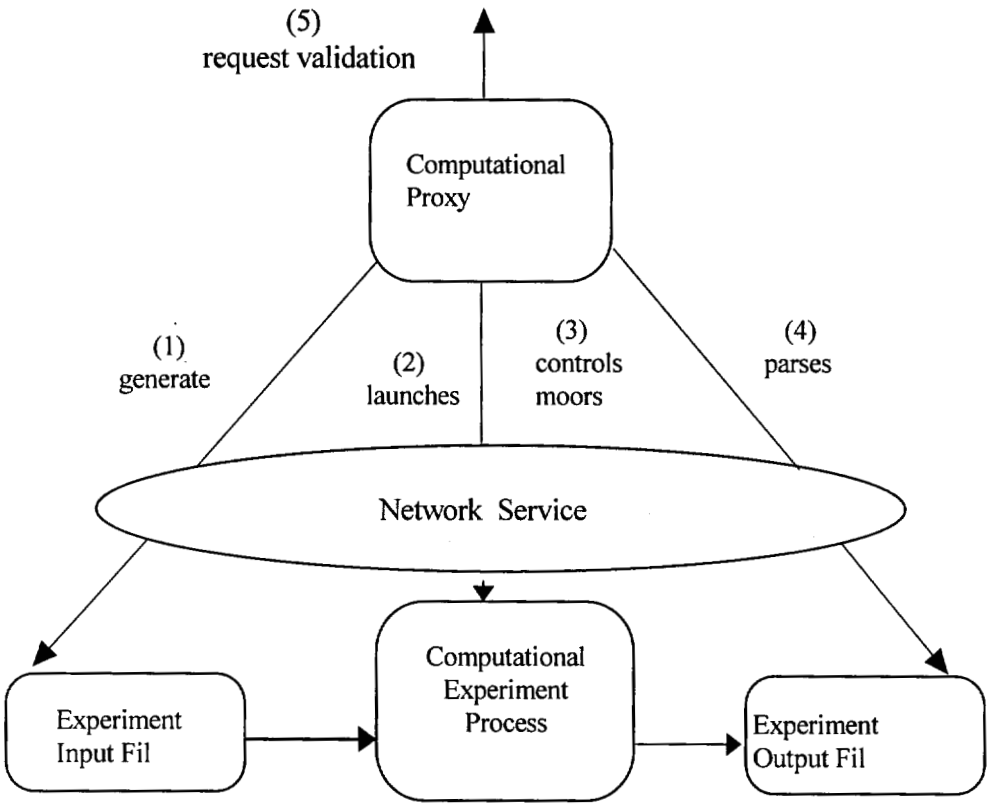

Figure 3.1 Computational Proxy functions for managing ongoing experiments

3.2 Functional Components of the Computational Proxy

The goal of the computational proxy is to provide the database user with a "transparent" interface to experiments performed using different codes or computational applications. By "transparent" we mean that the proxy shields the user from the syntactic details of input and output files, as well as from operating system and network details. It is important to note that a database user never directly sees or invokes a proxy. To provide this transparent interface between the database and a computational experiment, the proxy provides the services of *data input* to computational experiments, *data capture* from their output and *control* of the experiment processes themselves. However, prior to launching any experiment based on a particular computational code or application, the database must have access to information about the application, such as calculation types supported, input and outputs formats, and the mapping of input and output parameters to database objects. We call the process of making information about the application available to the database *application registration*. An application must be registered before a computational proxy object can be created for an experiment. Thus, registration is not a computational proxy function, but an ancillary function required by the computational proxy. We collectively call the computational proxy and application registration the "computational proxy mechanism". We discuss the functional components of the proxy mechanism below.

Application Registration: Before any experiment based on a computational application can be launched through the computational proxy mechanism, some information about the application has to be available to the database. Registration is the process of providing the database with sufficient information about the application and its computing environment to enable the proxy to run an experiment and capture its output. The application-specific information that has to be registered with the database falls into two categories:

(i) Information about the application itself. This includes the name of the application, the calculation types performed, the version of the application and the target processor types.

(ii) Information required by the proxy to run the application. This includes input and output parameters for the different calculations performed, the mapping between input

and output parameters and the database objects as well as the input and output formats of the application.

Once an application has been registered, the database has the information needed to determine the types of calculations performed by the application, the required and optional parameters for the different types of calculations and input and output formats (see Figure 3.2). Every experiment that is an invocation of this application will access this information to produce the experiment specific data files and controls.

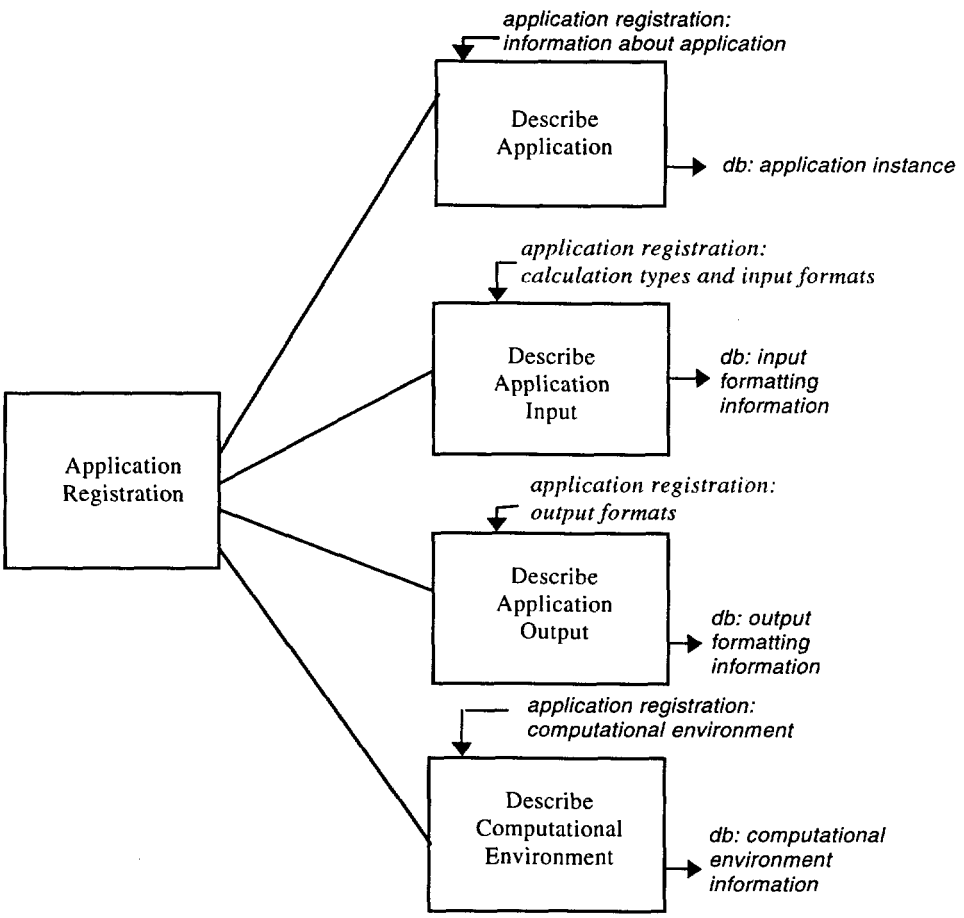

Figure 3.2 Application Registration functions

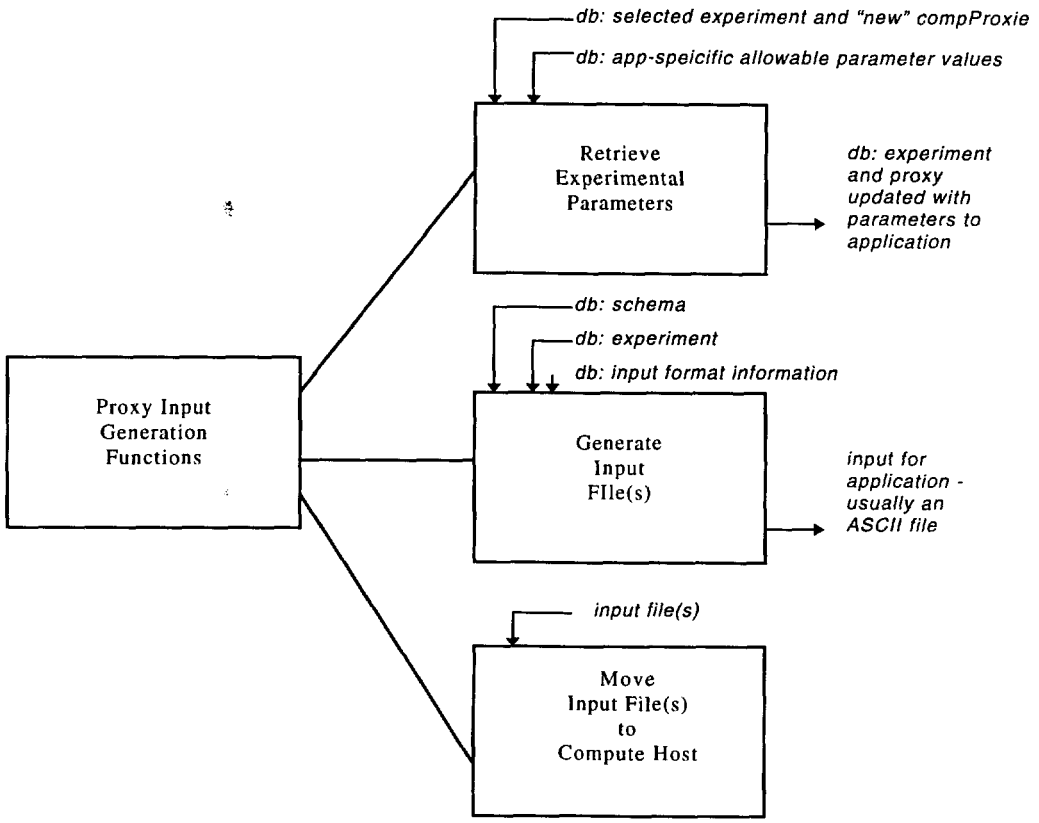

Figure 3.3 Application Input Functions needed by the Computational Proxy

Data Input: Prior to launching an experiment, the computational proxy goes through the following steps to ensure that the appropriate input parameters are available to a computational experiment (see Figure 3.3):

1. Retrieves experimental parameters: Before an input file is generated, all required input parameters are retrieved from the database. The computational scientist may have to be queried for any missing parameters. The input parameters required by the experiment may be specific to the scientific domain being modeled, specific to the application or the site.
2. Generates the input file(s): When a user requests the launching of an experiment, the proxy generates the input file(s) for the application. The input file is generated using

information registered with the database about input formats and mappings from database objects to data elements in the input file.

3. Moves input file to target machine: Finally, the proxy must transfer the input file(s) to the processor on which the application is to be run.

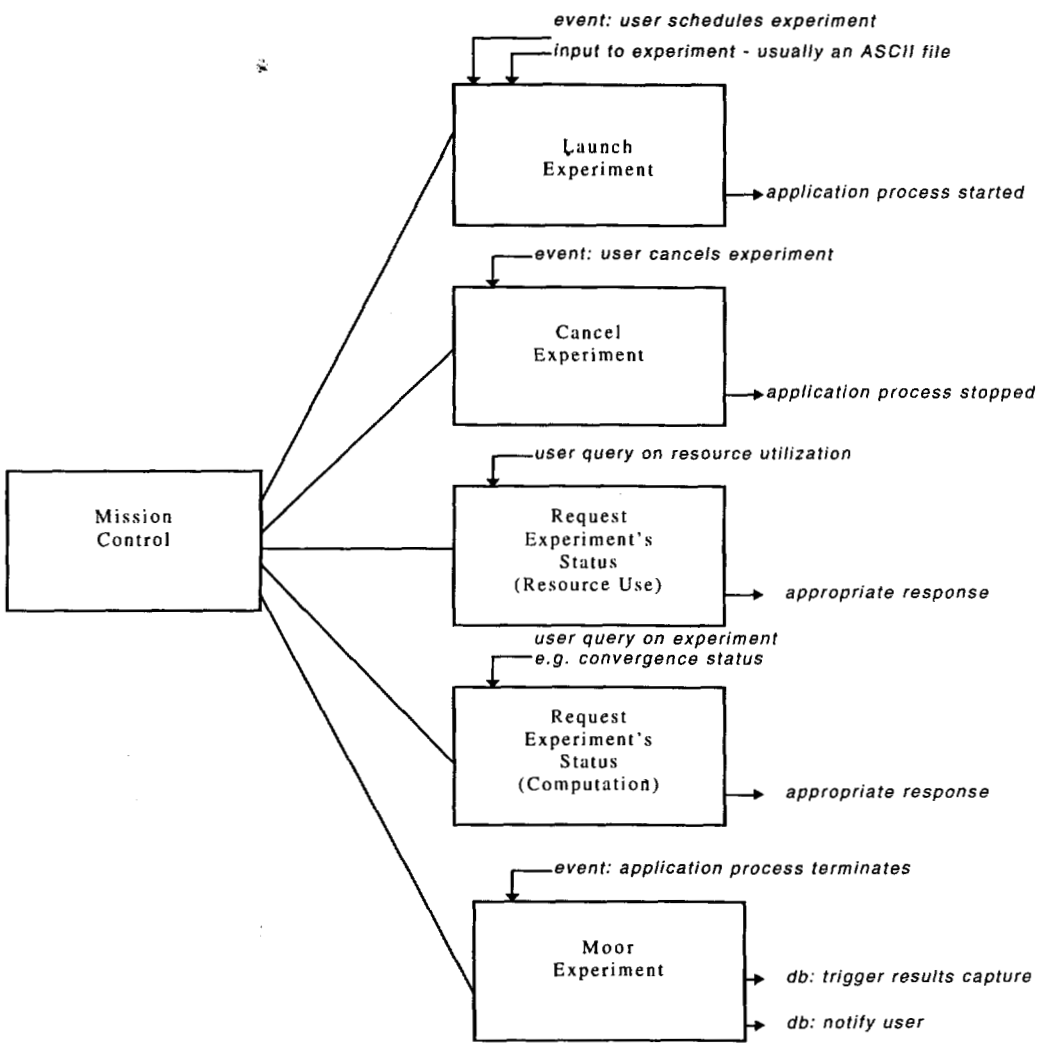

Figure 3.4 Mission Control functions needed by Computational Proxy

Mission Control: Once an experiment has been launched, a computational scientist may wish to query the ongoing process. Typical queries that a computational scientist may have of an ongoing process include:

- Has the experiment terminated?
- How much CPU time and disk space has the experiment consumed?
- How much clock-time has elapsed since the experiment was launched?
- Is the computation converging?

Since the experiment is unlikely to be running on the same host as the database, mission control must interface with the network. Mission control is responsible for launching and canceling experiments. Mission control also notifies the proxy when the experiment has completed and triggers it to capture the experiment output. Once the output has been captured, the experiment is considered *moored.*

Data Capture: When an experiment process has terminated, the results of the experiment are automatically captured into the database. Two general cases arise when an experiment terminates.

1. The application process terminates successfully. In this case, once the proxy is notified of successful termination, it parses the experimental results and places them in the database, where they can be viewed by the user. Data capture, or mooring, must be distinguished from experiment validation. Data capture is an automatic proxy function, which precedes data validation, an explicit user action. Once the user validates the experiment, that database is appropriately updated and the proxy deleted.
2. The process terminates, but the computation is incomplete. Data capture after an abnormal termination is not straightforward. A computation may have terminated abnormally due to insufficient resources. In this case, the user may wish to restart the experiment after increasing the resource availability. Or the computation may have failed to converge, and in this case the user may wish to change the input parameters to the experiment and restart it.

In either case, the termination status of the experiment has to be noted, the output files (if any) have to located, parsed and the output data elements mapped into the appropriate database objects.

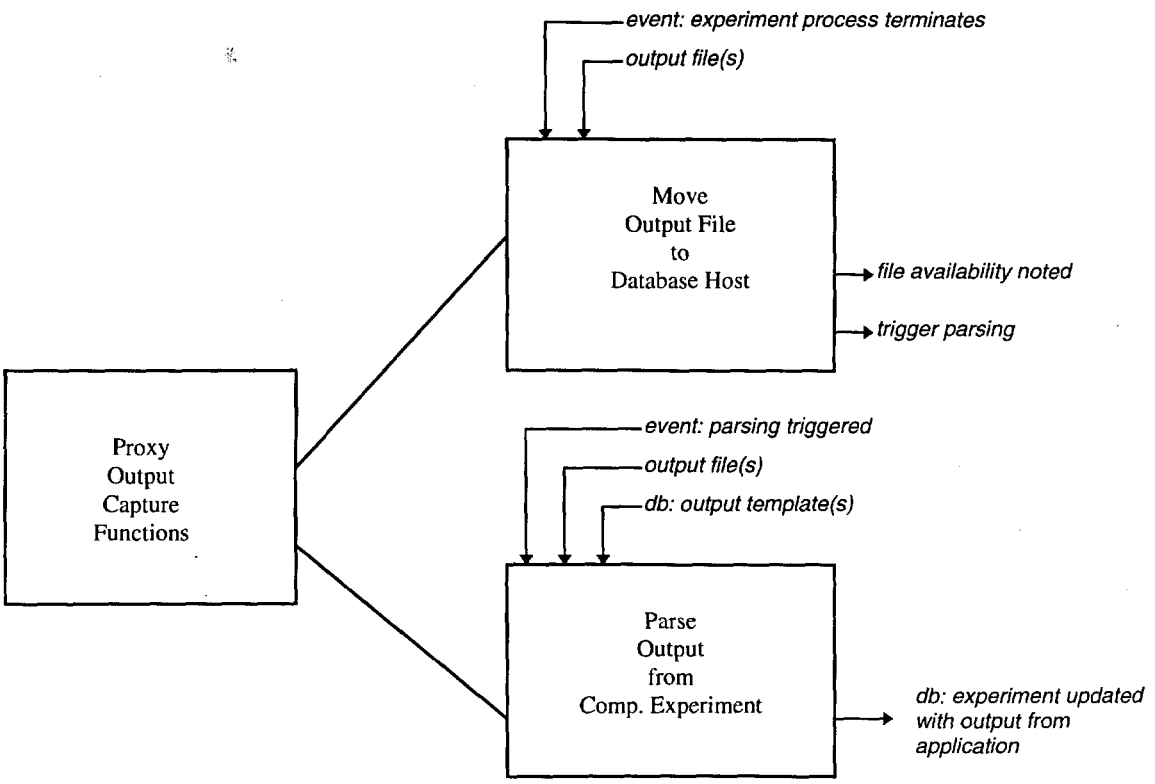

Figure 3.5 Experiment output capture functions needed by the computational proxy

3.3 Computational Proxies: The CCDB scenario

In the preceding sections, we presented a conceptual model of the computational proxy mechanism as a general method for providing computational services from within a database for computational applications that have complex input, output and metadata information associated with them. In the context of the CCDB and ab initio computational chemistry applications, the computational proxy mechanism solves the problem of loading the CCDB with the results of past experiments by providing a mechanism for interfacing legacy computational chemistry applications to the CCDB. We emphasize legacy applications, since more options are available when interfacing new applications, including writing the application as a CCDB method or coding the application to conform to data exchange standards.

The original goal of the CCDB was to provide a central repository of experimental results for the domain of ab initio computational chemistry. Extending the CCDB to provide computational services for the domain of ab initio computational chemistry would allow a user to:

1. Browse the CCDB to examine existing experiments.
2. Execute a database command to load inputs for a computational experiment into the CCDB.
3. Execute a database command to run the experiment. The CCDB may query the user for the name of the host or target processor on which to run the experiment.
4. Examine the termination status and outputs of the experiment, and validate the results.

In Section 2.2, we presented the scenario for a computational chemist who wished to run a computational chemistry experiment. Without a central database, he has no easy access to the results of past experiments. He has to undertake a literature survey and talk to computational chemists active in the field to find information about past experiments of interest to him. If he wishes to run an experiment, he has to familiarize himself with the syntactic details of input and output format for each of the computational applications and basis sets of interest to him. In

Section 2.3, we presented a user scenario for running computational experiments given a central database containing computational results. The results of past experiments are available in one central location. When browsing past experiments, the user does not have to familiarize himself with the output syntax of different computational experiments, only the database format. Information about basis set libraries and their usage can also be gleaned by browsing past experiments. However, if he wishes to run a computational chemistry application himself, he still has to learn the input and output syntax of the application. In addition, he will have to go through a number of steps to load the results of his experiment into the database to make it available to future database browsers. Contrast this with the user scenario for a CCDB extended to provide computation services. We see that the computational proxy mechanism promises an environment for a computational scientist that allows him to concentrate on the science. To determine the feasibility of implementing the computational proxy design, we undertook to implement a prototype proxy for the ab initio computational chemistry application GAMESS in the CCDB. In the next chapter, we discuss the design and implementation of a computational proxy for GAMESS in detail.

CHAPTER 4

Computational Proxy Prototype for GAMESS

In Chapter 3 we presented the definition and functional specifications for the computational proxy. The goal of this thesis is to offer a proof of concept of the proxy mechanism. Towards this end, a prototype proxy mechanism was implemented in the CCDB, interfacing the CCDB to the stand-alone ab initio computational chemistry application GAMESS. In Section 4.1 we present an overview of the input and output parameters and formats for GAMESS. Section 4.2 discusses the design constraints and architecture of the proxy prototype. In Section 4.3 we describe the implementation of the prototype computational proxy. In Section 4.4 we present the validation for the prototype proxy implementation.

4.1 GAMESS

General Atomic and Molecular Electronic Structure System (GAMESS) is one of the ab initio computation applications used by chemists to predict molecular properties [11]. It determines the wave functions for the molecule under study, based on one of a range of quantum mechanical methodologies such as the self-consistent field (SCF) or configuration interaction (CI). Molecular properties such as minimum energy, vibrational frequencies or optimized geometry are then determined, based on the molecular wave functions. GAMESS is written in FORTRAN, and available on many platforms, including SUN, DEC and IBM. Input and output is via ASCII files. Names of input files, temporary file location and output file locations are communicated to GAMESS through environment variables in a Unix environment.

BASIS group (optional): The BASIS specification allows the user to choose one of the internally defined basis sets in GAMESS. The use of this parameter is discouraged at PNL. PNL maintains a basis set library, and chemists are encouraged to input basis set values from the library. Basis sets can be explicitly specified in the DATA group.

DATA group (required): Global data about the molecule under investigation, such as point group symmetry and nuclear coordinates of the symmetry unique atoms are specified in the DATA group. The basis functions for the molecule can also be explicitly specified in this group, if the chemist prefers not using the internally defined basis set library in GAMESS. The data is input as a series of card images (lines) in a fixed order. Each card image is free-format.

Other groups : A number of other groups can be specified. For example, there is the SCF group that provides additional control over the SCFTYP parameters. Similarly, for each RUNTYP possible, an additional control group can be specified to either refine or further constrain input for that run type.

The description of all possible input parameters and their dependencies is quite complex. We learned through discussions with the computational chemists at PNL that most of the GAMESS runs at PNL use only a small subset of all possible allowable input parameters. We have focused on supporting only a subset of the commonly used input parameters in the prototype. Supporting all the parameters would enhance the usability of the database, but would take a lot of time and effort on our part to understand the domain, without adding relevant feedback on the feasibility of the proxy. The input parameters that we decided to support in the prototype are as follows:

CONTRL group: The TIMLIM and MEMORY parameters are automatically specified by the database. SCFTYP can only take the value RHF or the defaults, which are RHF for even number of electrons and UHF for odd number of electrons. RUNTYP can be either ENERGY or OPTIMIZE. MPLEVL can take all allowed values, provided it is consistent with the RUNTYP and SCFTYP values.

BASIS group: The use of the BASIS group is not supported by the CCDB. Basis set functions are input via the DATA group from the PNL basis set library.

DATA group: In this group, the run title, the molecular symmetry, the nuclear coordinates of the symmetry unique atoms constituting the molecule are provided. Also, the basis set functions are input from the PNL basis set library.

Other groups: We do not use any other group to provide additional control information.

A sample input file for a GAMESS experiment on the water molecule to determine the minimum energy using the STO-3G basis set is shown below. The input files for the subset of parameters we support follow a simple pattern. The input begins with the CONTRL group parameters of TIMLIM, MEMORY, SCFTYP, UNITS RUNTYP and MPLEVL. The CONTRL group is followed by the DATA group. First in the DATA group is the title of the experiment, followed by the symmetry group of the molecule under investigation. The symmetry group specification is followed by each of the symmetry-unique atoms constituting the molecule, with its atomic number, position in the atom and its basis functions.

```
 $CONTRL TIMLIM=999.0 MEMORY=2000000 $END
 $CONTRL SCFTYP=UHF UNITS=BOHR  $END
 $CONTRL RUNTYP=ENERGY $END
 $CONTRL MPLEVL=0  $END
 $DATA
Energy - H2O/STO-2G/MP0
CNV  2

Hydrogen    1.000000   0.000000   -1.430429   0.983250
  S    2
    1   1.309756   0.430128
    2   0.233136   0.678914

Oxygen    8.000000   0.000000   0.000000   -0.123907
  S    2
    1   49.980971  0.430128
    2   8.896588   0.678914
  S    2
    1   1.945237   0.049472
    2   0.493363   0.963782
  P    2
    1   1.945237   0.511541
    2   0.493363   0.612820
 $END
```

Output from a GAMESS run cannot be described by a single, simple pattern or format. The output format depends on the input parameters, especially RUNTYP. The properties of interest that should be captured into the database and the output file format, vary based on the

RUNTYP. For example, if the RUNTYP=ENERGY, the property of interest is the minimum energy, and the output file format is:

```
ITER EX DEM  TOTAL ENERGY      E CHANGE  DENSITY CHANGE   DIIS ERROR
   1  0  0   -77.004636477   -77.004636477   0.212191918   0.607223863
           * * *   INITIATING DIIS PROCEDURE   * * *
   2  1  0   -77.071803224    -0.067166747   0.033634539   0.040156078
   3  2  0   -77.072855730    -0.001052506   0.010542709   0.008447370
   4  3  0   -77.072916465    -0.000060735   0.001713877   0.001119763
   5  4  0   -77.072917783    -0.000001318   0.000028962   0.000023533
   6  5  0   -77.072917783    -0.000000001   0.000002953   0.000002268
   7  6  0   -77.072917783     0.000000000   0.000000076   0.000000108

          -----------------
          DENSITY CONVERGED
          -----------------

 FINAL ENERGY IS      -77.0729177835 AFTER   7 ITERATIONS
```

If the RUNTYP=OPTIMIZE, then the property of interest is the final optimized geometry, which includes the nuclear coordinates of the atoms in the molecule and the molecular orbitals. The output file format in this case is:

```
Geometry Optimization of H2O/STO-3G
     ------------------------------------------------------------------------------

 COORDINATES OF SYMMETRY UNIQUE ATOMS (ANGS)
   ATOM    CHARGE       X              Y              Z
 ------------------------------------------------------------
 HYDROGEN     1.0   0.0000000000  -0.7581188881   0.5369444439
 OXYGEN       8.0   0.0000000000   0.0000000000  -0.0988230797
 COORDINATES OF ALL ATOMS ARE (ANGS)
   ATOM    CHARGE       X              Y              Z
 ------------------------------------------------------------
 HYDROGEN     1.0   0.0000000000   0.7581188881   0.5369444439
 HYDROGEN     1.0   0.0000000000  -0.7581188881   0.5369444439
 OXYGEN       8.0   0.0000000000   0.0000000000  -0.0988230797

          INTERNUCLEAR DISTANCES (ANGS.)
          ------------------------------

                        HYDROGEN       HYDROGEN       OXYGEN

   1  HYDROGEN         0.0000000      1.5162378 *    0.9894163 *
   2  HYDROGEN         1.5162378 *    0.0000000      0.9894163 *
   3  OXYGEN           0.9894163 *    0.9894163 *    0.0000000

   * ... LESS THAN  3.000

NUCLEAR ENERGY    =         8.9064118624
          ELECTRONIC ENERGY =       -83.8723124619
          TOTAL ENERGY      =       -74.9659005995

          ------------------
          MOLECULAR ORBITALS
          ------------------
1          2          3          4          5          6          7
              -20.2516    -1.2575    -0.5939    -0.4597    -0.3926     0.5818     0.6927
                 A1         A1         B2         A1         B1         A1         B2
1  H    1 S  -0.005584   0.155595   0.449220   0.295099   0.000000  -0.769167  -0.814635
2  H    2 S  -0.005584   0.155595  -0.449220   0.295099   0.000000  -0.769167   0.814635
3  O    3 S   0.994216  -0.233767   0.000000   0.104032   0.000000  -0.125822   0.000000
4  O      S   0.025847   0.844453   0.000000  -0.538148   0.000000   0.820148   0.000000
5  O      X   0.000000   0.000000   0.000000   0.000000   1.000000   0.000000   0.000000
6  O      Y   0.000000   0.000000   0.612686   0.000000   0.000000   0.000000   0.959817
7  O      Z   0.004164   0.122830   0.000000   0.755894   0.000000   0.763521   0.000000
```

4.2 Design of the GAMESS Proxy

In Section 2.3.3, we discussed why populating the CCDB with the results of successful experiments is a hard problem. Cushing proposed the data-centered computational proxy mechanism as a way of extending the CCDB to provide computational services and automatically populating the CCDB with the experimental results as a by-product of running experiments. We have designed and implemented a prototype computational proxy mechanism in the CCDB for the stand-alone computational package GAMESS. The goal of the prototype computational proxy implementation was to gain a better understanding of:

- the effort involved in implementing the proxy mechanism within the framework of an object-oriented database,
- the efficacy of the prototype computational proxy mechanism in interfacing GAMESS experiments to the CCDB, and
- the extensibility of the prototype proxy mechanism to other computational chemistry packages.

Design and implementation choices for the prototype computational proxy mechanism were guided by the following considerations:

(i) We wanted the prototype design to be as independent of the particular computational package GAMESS as feasible. Although we expected to implement the prototype for the single package GAMESS, the computational proxy mechanism was proposed to address the need to interface an entire class of applications to a central database, and we wanted the design to reflect this.

(ii) The computational chemist typically works in a networked environment, containing heterogeneous computing hosts. Although our implementation was targeted to be on a network consisting only of SUN workstations, we wanted the prototype design to work on heterogeneous platforms.

(iii) The computational proxy mechanism aims at interfacing an entire class of applications to a central database. Hence, as a usability issue, it is important that new applications can be interfaced to the central database without extensive programming.

GAMESS is a stand-alone FORTRAN program, typically available on a variety of platforms and host machines in the chemist's computing environment. Extending the CCDB to provide computational services for GAMESS via the computational proxy mechanism implies that a chemist can launch, monitor and capture results for his GAMESS experiment from within the CCDB. Since the chemist may wish to run his GAMESS experiment on any suitable host in his computing environment, the GAMESS proxy has to perform the functions of data input, mission control and data capture across the network. Thus the GAMESS proxy has to encapsulate some level of network communications. One of our first design decisions was to encapsulate all network services required by the proxy into an entity we called the Compute Monitor. The function of the Compute Monitor is to act as the locus of communication and control for experiment processes running across the network. We expect the computational proxy for the GAMESS experiment to communicate all control requests to the Compute Monitor, which is then responsible for communicating with the appropriate compute host across the network. Analogous to the Compute Monitor, we designed an entity called the Database Monitor, whose main function is to act as the locus of control for all database access by the ongoing experiments. For example, when an experiment terminates, the Database Monitor is notified so that parsing can be triggered and the results captured into the database. Conceptually, the computational proxy mechanism provides two kinds of services: database services through the Database Monitor, and network services through the Compute Monitor.

Figure 4.1 is a diagrammatic representation of the computational proxy mechanism architecture outlined above. The computational proxy mechanism comprises the following entities:

Computational Proxy Object: The computational proxy and related objects reside in the database. The computational proxy object is essentially a database handle for an ongoing experiment. It stores information about the ongoing experiment such as the compute host of the experiment and the process identifier. The relationships of an ongoing experiment with other database objects is maintained through the computational proxy object.

Compute Monitor: The compute monitor is the process that encapsulates the interface between the computational proxy object and the network and acts as the locus of control for experiments

running across the network. All experiment control requests made of the CCDB are forwarded by the computational proxy object to the Compute Monitor.

Database Monitor: The database monitor is the process that acts as the locus of control for access to the database. It acts as a buffer between the CCDB and the compute monitor, and is responsible for transferring data to and from the CCDB and the computational chemistry experiments.

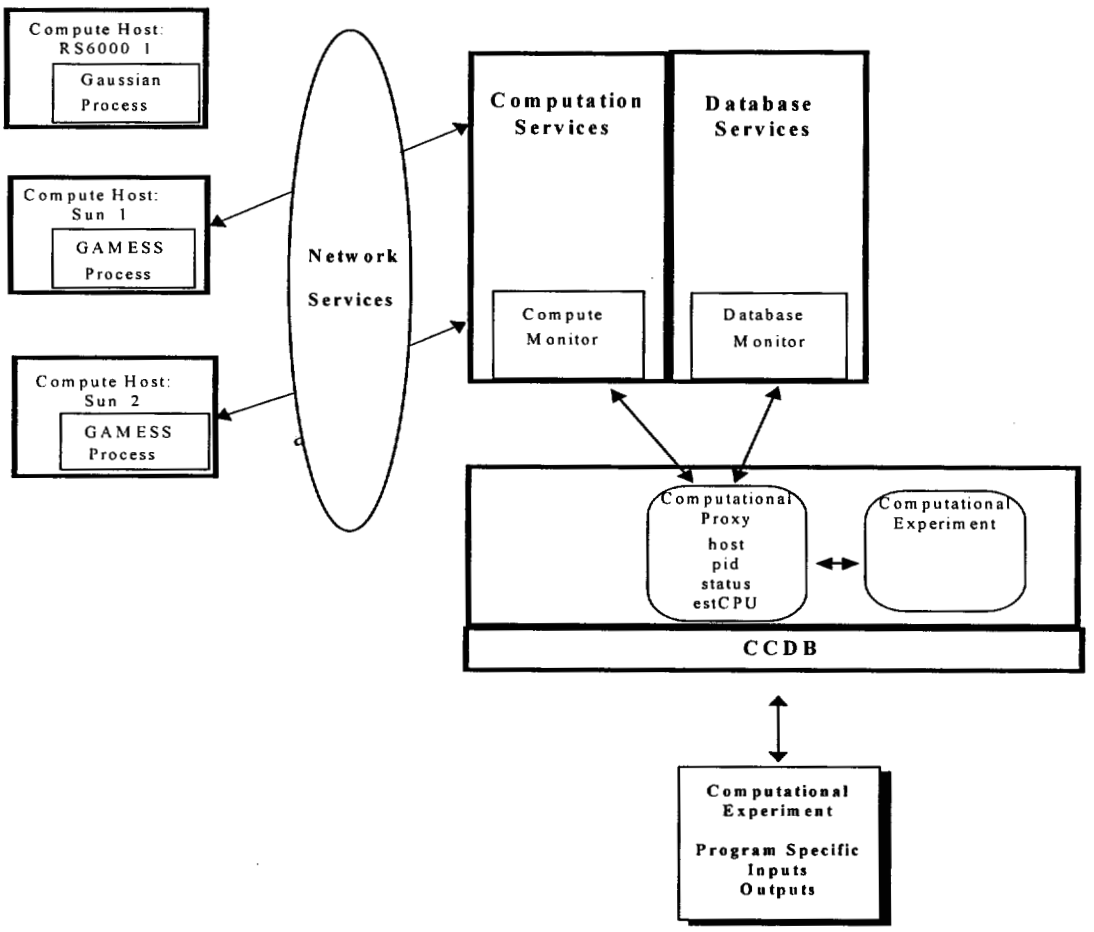

Figure 4.1 Computational Proxy Architecture

4.3 Implementation of the GAMESS Proxy

The first step in the implementation of the prototype computational proxy mechanism was to introduce the computational proxy and related objects into the computational chemistry information model (Section 2.3.1). As described earlier, the computational proxy and related objects store two categories of persistent information, application-specific and experiment-specific information. The Computational Application entity in the information model (see Figure 2.1) already stores some application-specific information, such as name of the package, target platform or compute host, version of the application and compiler version. The entity Computational Chemistry Experiment stores some experiment-specific information, such as the title of the experiment, its subject molecule, the basis set used and the results (as a set of Observable Property). However, information required by the computational proxy mechanism, such as input and output formats, compute host or process id, is not stored by either the Computational Application or Computational Chemistry Experiment objects. Hence, we introduced an entity called a Computational Proxy in the model to store the required information. Each Computational Chemistry Experiment entity has either zero or one Computational Proxy entity associated with it. The Computational Chemistry Experiment holds the process-independent data for an experiment, while its associated Computational Proxy maintains process information. A Computational Chemistry Experiment that has not yet been scheduled or that has already been validated has no associated Computational Proxy. The Computational Proxy essentially acts as a place holder in the CCDB for an *ongoing* experiment, while the Computational Chemistry Experiment represents the experiment in all other respects within the CCDB. The Computational Chemistry Experiment itself is linked to the input data, such as the Basis Set and Molecule entities, Computational Application entity, output data information such as (a set of) Observable Property and metadata data information such as performing Chemists(s), date started, date completed and CPU time used.

4.3.1 The Class compProxy

After modifying the information model, our next step was to implement the Computational Proxy entity in the CCDB. We added a class called *compProxy*[2] in the CCDB schema to represent the Computational Proxy entity. (See Appendix A for a listing of the CCDB schema). An instance of the *compProxy* class is automatically created for every Computational Chemistry Experiment (an instance of the class *compExperiment*) that is scheduled. The *compProxy* instance holds information about the experiment process such as process identifier and compute host name. Initially, we stored the data format information with the *compProxy*, since input and outputs have to be formatted for every experiment. However, data formats are application-specific, and the format information is uniform across all experiments based on that particular application. Logically, format information should be part of the Computational Application entity (represented by the class *codePackage* in the CCDB). Associating the data formats with the *compProxy* means recording them redundantly for all experiment processes using the same *codePackage*. Therefore we simplified the *compProxy* class to hold only process information, which is uniform across all applications. Thus a single class *compProxy* suffices for all experiments, irrespective of *codePackage* used. Simplifying the *compProxy* class required us to extend the class *codePackage* to include format information. In modifying the class *codePackage*, we considered two alternatives:

- Implementing the class *codePackage* as a superclass, and deriving a subclass for each computational chemistry application in the CCDB from it. The formatting information for each application would be stored as member functions in the derived class. We would thus use the power of virtual functions to transparently choose the correct formatting method for an experiment. For example, from the superclass *codePackage* we would derive the subclass *GAMESSPackage* to store GAMESS specific format information. This approach is conceptually clean: each package has its own subclass and package boundaries are maintained. Methods and implementations for one package can be changed without affecting

[2] In the rest of the this thesis, class names will be *italicized*, command names will be in **bold** and host names will be in `Courier`.

other packages. Because of virtual functions, all messages for formatting will appear to go to the class *codePackage*, and a uniform interface is maintained.

- Implementing the Computational Application entity as the class *codePackage*, with each instance of the class *codePackage* representing a particular application. For example, to interface a new application such as Gaussian to the CCDB, we would create a new instance of the class *codePackage*.

Unfortunately, both these approaches have drawbacks that go against the goals of the proxy implementation. Deriving a new subclass from *codePackage*, as in the first approach, requires changing the CCDB schema. A schema change, at a minimum requires a recompilation of the CCDB code, and could potentially trigger a need for migrating existing databases to the new schema. The version of ObjectStore we used (version 1.2) does not support schema evolution. Currently, work is being done on schema evolution, and there are many questions to which we do not have answers: Will the schema evolution be on line or off line? How much time would it take? How much user intervention would be needed?

We chose the second approach because of the unanswered questions in the approach above. Using this approach, we add a new instance in the CCDB for every new computational chemistry application. The problem however, is that in C++ and ObjectStore DML, only data members of a class can take instance-specific values, member functions apply to the whole class. So methods for formatting data apply to all instances of the class *codePackage*. Thus, in opting to instantiate an object for each computational application, we choose not to use the superclass-subclass relationship, and consequently, lose the power of virtual functions to determine the appropriate formatting methods for a given application. Given this, we considered associating the data formatting methods with the *compProxy* class. What we would have liked to do was to create a "master" *compProxy* when an application was first registered, and clone it for every experiment using that *codePackage*. However, we did not progress along this route as ObjectStore does not support cloning of objects.

Our solution to the problem with the second approach was to represent each computational chemistry application as an instance of the class *codePackage*, while implementing our own means of associating formatting functions with each instance. The class

codePackage includes the single input formatting function member "fmtBSI". The fmtBSI is essentially a function pointer to a case statement that maps from the *codePackage* name to the appropriate formatting function for the computational chemistry application represented by that *codePackage*. We opted to have fmtBSI as a wrapper function, putting the case statement and the actual formatting functions outside the database so that on adding a new application, only the case statement and formatting functions can be recompiled, rather than triggering a recompilation of the entire CCDB code (see code below).

```
codePackage::fmtBSI(CompProxie* cp, FILE* )
{
        callit(cp, fp);
}

extern  void fmtBSIGAM(CompProxie*, FILE*);
extern  void fmtBSIGauss(CompProxie*, FILE*);

void callit(CompProxie* cp, FILE* fp) {
   if (strcmp(cp->name, "GAMESS") {
       fmtBSIGAM(cp, fp)
   }
}
```

To interface a new computational chemistry package to the CCDB, with this scheme, we first create a *codePackage* instance to represent the application. In the CCDB, a new *codePackage* instance can be created with the **loadCP**[3] command. The *codePackage* instance stores information such as the name of the application, compiler version, date available and whether it is an archived or active version. Next, we write the functions required to handle data formatting for runs of that *codePackage*. Then we modify the case statement to include the case for the new application, ensuring that the appropriate formatting functions get called for this new instance of *codePackage*. For example, to interface the code package Gaussian to the CCDB, we first create a *codePackage* instance for Gaussian using the **loadCP** command. The *codePackage* instance contains the name of its data input formatting method, such as fmtGauss. We then write the fmtGauss function (outside the database), modify the case statement, recompile these two functions, and link the modified functions with the CCDB object modules.

We have used the approach above for registering only the data input formatting functions. We could have followed the same methodology for implementing the data capture functions. It is logical for the data input formatting functions to execute on the CCDB server,

[3] For a short summary of the CCDB commands, see Appendix C.

since the input functions access the database to retrieve the information to be formatted. However, the user may want to control whether the data capture functions are executed on the compute host or the CCDB server. The user may wish to examine the output files before the files are parsed and the data placed in the database, in which case it makes sense for the files to be transferred to the CCDB server, and eventually parsed there. On the other hand, the data files may be extremely large, and it may make more sense to parse the output data files and only transmit the subset of relevant data over the network. In ObjectStore DML, the user cannot control where methods execute. Thus, we devised an alternative approach to register the data capture information, so that the user has some control over where the output data parsing occurs, as is discussed in Section 4.3.2.

We have outlined above our first effort at implementation of the application registration functionality of the computational proxy mechanism. Lessons learned from this implementation helped us develop a more general approach, discussed in Chapter 5.

4.3.2 The Data Control Functions

In Section 4.3.1, we described how we modified the class *codePackage* and introduced the class *compProxy* to implement a first version of application registration and to provide a place-holder for information required for mission control in the database. The computational proxy mechanism for GAMESS must be able to handle data input, data capture and mission control for a GAMESS experiment. We refer to these functions as the data control functions. In this section we describe how we implemented the functions to manage the data input, data capture and mission control of GAMESS experiments.

Data input for a GAMESS run requires accessing CCDB objects related to the *compExperiment* instance for input information and creating an input file with the appropriately formatted input data. We have written a method, fmtGAMESS (Appendix B), that is called from the fmtBSI method of class *codePackage* to handle the data input for a GAMESS run. The prototype implementation limits the types of GAMESS calculations supported within the CCDB

to a few typical run types. The formatting information for the input of these types of runs was quite uniform and easily encoded within the fmtGAMESS function. The function fmtGAMESS retrieves data from the *compExperiment* object and its associated *BasisSet* and *Molecule* in the CCDB. Each *BasisSet* is essentially a collection of basis functions for different atoms, derived under the same assumptions. As part of the data input, the basis functions for the atoms comprising the subject molecule are extracted from the *BasisSet* into a *BasisSetInstance*. The information about basis functions is appropriately formatted and added to a *.inp file. The extension ".inp" is required by GAMESS. We generate a unique name for the file, using parts of the chemist's name, molecule name and the basis set used. For a more general solution, we would probably append date and a random number (making for very long file names). The generated file name is communicated to the associated *compProxy*. From a chemist's perspective, all he does is to execute the command **runCE**, and the CCDB launches the experiment on the compute host requested by him.

Mission control of a GAMESS run can be at two levels. At the process level, information about a GAMESS run, such as CPU time used and disk space are monitored. At the computation level, intermediate stages of a GAMESS computation are monitored. The intermediate values of interest include the minimum energy calculated at each iteration and the intermediate geometry for a geometry optimization run. A GAMESS run, irrespective of type of calculation it performs, iteratively computes the minimum energy and molecular orbitals. The molecular orbitals that describe the electron distribution of the molecule are output as a matrix of coefficients of the basis functions. Typically, a computational chemist uses a visualization program to display the molecular orbitals. Ideally, a computational chemist would like to see the minimum energy (and possibly also the first derivative of the energy function at the minimum energy) displayed graphically, and a visual depiction of the molecular orbitals, updated at every iteration. We have not implemented any monitoring of intermediate computational values, because (i) monitoring intermediate computational values is similar to parsing the output values, as both the intermediate values and the final values are written to the same output file and in the same format, and (ii) to the best of our knowledge, computational packages currently do not provide an easy way to examine intermediate computational values either. Thus, we have opted, given the time constraints, not to duplicate functionality in this feasibility study.

An interested chemist currently monitors his experiment at the process level by executing the appropriate process status command on the machine running his GAMESS experiment. We implemented the equivalent mission control functionality at the process level in the CCDB command **queryCE**. The **queryCE** command has been implemented as a CCDB command rather than a method associated with the *codePackage* or *compProxy* class because accessing process level information is uniform across all experiment processes of all applications. Essentially, we are querying the operating system about a process, and the only information that is retrieved from the CCDB is the process id and compute host. In response to the **queryCE** command, the title of all experiments that have not been validated are displayed. The chemist can enter the number of the experiment he wishes to query. The **queryCE** command causes a "ps" to be forked on the host where the experiment is running, and displays information about the process such as CPU time used and process status. If the experiment process has terminated, the CPU time used, termination time and termination status are displayed. The Unix ps command itself is not portable, but most operating systems provide a similar interface to determine process status.

At the normal termination of a GAMESS run, the output data has to be captured into the CCDB. For a GAMESS experiment, the output data is in an output file, which has to be parsed to extract the relevant properties and values into the appropriate CCDB objects. The format of the output files exhibits much more variation than the input file formats, even in the limited types of GAMESS runs supported by the CCDB. The output format depends on the type of calculation performed, the molecule, and the basis set used. Hence, it is more difficult to encode the output formats than the input formats. Given the complex dependency of the output formats on both the input and the type of calculation performed, we chose to capture the minimum energy only. The minimum energy is output by all GAMESS experiments, and in the same format. We have written a PERL [36] script that parses the output file for the minimum energy value, and updates the minimum energy attribute of the *compExperiment* object. The output formatting functions differ from the input formatting functions in that the chemist may wish to control, on a per experiment basis, where the output is parsed. The input formatting functions are linked into the CCDB code, are called through the *codePackage* member function fmtBSI, and always execute on the client host. We could have written an independent C parsing program that was called on experiment termination, but then we would have to ensure that an appropriate executable was

available for all compute hosts. Instead of dealing with cross-platform compiler issues, we chose PERL, which is portable and interpretive, and the script is small enough to be shipped over the network if not found on the compute host.

An experiment is considered to have finished or terminated only after the chemist has examined the output and decided whether it should be made persistent or discarded. We call this step validation. A chemist validates the result of a CCDB experiment using the **validateCE** query. On validation, the *compProxy* is deleted. The minimum energy is either maintained (as a *compExperiment* attribute) or deleted, depending on the chemist's request. The prototype implementation does not incorporate any security measures and permits any user to validate any completed experiment.

4.3.3 The Compute Monitor

The Compute Monitor is the part of the computational proxy mechanism that coordinates communications between the proxy and experiments that may be running on a variety of compute hosts on the network that constitutes the chemist's working environment. All requests for launching an experiment from the CCDB are directed to the Compute Monitor. Ideally, the Compute Monitor would be able to either recommend a host for the experiment based on availability of the application, projected resource requirements of the experiment, resources available on the compute hosts and load information, or automatically schedule the experiment on the optimum choice of compute host. In the prototype implementation, however, the chemist specifies the compute host for the experiment. Once an experiment has been scheduled, the Compute Monitor coordinates all communications between the CCDB and the experiment, including transferring input and output files, output property values and mission control requests. In a network of heterogeneous compute hosts, we expect the Compute Monitor to handle the data conversions that may be required in moving data from one host to another. We have designed the Compute Monitor to be autonomous because the functions the Compute Monitor performs are not specific to the domain of computational chemistry. The Compute Monitor essentially

encapsulates the communications from a database to diverse applications running in a heterogeneous network.

The Compute Monitor consists of a "monitor" daemon process running on the CCDB host and "compute" daemons running on each compute host on the network that may be host to an experiment launched from the CCDB. Requests to launch, monitor or moor an experiment are directed by the CCDB to the appropriate compute daemon. At the termination of an experiment process, the compute daemon transfers the output files or property values to the monitor daemon, which then updates the CCDB.

In the prototype implementation, the CCDB resides on the SUN/Unix platform `smoked`. Two SUN/Unix platforms, `smoked` and `coho` are networked together and have a shared file system, managed by NFS. GAMESS can be invoked on both `smoked` and `coho`. The prototype Compute Monitor manages communications over the network between the CCDB and GAMESS experiments running on `smoked` or `coho`. The CCDB Compute Monitor consists of a monitor daemon, running on `smoked`, and a compute daemon running on each of `smoked` and `coho`.

A computational chemist who wants to run a GAMESS experiment executes the CCDB command **runCE**. The command requests the name of the host on which to run the GAMESS experiment. The request to schedule a GAMESS run and the input file are communicated by the CCDB to the compute daemon on the requested compute host via the Compute Monitor. The compute daemon sets up the environment for the GAMESS run on the requested host and then starts the GAMESS process. The process ID (pid) of the process is communicated back to the CCDB, and placed in the appropriate *compProxy* object, providing a mapping from the *compExperiment* instance to the running experiment. When the process terminates, notification is sent to the monitor daemon on the CCDB host. The CPU time used, the termination time and the termination status, and the location of the output files are also communicated to the monitor daemon, which then updates the *compProxy* object.

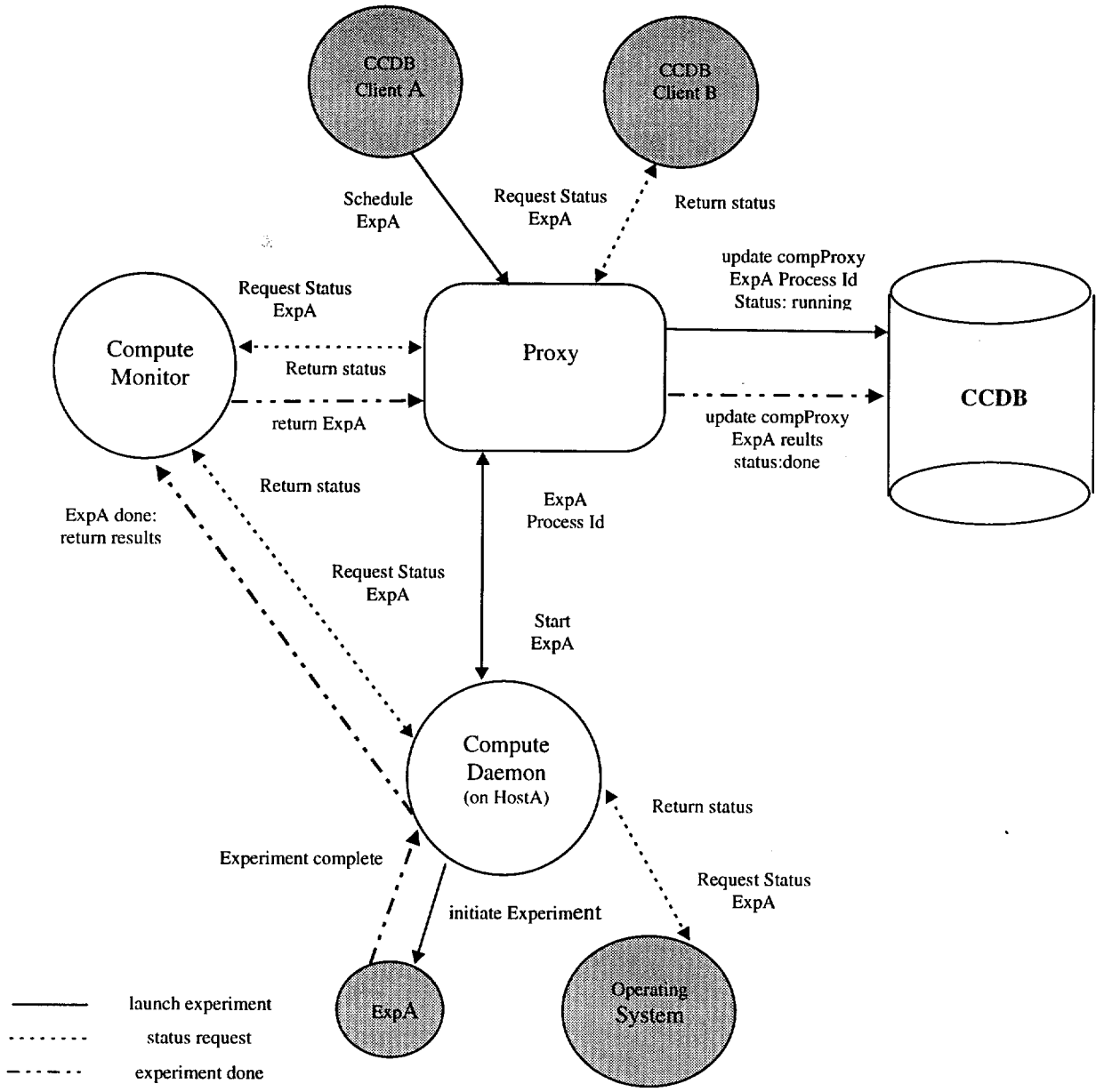

Figure 4.2 The Compute Monitor communications

The monitor daemon manages all communications from the compute daemons to the CCDB. Figure 4.2 sketches the communications between the monitor daemon, the compute daemon, the proxy and the CCDB. A CCDB client, Client A, schedules an experiment, ExpA, using the CCDB command **runCE**. A proxy is created for ExpA, the input file is generated, and the input file name and the compute host name are communicated to the compute monitor. The compute monitor transfers the input file and the compute request to the appropriate compute daemon. The compute daemon starts the experiment and communicates the process id back to the compute monitor, which returns this information to the proxy. Subsequently Client B may request the status of ExpA. The proxy forwards the status request and the process host and id to the computes monitor, which executes a ps on the compute host. Communications between the CCDB and the daemons are effected by sockets. Sockets are more portable than remote procedure calls (rpc).

4.4 Validation

4.4.1 Summary of Implementation

The prototype computational proxy was implemented on a Sun/Unix network consisting of two hosts: `smoked` and `coho`. The hosts `smoked` and `coho` use a file system shared through Sun NFS. The computational package GAMESS could be run on either `smoked` or `coho`. The machine `smoked` was the CCDB host and ran the monitor daemon. A compute daemon was run on both `smoked` and `coho`. The computational proxy implementation used the ObjectStore DML, PERL and Unix system calls, specifically the socket interface. The tools used were the ObjectStore schema designer and the ObjectStore browser. We relied on the ObjectStore browser and printing to stderr for debugging, because at the time of the implementation, the Sun debugger, dbx, had not been modified to work with the ObjectStore DML. The entire implementation for the computational proxy prototype was completed in eight months by a single programmer, working half-time.

4.4.2 Validation

The goal of this study was to implement a prototype of the computational proxy mechanism that would enable us to study the feasibility of implementing a full-fledged computational proxy mechanism based on current technology, and the generalizability of the mechanism to handle other run types and other computational chemistry codes. Towards that end, we outlined criteria to ensure that the prototype fulfilled the key functional requirements of the computational proxy mechanism.

The computational proxy mechanism aims to provide a seamless interface between a domain database and a domain application. Through the process of application registration, the computation proxy "knows" the application-specific information it needs to handle the functions of data input, mission control and data capture for each experiment run of a registered application. Proxy functions must be provided in a networked environment, which is the typical working environment of computational scientists.

For this thesis, we implemented a prototype computational proxy in the CCDB for the ab initio computational chemistry application GAMESS. Prior to the implementation, we identified certain tasks that the prototype should perform to demonstrate at least minimum functionality in the areas mentioned above. These tasks were:

- Adding new experiments to the CCDB and browsing the existing experiments. Adding and browsing experiments are really database capabilities rather than proxy requirements. However, we cannot run an experiment from the CCDB unless data for the experiment resides in the CCDB.
- Running at least one type of GAMESS experiment from the CCDB, which would demonstrate that data input for at least one type of GAMESS experiments could be handled by the prototype computational proxy implementation.
- Running GAMESS on at least one host other than the CCDB host to demonstrate the ability to address the need for providing a networked environment to the chemists.
- Automatically capturing at least one result of a GAMESS run into the CCDB to demonstrate the data capture functionality.

- Implementing at least one mission control query on a running GAMESS experiment to demonstrate the monitoring capabilities.

We have implemented the CCDB commands **loadCP** (to create a new instance of *codePackage*), **loadCE** (to load the input data associated with a *compExperiment*), **CE** (to browse existing *compExperiments*), **runCE** (to schedule a *compExperiment*), **queryCE** (to check on the status of a scheduled *compExperiment*) and **validateCE** (to validate a *compExperiment*). When a user invokes **runCE**, he is presented with the list of experiments in the CCDB. He can opt to run zero, one or more experiments. For each experiment to be run, the name of the compute host, smoked or coho, is requested. The input files are automatically formatted and shipped to the requested compute host and a GAMESS experiment process is started on it. When the GAMESS process terminates, the minimum energy is captured and stored in the CCDB. The user, at any point, can invoke the command **queryCE,** and is presented with a list of unvalidated experiments. He can request the process status of any experiment in the list. If the process corresponding to the selected experiment is still running, the Unix process status request "ps" is forked on the compute host and the information relayed back to the user. If the process has terminated, the termination status and CPU time used, which are stored in the CCDB, are displayed for the user. The user can execute the command **validateCE** to validate an experiment. Validating an experiment means that the user examines the result (the minimum energy in the prototype implementation) and decides if the value is meaningful or not. If he deems it meaningful, the value is stored persistently in the database, otherwise the value is rejected. With the commands **runCE**, **validateCE** and **queryCE** a CCDB user can perform the functions of launching an experiment, have the results automatically captured into the CCDB and monitor an ongoing experiment. The application-specific information required for GAMESS has been stored in the CCDB. The application registration mechanism used for GAMESS can also be used to register other computational chemistry applications with the CCDB. The prototype performs the functions of data input, mission control and data capture within a networked environment, and thus we can view the prototype as a valid computational proxy implementation.

CHAPTER 5

Analysis

In Chapter 4, we described the design and implementation of the prototype computational proxy, for the computational application GAMESS. The aim of this thesis was to implement a prototype proxy to demonstrate the feasibility of the computational proxy mechanism, and to provide a basis for better understanding its usability and extensibility. Section 5.1 deals with the usability of the proxy and analyzes how effectively the prototype met the proposed functional requirements. In Section 5.2 we present an analysis of the feasibility of implementing a full-fledged computational proxy mechanism in an object-oriented database. Section 5.3 discusses the extensibility of the prototype to other computational chemistry applications.

5.1 The Prototype Computational Proxy

A prototype proxy was implemented for the computational chemistry application GAMESS in the CCDB, for a network of homogenous compute hosts. The implementation helped us better understand:

- the efficacy of the prototype computational proxy mechanism in interfacing GAMESS experiments to the CCDB,
- the effort needed to implement the proxy mechanism within the framework of an object-oriented database, and
- the extensibility of the proxy mechanism to other computational chemistry packages.

The prototype proxy mechanism implemented minimal, representative functionality in the areas of data input, mission control and data capture. Additionally, the proxy prototype encapsulated a mechanism for registering applications and network communications. While designing and implementing the prototype proxy, we always kept in mind that:

(i) Although we expected to implement the prototype for the single application GAMESS, the computational proxy mechanism was proposed to address the need to interface an entire class of applications to a central database. Thus we tried to keep prototype design and implementation as independent of the computational package GAMESS as feasible.

(ii) The computational chemist typically works in a distributed computing environment, containing heterogeneous computing hosts. Our implementation was targeted for a network consisting of SUN workstations only. The prototype design is aimed at a network of heterogeneous compute hosts.

(iii) The proxy mechanism aims to interface an entire class of applications to a central database. Thus, as a usability issue, a computational chemist should be able to interface a new computational application to the database without an extensive knowledge of programming or database systems.

In the remainder of this section, we discuss the implementation choices made and the lessons learned while implementing the computational proxy's functionality of application registration and data control within a networked environment.

5.1.1 Application Registration

Application registration is the process by which application-specific information is registered in the CCDB. Information required by the CCDB to interface with an application includes information such as the name of the application, the version, the type of calculations performed, input and output file formats, as well as a mapping from CCDB objects to input and output file data. As a usability issue, this process should not require extensive programming or database

knowledge, so that an expert user of a computational chemistry application can interface that application to the CCDB.

In the prototype implementation, application registration has two parts. Application information such as name of the application and version is loaded into the CCDB through a interactive, user-level command **loadCP**. The input and output formatting information for GAMESS has been provided via functions outside the CCDB. The information was intentionally placed outside the CCDB, since we wished to avoid recompilation or schema evolution problems when adding a new application. There are several disadvantages to this approach. The functions are outside the CCDB, and therefore, cannot be browsed, dumped, backed-up or queried along with the rest of the CCDB data. Furthermore, writing such formatting functions requires programming knowledge, knowledge of the database schema and of course, the application. In addition, the functions must be re-programmed for each application, i.e., we cannot reuse any of the work done in interfacing previous applications. This approach makes it relatively hard to interface a new application using the prototype. Despite these disadvantages, we chose this approach because it was the optimum choice for the prototype implementation. Writing formatting functions is the simplest solution, especially in this case, where the programmer writing the application registration functions is the same as the database implementor. Because the prototype was aimed at interfacing only GAMESS to the CCDB, lack of code reuse was not a deterrent. Any approach to application registration that minimizes programming for the application registrar increases programming complexity for the database implementor. Prior to implementing the proxy prototype, we had no experience in implementing application registration. We reasoned that it would be difficult to design an application registration interface for the non-programmer without any application registration experience on our part, and that the experience gained in the procedural implementation would be useful in designing an interface for the non-programmer.

As is obvious from the discussion above, the prototype application registration implementation did not meet our goal of an interface that would enable a non-programmer to interface a computational chemistry application to the CCDB. However, the exercise of developing even this simplistic, procedural interface for registration of input and output functions provided us with important feedback for refining the functional requirements and

redesigning the application registration function. The experience of the prototype implementation emphasized the following design criteria:

1. From the perspective of the database implementor, the registration mechanism should be general and not require custom programming for each new application.
2. The mechanism should be part of the database, not outside it.
3. Adding a new application should not trigger a schema evolution or even a recompilation of the database.
4. From the perspective of the application registrar, the mechanism must assume minimum programming knowledge on his part, provide an easy mechanism to map input and output data onto CCDB objects, require minimal knowledge of the database schema, and provide a mechanism for specifying multiple, complex input and output file formats and a way of associating the correct output format based on an experiment's input.

From the database implementor's point of view, the application registration mechanism generates data that should be placed inside the CCDB. The best way to store data in an object-oriented database – especially if we want to avoid recompiling the database for every new application – is as database (persistent) objects. Persistent objects are easy to create, can be browsed, queried, dumped and backed-up with the rest of the database. The ObjectStore DML does not support functions at the instance level, so new instances of objects require no new code and hence can create no recompiling problems. The experience of the prototype implementation suggested that an alternative to the procedural approach would be to store the input and file formats as objects in the database. We have tentatively named these objects "templates", since they specify the template to be used to generate the input and parse the output files for each application's experiments. The next question is: how can these templates be created? Logically, the registrar would either create or cause the templates to be created when interfacing an application to the CCDB. Since the registrar need not have any programming experience or knowledge of the database schema, we have to provide him with a way of specifying the template objects. One way of doing this would be to develop a simple, declarative language to specify the input and output formats to the CCDB, and let the CCDB generate the appropriate template objects. Cushing has refined this approach, and with Abel, defined a declarative language, Computational Chemistry Output Language (CCOL) to specify output formats, and a Computational Chemistry Input Language (CCIL) to specify input formats to the CCDB [1, 5].

5.1.2 The Data Control Functions

The data control functions provide the ability to a chemist to launch, monitor, parse and moor a GAMESS experiment from within the CCDB. The data control functionality in the prototype implementation is managed by the CCDB commands **runCE** (data input and data capture), **queryCE** (monitoring). It was fairly straightforward to implement the representative data control functionality once we encapsulated all functionality dealing with communicating over the network using the Compute Monitor. There are two issues that will need attention in any full-fledged computational proxy implementation.

(1) In the prototype implementation, GAMESS experiments could be run on one of two SUN workstations, `smoked`, the CCDB server, or `coho`. Since `smoked` and `coho` share an NFS[4] file system, we did not face the problem of shipping output files over the network from the compute host to the CCDB server. The output files tend to be large and transmitting them over the network using the socket protocol implemented by the prototype can be time consuming. A chemist may wish to have alternative mechanisms for transferring such files, especially as a shared file system across all compute hosts may not be the norm. One option is parsing the output file on the compute host, and having only the relevant data transmitted back to the CCDB server. Having the ability to parse the output on the compute host is even more useful if the compute host is faster and more powerful than the CCDB host or if the underlying database is a distributed database. A more general implementation of the proxy mechanism should support multiple file transfer protocols, provide a mechanism for parsing files on either the compute host or the CCDB server, and consider the impact of distributed databases on the Compute Monitor design.

(2) We have not implemented any monitoring of the computation state itself. Currently, a chemist can monitor a computation by browsing intermediate files written by the experiment. The main point of such monitoring is to terminate a computation that seems to be diverging before it uses a lot of system resources. To provide monitoring at the computation level, we would be looking at intermediate values such as the number of the iteration, the current minimum energy and the energy gradient. We would need a parser, similar to the output

[4] NFS is a registered trademark.

parser, for the intermediate files. The most elegant way to monitor an application would be to have the intermediate results passed directly to an application-specific browser or visualizer that would display the results appropriately. More generally, it would be useful to interface accessory programs such as visualizers and analysis tools to the database in addition to the computational codes. The computational proxy mechanism is a viable way to do so, as long as the accessory programs have the same information model as implemented by the database.

5.1.3 The Compute Monitor

The Compute Monitor encapsulates all network communications between the CCDB and ongoing experiments. The Compute Monitor in the prototype consists of the monitor daemon running on `smoked`, and compute daemons running on both `smoked` and `coho`. All data control requests by the CCDB are routed to the Compute Monitor, which then takes the appropriate action. Both `smoked` and `coho` are SUN workstations, thus the prototype Compute Monitor did not deal with issues arising from porting data to a different architecture or handling system request across diverse operating systems. For example, one of the requests handled by the prototype Compute Monitor implementation is the monitoring function **queryCE**, which causes the Unix process status request "ps" to be forked on the compute host. This implementation of the mission control command **queryCE** works in the prototype since the compute hosts on the network are Unix machines. In a network consisting of hosts running diverse operating systems, interfaces to access process information will vary with the operating system. Thus a more general implementation requires extending the Compute Monitor to include a portability layer. The function of the portability layer would be to map the CCDB commands to the appropriate operating system calls when requesting process information or exercising process control in an environment containing heterogeneous operating systems.

We initially separated the data control functions from the network communications because the data control functions require familiarity with the domain of computational chemistry, while network communications require familiarity with the operating system. This is true even when the functionality of the Compute Monitor is extended to address portability

issues. At the end of the prototype implementation phase, we realized that the Compute Monitor was totally independent of the domain of computational chemistry. By integrating the Compute Monitor into the computational proxy mechanism, we have reduced the complexity of the chemist's environment – without reducing its flexibility or power. The Compute Monitor incorporates functionality that is useful in any domain where multiple applications, running across a network of heterogeneous platforms, need to access data in a central database. Partial solutions to this problem are available. Most databases that have client-server model will handle data layout conversions across heterogeneous platforms. Standards such as OLE [4] and CORBA [33] permit applications to exchange data (one application could be a database application), but for this to be possible, both applications must conform to the protocol. Thus, simply having an OLE or CORBA compliant database application does not address the issue of legacy applications, since legacy applications are not OLE or CORBA compliant.

5.2 The Computational Proxy and OODBMS

We have implemented the CCDB and the prototype computational proxy for GAMESS on the object-oriented database management system (OODBMS) ObjectStore. OODBMSs provide a richer modeling capability than the traditional record-based database systems. During the course of the implementation, we used many of the object-oriented features of ObjectStore. The object-oriented features we used most extensively included the support for types and classes, which enabled us to implement the entities and their relationships in the information model (see Figure 2) relatively easily. For example, a **Chemist**[5] can perform zero or more **Experiment**s. Conversely, an **Experiment** can be performed by one or more **Chemist**s. ObjectStore's support of one-to-one, one-to-many, many-to-many, one-way and bi-directional relationships made the implementation straightforward. Extensibility allowed us to define domain-specific data types, such as molecular orbitals and basis set instance. Support for complex objects enabled us to construct more complex objects based on the user-defined classes. For example, each *compExperiment* in CCDB has a *BasisSetInstance* associated with it. The *BasisSetInstance* specifies the particular basis functions, generated from the *BasisSet* used by the

[5]In this section, we use **bold** to represent entities in the information model, and *italics* for CCDB classes.

compExperiment, for each atom in the subject molecule. We have implemented the *BasisSetInstance* as a list of *AtomBs*, each of which is a list of contractions, where each contraction is a list of primitives. A primitive contains a contraction and a coefficient, both of which are floating point numbers. (See Figure 5.1.) Encapsulation allowed us to associate meaningful behavior with the domain specific types. For example we used encapsulation to associate "equality" methods with class *Chemist* in the CCDB. We used support for inheritance to factor out common behavior of *labExperiment* and *compExperiment* into the superclass *Experiment*. We would have liked to use virtual functions to associate the appropriate formatting functions for a scheduled experiment. As described in Section 4.2, we opted not to do so.

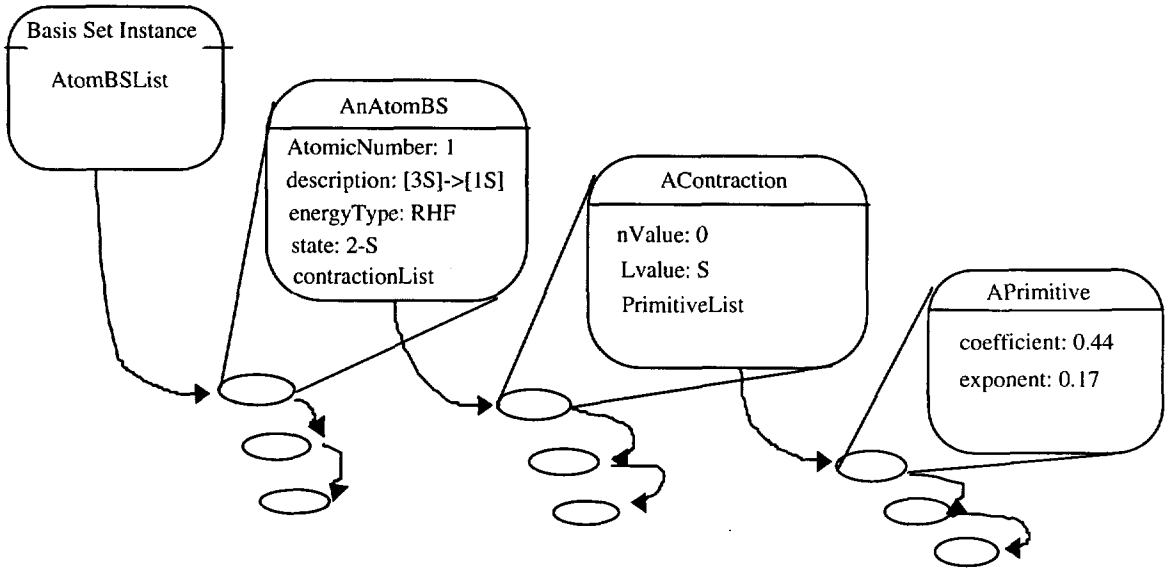

Figure 5.1 Basis Set Instance Implementation

Other object-oriented features did not affect the CCDB implementation much. For example, the ObjectStore DML is computationally complete, but this feature did not play a role when building the prototype. However, computational completeness will be an important consideration when going beyond the legacy applications and considering the development of the

next generation of computational chemistry applications. The object-oriented concept of object identity says that two objects need not be identical even if they have the same values. For the CCDB implementation, this implied that we would have had to specify the database identifier to uniquely identify the **Chemist** when loading in a new **Experiment** performed by that Chemist into the database. In order to correctly associate objects being loaded into the database we provided "equality" methods for every class in the CCDB. The "equality" methods in the CCDB essentially say that two objects can be considered equal if the key data values match – an approach taken from relational databases – and allow us to associate objects in a semantically meaningful way, even without knowing the database identifiers of the objects.

Two ObjectStore tools that we used were the schema designer and the database browser. We entered the entities and their relationships graphically from the information model into the schema designer, and the schema designer created the class definitions in ObjectStore DML (a superset of C++). Method names and signatures could also be entered through the schema designer. The biggest drawback to the schema designer is that subsequent changes made directly to the ObjectStore DML schema file are not reflected in the graphical display. This tool was very useful to us in the initial phases of designing the classes to implement the information model, and provided useful documentation of the CCDB schema. However, we rarely used it in the latter stages of the implementation, because it was hard to keep the schema designer data up-to-date, and because the database browser provided a way to browse the most up-to-date schema. The database browser was indispensable when we were populating the CCDB. Using the browser, we could examine both the schema and the persistent objects in the CCDB. It was easy to follow references to other objects. The biggest drawback of the database browser is that member functions cannot be executed from the browser. Queries written in the ObjectStore query language can be run against the database, but not member functions. The browser, as the name implies, is read-only. Changes cannot be made to the database through the browser. So, we could not use the browser to correct erroneously entered data values, or to populate the database with test cases. The browser also displayed only two digits after the decimal point, which is frustrating when examining floating point numbers associated with scientific experiments.

ObjectStore offers features beyond those traditionally associated with database management systems and the object-oriented features discussed above. These features include

support for long transactions (check-in, check-out), workspaces, configuration management and versioning. These features collectively permit the user to build his own custom view of an evolving, shared database. While we did not use these features in the prototype proxy implementation, we expect that any generalized proxy implementation will rely on these features to provide the computational chemist with the ability to configure his environment optimally. For example, a computational chemist working on the ethane molecule may wish to check out all ethane experiments into his workspace, run his own experiment(s) on ethane, and eventually check in the validated results.

The implementation of the prototype computational proxy was obviously influenced by the choice of ObjectStore as the underlying OODBMS. Would we have been successful in implementing a valid prototype proxy if we had chosen another OODBMS? Although there exists no general data model for object-oriented databases [27], a database system must display a minimum set of object-oriented features to be classified as an OODBMS . (See Section 2.1.) These features may be implemented differently in different OODBMS, and the same features may have different flavors in different systems. For example, while GemStone [28] supports inheritance, ObjectStore supports multiple inheritance. Both ObjectStore and GemStone provide encapsulation. However, in ObjectStore behavior is encapsulated with the class, and no mechanism exists for associating methods or code with an instance. In GemStone, code "blocks" can be associated with an instance.

The object-oriented features of ObjectStore that we relied on in developing the CCDB and computational proxy were support for complex objects and classes, extensibility, encapsulation, and, to a lesser extent, inheritance. Other features that we used during the implementation were the query language, the database browser and the schema designer. Of course, the object-oriented features are common to all object-oriented databases. A query language is important, and most OODBMS offer some support for a query language. There is some effort in the field to develop a standard for a query language, similar to SQL, called OQL, for OODBMS. Database browsers are another important tool supplied by most commercial OODBMS vendors. Thus we believe that OODBMS in general provide sufficient infrastructure to support the computational proxy mechanism. However, due to the high degree of variability in the implementation of the object-oriented features and the tools provided by various OODBMS,

we do not expect the implementation design of the prototype proxy to be directly portable from ObjectStore to any other OODBMS. For example, GemStone supports instance-level methods, which could possibly have simplified the implementation of the application registration function. GemStone also has a visual programming interface (GeODE) that would have make it easier to develop a graphical user interface, something we did not even attempt in ObjectStore. Alternatively, schema evolution is offered by many OODBMS vendors, and the schema evolution support is becoming more sophisticated. So in an OODBMS like ITASCA [3], which has good support for schema evolution, we may have done the application registration differently.

Object-oriented database systems represent a relatively new technology, especially in the commercial world. They do not yet provide tools of the range and maturity of those available from relational database vendors. There are several features we would have liked to see in ObjectStore, in particular, and OODBMS in general. These include a uniform query language, more sophisticated support for schema evolution and data migration[6], support for loading bulk data from files, enhanced support for communications between database and operating systems, and a well-defined, formal interface through which external processes can access the database.

5.3 Extending the Prototype Proxy to include other Computational Chemistry Applications

The prototype computational proxy was designed to interface the computational chemistry application GAMESS to the CCDB. The computational proxy mechanism itself was proposed to address the issue of interfacing an entire class of applications to a central database. The prototype implementation provides us with valuable insight into the effort required to extend the proxy mechanism to another computational chemistry application. Below we outline the steps required to interface another computational chemistry application, taking Gaussian as an example, and give an assessment on how difficult or easy that step would be.

1. Install Gaussian on either `smoked` or `coho`, or both.

[6]ObjectStore version 2.1 provides an API to support schema evolution and data migration.

2. Create the *codePackage* instance in the CCDB to represent the computational chemistry application Gaussian. In the current CCDB, the *codePackage* instance for Gaussian can be created by invoking the interactive command **loadCP**, which prompts the user for the relevant information. We have created *codePackage* instances for several computational chemistry applications in the CCDB, including Gaussian.
3. Next, we have to write the input and output formatting functions for the Gaussian calculation types to be supported to the CCDB.

   Steps (2) and (3) together represent application registration. Writing the input formatting function for a similar subset of input types as supported by GAMESS is relatively easy. GAMESS and Gaussian have similar input and output formats, for historical reasons, and the input formatting function for Gaussian can be closely modeled on the one for GAMESS. (See Appendix B.) Then, the case statement called from *codePackage::fmtBSI* has to be modified to include the case for Gaussian experiments (below):

```
codePackage::fmtBSI(CompProxie* cp, FILE* )
{
      callit(cp, fp);
}

extern  void fmtBSIGAM(CompProxie*, FILE*);
extern  void fmtBSIGauss(CompProxie*, FILE*);

void callit(CompProxie* cp, FILE* fp) {
  if (strcmp(cp->name, "GAMESS") {
     fmtBSIGAM(cp, fp)
  }
   if (strcmp(cp->name, "Gaussian") {          <<<
     fmtBSIGauss(cp, fp)                       <<<  (case added)
  }                                            <<<
}
```

The Gaussian formatting function fmtBSIGauss and the case function would have to be recompiled, and these new object files linked with the CCDB code. Also, an output parser would have to be written. Again, if we wish to extract only the minimum energy value, as we do for GAMESS (see Appendix B), the same PERL script can be used. In retrospect, we could have placed the case statement in the

codePackage::fmtBSI itself, and placed *codePackage::fmtBSI* in a source file by itself, to achieve the same level of recompilation.

4. Modify the command **runCE** to pass the name of the computational chemistry application to the compute daemon.
5. Modify the compute daemon to fork either the GAMESS or Gaussian process, based on the application name passed to it from **runCE**, with the appropriate environmental setup. (Appendix D lists the environment encoding and the forking of the new GAMESS experiment.)

In a more general implementation, we would modify **runCE** to always pass the name of the computational application to the compute daemon. Correspondingly, we would modify the compute daemon to read an external file and map the application name passed in by **runCE** to the path of the executable script or program that sets up the environment for that application. The changes required to generalize **runCE** and the compute daemon are programmatically minor. But they result in two major advantages: (i) the program for setting up the environment and forking the application process can be any executable file such as a compiled C program, a shell script or a PERL script, (ii) computational applications can be added and removed from a compute host simply by changing the external file mappings, without having to change the compute daemon or **runCE** code.

Assuming the more general implementation of **runCE** and the compute daemon, when adding a new application, we would have to (i) create the codePackage instance, (ii) write the input formatting function, (iii) write the output parser, (iv) write the script that sets up the compute environment and forks the process for the application, and (v) update the external mappings files on the compute hosts supporting the new application. We do not need to change the CCDB schema (though we do have to relink the CCDB code), and we do not have to change the CCDB commands such as **loadCP**, **runCE** or **queryCE**. Nor do we need to change the code for the Compute Monitor or the compute daemons. The table below summarizes the changes required on adding a new application:

CCDB	
CCDB schema	No code changes, re-linking required.
CCDB commands	No code changes.
CCDB database	Create a new instance of *codePackage*.
Computational Proxy	
Application registration	Has to be done anew each time, since there is no code reuse in our implementation.
Data input	No changes (the data input to the experiment is done by calling the input formatting functions written at application registration)
Data capture	No changes
Mission Control	No changes

The way we chose to implement the application registration mechanism is clearly very application-specific, and requires reprogramming for every new application, and possibly for every new version of a computational chemistry application. Moreover, writing the application-specific input and output formatting functions requires knowledge of both the CCDB schema and the computational application. The input formatting function for the limited subset of calculation types supported by the CCDB (Appendix B) is about 60 lines of code. We estimate that a GAMESS formatting function that supports most input options and more robust error checking would be about a 600-1000 lines of code. The output parsing is much more complex because of the variability of the output file formats. We have used PERL to parse the output for the minimum energy. At the time of developing the output parsing script, the idea was to provide a script for each observable property of interest. Thus, associated with each application would be a cluster of PERL scripts, probably one per output property of interest. At the termination of an experiment the appropriate set of PERL scripts would be called, based on the calculation type and the input parameters. It is possible that some of the parsing scripts are portable across applications. However, we did not investigate in detail the development of application-independent, property-specific PERL scripts. (See Section 5.1.1 for a discussion of templates as an avenue to explore for creating a less application-specific way to register applications.)

CHAPTER 6

Conclusions

This thesis presents a proof of concept of the computational proxy mechanism proposed by Cushing to address the problem of interfacing legacy applications to a central domain database. We implemented a prototype computational proxy for the ab initio computational chemistry application GAMESS, interfacing it to the prototype computational chemistry database, CCDB. The CCDB and the prototype proxy were successfully implemented using C, PERL and the OODBMS ObjectStore from Object Design, Inc., on SunOS (Unix-based) SparcStation. The prototype proxy demonstrated representative functionality in the areas of application registration, data control and network communications. A computational chemist using this mechanism can launch, control, parse and moor GAMESS experiments from the CCDB.

Information relating to a new application to be interfaced to the CCDB is conveyed to the CCDB through the process of application registration. Application-specific information includes data such as the name of the application, its version number, the calculations supported, default calculation units and input and output file formats. The prototype proxy takes a programmatic approach to making formatting information available to the CCDB. This approach is viable for registering a small number of applications with moderately complex input and output formats to the CCDB, but requires the application registrar to know both programming and the database schema. Furthermore, this approach does not scale well to the computational chemistry domain, where new computational chemistry applications are being developed and newer versions of existing applications released. A more general solution must avoid the custom programming required for interfacing each new application and minimize the programming knowledge required by the application registrar. An alternative approach to ours would be to store the formatting information as database objects and provide a declarative language to the

application registrar for creating these objects. Cushing [5] and Abel [1] have further investigated this approach, implementing a declarative language CCOL for creating database objects, called templates, that store formatting information.

The data control functions provide the computational chemist with the ability to launch, control, parse and moor computational chemistry experiments from the CCDB. Separating all network communications required to support data control across a network into the Compute Monitor made implementation of the data control functions themselves straightforward. There are two issues that require further consideration for a more general implementation of the computational proxy mechanism:

(1) The output files generated by the computational chemistry applications tend to be large. Transferring them over the network using the socket protocol implemented in the prototype would be slow. We did not address this problem in the prototype, since the prototype was implemented on a network of SUN/UNIX workstations with a shared NFS file system. A more general implementation will probably have to support multiple file transfer protocols. Alternatively, a chemist may choose to have the output file parsed on the compute host, and have only the relevant data transferred back to the compute server. Providing the chemist with this ability requires that parsing functions be invoked on the compute host. Since the compute hosts may have different operating systems, the parsing functions must be portable across diverse operating systems. Note that the parsing functions cannot be member functions, since in ObjectStore, member functions always execute on the ObjectStore client, and the programmer cannot request that the functions execute on another host.

(2) In the prototype implementation, we did not address the issue of monitoring intermediate stages in a computation. Currently, a user monitors the intermediate stages of a computation by browsing the intermediate files written by the application. To provide equivalent functionality, we would have to provide parsing functions for the intermediate files. Conceptually, parsing intermediate files requires parsing functions equivalent to those needed for parsing the output files. In the future, chemists may wish to use visualizers or plotters to display both the intermediate and final results.

(3) Standards for data exchange between applications do not address the issue of data exchange for legacy applications. We think the computational proxy can provide a mechanism for

applications in a domain to exchange data. The data model of the CCDB acts as the "standard" format, and the formatting information registered with the CCDB at application registration provides a means of converting data from the CCDB format to the application format.

The Compute Monitor encapsulates the communications between the CCDB and compute hosts over the network. The prototype Compute Monitor was implemented on a network of homogeneous operating systems, with a shared NFS file system. However, this is not representative of the typical working environment of the computational chemist, which includes a network of heterogeneous operating systems, only some of which may have a shared file system. We believe the issue of heterogeneous operating systems can be addressed by adding a portability layer to the Compute Monitor. The Compute Monitor is totally independent of the domain of computational chemistry. Its function is to insulate the user form the complexities of the network, and operating systems in the more general case, without reducing the user's computing power or flexibility. The functionality represented by the Compute Monitor is applicable in any area where several applications across a network need to access data from a central database because the Compute Monitor, along with the data control functions, provides a mechanism to control processes utilizing data from a central database. We believe that in the future this functionality will be available from database vendors or third party software developers.

The prototype proxy was implemented using the commercially available OODBMS ObjectStore. We believe that object-oriented databases offer a rich enough modeling paradigm to model scientific data types. ObjectStore offered tools, such as the schema designer and the database browser, that are not intrinsically object-oriented themselves, but made the task of developing the CCDB a little easier. We expect that as OODBMS become more widely used, a wider range of tools and accessory applications will become available to the developer. Object-oriented databases also offer features such as versioning and configuration management, long transactions and workspaces that we did not use in the prototype proxy development.

The focus of this implementation was a proof of concept of the proxy mechanism. We have demonstrated that we could implement a prototype computational proxy with representative

functionality in the areas of application registration, data control and network communications. Using the prototype proxy, we could launch, control, parse and moor GAMESS experiments on a network of SUN/Unix workstations. The implementation of the proxy gave us important feedback, specially in the area of application registration, leading to a specification of a more general way of registering new applications using templates, the Computational Chemistry Input Language (CCIL) and the Computational Chemistry Output Language (CCOL). We have sketched how the prototype could be extended to another computational chemistry application, such as Gaussian. We believe that the proxy implementation demonstrates that computational proxies offer a viable solution to the problem of interfacing computational chemistry applications to the CCDB.

Bibliography

[1] Abel, D. *The PCL: An Implementation of the Computational Chemistry Output Language*. Master's thesis, Dept. of Computer Science, Portland State University, to be published, 1995.

[2] Atkinson, M. et al. "The Object-Oriented Database Manifesto". In *Deductive and Object-Oriented Databases,* Elsevere Science Publishers, Amsterdam, Netherlands, 1990.

[3] Banerjee, J. et al. "Data Model Issues for Object-Oriented Applications". In Stanley B. Zdonik and David Maier, editors, *Readings in Object-oriented Databases,* Morgan Kaufmann Publishers, San Mateo, California, 1990.

[4] Brockschmidt, Kraig. *Inside OLE 2*. Microsoft Press, 1993.

[5] Cushing, J.B. *Computational Proxies: An Object-based Infrastructure for Computational Science,* Ph.D. thesis, Dept. of Computer Science and Engineering, Oregon Graduate Institue, 1995.

[6] Cushing, J.B. "Computational Proxies: Interfacing legacy applications to scientific databases". A position paper for *OOPSLA '92 workshop on Applications of SmallTalk in Scientific and Engineering Computation.* October 1992.

[7] Cushing, J.B., D. Maier and M.Rao. *Computational Chemistry Database Prototype.* Technical Report CS/E-92-002, Oregon Graduate Institute of Science and Technology, Beaverton, OR, 1992.

[8] Cushing, J.B., et al. "Object-Oriented Database Support for Computational Chemistry". *Sixth International Working Conference on Statistical and Scientific Database Management (SSDBM),* 58 June 1992.

[9] Davidson E.R., and D. Feller. "Basis set selection for molecular calculations". *Chemical Review*, **86**, 681, 1986.

[10] Dunning, T.H., Jr. "Gaussian Basis Functions for Molecular Caluclations. 1. Contraction of (9s,5p) Atomic Basis Sets for the First-Row ". *J. Chem. Phys*, ***53***, 2823, 1970

[11] Dupuis, M., D. Spangler, and J.J. Wendoloski, *NRCC Software Catalog*, Vol. 1 Program QG01 (GAMESS), 1980. This program was heavily modified by M.W. Schmidt and S.T. Elbert.

[12] Dupuis, M. *Hondo-8 Users' Guide.* IBM Center for Scientific and Engineering Computations, Kingston, NY, 1990.

[13] ElMasri, R. and S.B.Navathe. *Fundamentals of Database Systems*. The Benjamin/Cummings Publishing Company Inc., 1989

[14] Feller, D., and E.R. Davidson. "Basis Sets for ab initio Molecular Calculations and Intermolecular Interactions". In K.B. Lipkowitz and D.B. Boyd, editors, *Reviews in Computational Chemistry*, 1, VCH, New York, 1990.

[15] Feller, D., K. Schuchardt, et al. Computational Chemistry Input Advisor (CCIA). Personal communication, August 1993.

[16] French, J.C., A.K. Jones and J.L. Pfaltz. *NSF Scientific Database Management Workshop (Final Report).* Technical Report TR-90-21, University of Virginia, Charlottesville, VA, August 1990.

[17] French, J.C., A.K. Jones and J.L. Pfaltz. *NSF Scientific Database Management Workshop (Panel Report and supporting material).* Technical Report TR-90-22, University of Virginia, Charlottesville, VA, August 1990.

[18] Frisch, M. *Gaussian 90 User's Guide and Programmer's Reference.* Gaussian Inc., Pittsburg, PA, 1990.

[19] *GAMESS User's Guide.* Department of Chemistry, North Dakota State University, Fargo, ND. 1990.

[20] Hansen, David, Brian Kennedy and William Petroske. *Computational Chemistry Information Model: GemStone Implementation.* A project report for Object-Oriented Database Systems, Dept. of Computer Science and Engineering, Oregon Graduate Institute. Spring 1991.

[21] Hehre, W.J., R.F. Stewart and J.A. Pople. "Self-Consistent Molecular Orbital Mehtods 1. Use of Gaussian Expansions of Slater-Type Orbitals", *J. Chem. Phys.* **51**, *2657,* 1969.

[22] Hornick, M.F. and S.B. Zdonik. "A Shared, Segmented Memory System for an Object-Oriented Database". *ACM Transactions on Office Information Systems*, **5**:1, 1987.

[23] Keller, Tom, Garland Bayley and George Moberly. *POSTGRES CCDB Schema Description.* A project report for Object-Oriented Database Systems, Dept. of Computer Science and Engineering, Oregon Graduate Institute. Spring 1991.

[24] Labanowski, J.K. *Simplified introduction to ab initio basis sets. Terms and notations.* Computational Chemistry Electronic Bulletin Board (CHEMISTRY-REQUEST@osc.edu), Ohio Supercomputer Center, 1224 Kinnear Rd, Columbus, OH 43212-1163.

[25] Lamb, C., G. Landis, J. Orenstein, and D.Weinreb. "The ObjectStore Database System". *CACM*, **34(10)**, 50, October,1991.

[26] Lewis, Jeff and Brian Hansen. *Report on the Implementation of a Prototype Computational Chemistry Database using ENCORE.* A project report for Object-Oriented Database Systems, Dept. of Computer Science and Engineering, Oregon Graduate Institute. Spring 1991.

[27] Maier, D. "Why Isn't There an Object-Oriented Data Model?" In G.X. Ritter, editor, IFIP 11th *World Computer Congress - Information Proceesing '89.* Elsevier Science, August-September 1989.

[28] Maier, D. and J. Stein. "Development and Implementation of an Object-Oriented DBMS". In P.Wegner and B. Shriver, editors, *Research Directions in Object-Oriented Programming*, 355, MIT Press, 1987.

[29] *ObjectStore User's Manual.* Object Design, Inc., Burlingame, MA, 1991.

[30] O'Brien, P., B. Bullis, and C. Shaffert. "Persistent and Shared Objects in Trellis/Owl". In U. Dayal, and K. Dittrich, editors, *Proceedings of the International Workshop on Object-Oriented Databases*, Pacific Grove, CA, September 1986.

[31] Paton, N.W., and P.M.D. Gray. "Identification of Database Objects by Key". *Advances in Object-Oriented Database Systems*, **334**, 280, 1988.

[32] Rao, Meenakshi, Robert Kelley and Judy Cushing. *Object-Oriented Database Class Project: ObjectStore.* A project report for Object-Oriented Database Systems, Dept. of Computer Science and Engineering, Oregon Graduate Institute. Spring 1991.

[33] Rosenberry, W., D. Kenney, and G. Fisher. *Understanding DCE.* O'Reily & Associates, Inc., 1992.

[34] Rowe, L.A. and M.R. Stonebraker. "The POSTGRES Data Model". In Stanley B. Zdonik and David Maier, editors, *Readings in Object-oriented Databases*, 461, Morgan Kaufmann Publishers, San Mateo, California, 1990

[35] Strousstrup, B. *The C++ Programming Language.* Addison Wesley, 1991.

[36] Wall, L. and R.L. Schwartz. *Programming Perl.* O'Reilly & Associates, Inc., Sebastopol, CA, 1990.

APPENDIX A

The CCDB Schema

We list below the schema for the CCDB (including the class compProxie).

```
/****************************** DICT.HH ************************/
/*
 * Object Oriented Database Systems
   Computational Chemistry Dictionary Schema
   by Meenakshi Rao
     Judy Cushing

*/

#include <sys/types.h>
#include <string.h>

struct Date {
   short yyyy, mm, dd;
   Date() : mm(0), dd(0), yyyy(0) { }
   friend istream& operator>>(istream& s, Date &d);
};

struct Time {
   short hh, mn;
   long  ms;
   Time() : hh(0), mn(0), ms(0) { }
   friend istream& operator>>(istream& s, Time &d);
};

struct DateTime {
   Date date;
   Time time;
};

/*istream& operator>>(istream& s, Date &d)
{
        char c;
        return s >> d.mm >> c >> d.dd >> c >> d.yyyy;
}
*/


enum boolean {FALSE=0,TRUE=1,UNUSED=2,NOTaBOOL=3};
enum EnergyType {RHF,UHF,NONE};
enum Lvalue    {S=0,P=1,D=2,F=3,G=4};
```

```
#define LAB  1
#define COMP 2
#define EMPTY -2        // abnormal return values from getposint
#define ALPHA -1        // abnormal return values from getposint

#define MAXL 4
/*                      maximum L-value supported by any basis set and code */
#define MAXROW 4
#define MAXFILE 4
/*                      maximum number of parts (files) for any basis set */
#define MAXATOM 40
/*                      maximum number of atoms supported for any basis set */

#define defaultDB "/ogi/students/mrao/scidb/db/scidb"



/************************* CCDB.HH ********************************/

/*
 *  Object Oriented Database Systems
    Computational Chemistry Schema Design for Object Store
    by Meenakshi Rao
       Judy Cushing
       Don Abel
* Schema header file for ossd file 'prj'.
 * Produced by ossd on Tue May  7 17:42:40 1991
 *
 * Classes and their forward declarations are written
 * in base class to derived class order, as is necessary
 * for compilation.
 */

/*
 * Forward class declarations
 */

extern database *db;

#include "dict.hh"

class Chemist;
class Experiment;
class LabExperiment;
class LabApparatus;
class CompExperiment;
class CompProxie;
class Property;
class MolecularOrbital;
class OrbitalLabel;
class Symmetry;
class BasisSet;
class BasisSetInstance;
class LevelofTheory;
```

```
class CodePackage;
class Molecule;
class Atom;
class PersonalCE;

struct MOCoefficient;
int getposint(char*);
static char buffer[1024];

/*
 * Class declarations
 */

class Chemist {
 /* A chemist performs experiments. */

 public:

   persistent<db> os_Set<Chemist*> extent;
   os_Set<Experiment*> performs inverse_member isPerformedBy;
   os_Set<BasisSet*> authors inverse_member isAuthoredBy;

   int        id;
   char* firstname;
   char* lastname;
   char* address;
   char* email;

   Chemist (int Chemist_id) {
          id = Chemist_id;
      extent.insert(this);
          }
   ~Chemist () {
      extent.remove(this);
          }
   void terseDisplay();
   int  author_of(char*);
   int  perform_exp(char*);
   static Chemist* loadChemist();
   Chemist* chem_equals(char* namstr);
};


class Experiment {
 /* An experiment is either computational or laboratory. */

 public:
//  persistent<db> os_Set<Experiment*> extent;

   os_Set<Chemist*> isPerformedBy inverse_member performs;
   os_Set<Property*> produces inverse_member isProducedBy;
   Molecule* hasAsSubject inverse_member isSubjectOf;

   int                id;
```

```
  char* name indexable;
  char* citation;
  DateTime      begun indexable;
  DateTime      completed;
  char* site;

  virtual int WhatAmI() = 0;

  Experiment () {
        }
  void terseDisplay();
  static void loadExperiment(Experiment*);

};


class LabExperiment : public Experiment {
 /* A LabExperiment confirms a computational experiment. */

 public:

  persistent<db> os_Set<LabExperiment*> extent;
  LabApparatus* isConductedOn;
  os_Set<CompExperiment*> confirms inverse_member isConfirmedBy;

  virtual int WhatAmI() {
        return LAB;
        }
  LabExperiment () : Experiment() {
        id = extent.cardinality() +1;
        extent.insert(this);
        }
  ~LabExperiment(){extent.remove(this);}
  static LabExperiment* loadLabExperiment();
};


class LabApparatus {

  public:

   char* instrument;
   /* information about calibration for specific instruments */
};


class CompExperiment : public Experiment {
 /* A Computational Experiment uses a Code Package and
                Basis Set to some Level of Theory. */

 public:

  persistent<db> os_Set<CompExperiment*> extent;
```

```
  BasisSet* usesBS ;
  LevelofTheory* isTakenTo ;
  CodePackage* usesCP ;
  MolecularOrbital* calculates inverse_member isCalculatedBy;
  LabExperiment* isConfirmedBy inverse_member confirms;
  CompProxie* isRunning inverse_member isProxieFor;

  char*     runType;
  float  cpuTime;
  float  elapsedTime;
  float  escf;
  float  esdci;

  BasisSetInstance* generateBSI();
  virtual int WhatAmI() {
        return COMP;
        }
  CompExperiment () : Experiment() {
        id = extent.cardinality() +1;
        extent.insert(this);
        }
  ~CompExperiment () {
              extent.remove(this);
                        }
   static CompExperiment* loadCompexp( );

};


class CompProxie {
 /* A Computational Proxie is a standin for an
        executing computational experiment */

 public:

  BasisSetInstance* BSI ;
  CompExperiment* isProxieFor inverse_member isRunning;
  static CompProxie* loadCP(CompExperiment* );

  int   pid;
  int   status;
  char* host;    //name of host on which experiment is running
  char* basisFileName;

};


class MolecularOrbital {
 /* A molecular orbital is the output of a computational experiment.  */
 public:

  CompExperiment* isCalculatedBy inverse_member calculates;
  os_List<OrbitalLabel*> anOrbitalLabel;
  os_List<Symmetry*> aSymmetry;
```

```
};


class OrbitalLabel {
 public:
   int    atomicNumber;
  char*  label;
};


class Symmetry {

 public:
  char*  symmetryLabel;
  float  orbitalEnergy;  // eigenvalue for ... matrix
  float  occupancy;
  os_List<MOCoefficient*> anMOCoef;
};


struct MOCoefficient {
 float coef;
};


class Property {
 /* A property is a property/unit/value triple. */

 public:

  persistent<db> os_Set<Property*> extent;
  Experiment*  isProducedBy inverse_member produces;

  char*  name;
  char*  unit;
  float  value;

  static Property* loadProperty();
  Property () {
        extent.insert(this);
        }
  ~Property () {
        extent.remove(this);
        }
};


class BasisSet {
 /* A basis set is used in a computational experiment. */

 public:

  persistent<db> os_Set<BasisSet*> extent;
  os_Set<Chemist*> isAuthoredBy inverse_member authors;
```

```
  char* name;
  short nParts;
/*                          how many parts to this basis set?
                            -> number of files describing basis set */
  char* fileName[MAXFILE];
  short maxL[MAXROW];
/*                          does this bs support S(0),P(1),D(2),F(3)
                            for each row of the perodic table      */
  boolean   spherical[MAXL];
/*                          for each type (SPDF),
                            does this bs support pure spherical components
                            of the cartesian gaussians, e.g., 5 component D's */
  boolean         atoms[MAXFILE][MAXATOM];
/*                          for each bs part (file), which atoms supported by bs */
  short maxC[MAXL][MAXROW];
/*                          max no. of gaussians in a contraction for each L type,
        for each row in the periodic table */

  void terseDisplay();
  void verboseDisplay();
  static BasisSet* loadBS();

  BasisSet (char* bsname) {
        name = new(db) char[ strlen( bsname) + 1 ];
        strcpy(name, bsname);
        extent.insert(this);
        }

  ~BasisSet () {
        delete name;
        extent.remove(this);
        }
};

struct Primitive
 {
 double coefficient;
 double exponent;
 };

struct Contraction
 {
 Lvalue L;
 os_List<Primitive*> primitives;
 };

struct atomBS{
 short          atomicNumber;
 char*          description;
 EnergyType     energyType;
 double energy;
 char*          state;
 os_List<Contraction*> contractions;
```

```
};

class BasisSetInstance {
 /* Generated for a particular comp exp,
        using molecule and code
        reading basis set library (ascii files)
 */
 public:

 os_List<atomBS*> AtomBSList;
  };


class PersonalCE {
 /* Users can define their own sets of Comp Exp */

  public:

  persistent<db> os_Set<PersonalCE*> extent;
   char*      name;
   os_Set<CompExperiment*> my_set;

   PersonalCE (char* s){
            name= new(db) char[strlen(s)+1];
            strcpy(name,s);
            extent.insert(this);
   }
  ~PersonalCE () {extent.remove(this);}

   };


class LevelofTheory {
 /* A computational chemistry experiment is taken to a level of theory. */


 public:

   persistent<db> os_Set<LevelofTheory*> extent;
   char* name;

   LevelofTheory (){
        extent.insert(this);
        }
   ~LevelofTheory () {extent.remove(this);}

};


class CodePackage {
 /* A code package is used by a computational experiment. */

 public:
```

```
    persistent<db> os_Set<CodePackage*> extent;
    char* name;
    char* codeVersion;
    char* computer;
    char*     compilerVersion;
    char* fmtBSIFcn;
    DateTime      Available;
    DateTime      Archived;
    short  maxL;
/*                          does this code support S(0),P(1),D(2),F(3) */
    boolean    spherical;
/*                          does this code support pure spherical components
                            of the cartesian gaussians, e.g., 5 component D's */
    short  maxS;
    short  maxP;
    short  maxD;
    short  maxF;
    short  maxG;
/*                          max no. of gaussians in a contraction for each L type */

/*  table hosts
                            on which hosts does this ccc run?
                            the table should include architectural platform, and
                            name of the host                                    */

    void fmtBSI(CompProxie*, FILE*);
    static CodePackage* loadCP();

    CodePackage () {
          extent.insert(this);
          }
    ~CodePackage() {extent.remove(this);}
};


class Molecule {
 /* A molecule is the subject of an experiment. */

 public:
   persistent<db> os_Set<Molecule*> extent;

   os_Set<Experiment*> isSubjectOf inverse_member hasAsSubject;
   os_Set<Atom*> hasAtoms ;

   int     id;
   char*  name;
   char*  formula;
   char*  symmetry;  // symmetry group;  default is no symmetry C1

/* The class Symmetry is used as a member of class MolecularOrbital
    to store the symmetry-labelled columns of the molecular orbital matrix.
*/

   Molecule (int i){
```

```
        id=i;
        extent.insert(this);
        }
  ~Molecule () {extent.remove(this);}

  static Molecule* loadMolecule();
  os_List<int*> atom_list();

};


class Atom {
 /* An atom is a component of a molecule. */

 public:

  char* name;
  int           Z;
  float  mass;
  float  charge;
  float  x;
  float  y;
  float  z;

  Atom (char* str){
        name= new(db) char[strlen(str)+1];
        strcpy(name,str);
        }
  ~Atom () {}
};


class PeriodicTable;

struct PeriodicElement {
   int  atomicNumber;
   char* atomicSymbol;
   char* atomicName;
};


class PeriodicTable {

public:
persistent<db> PeriodicTable PT;

 PeriodicElement* Ptable[100];

/*  Yes, we know the periodic table has more elements than that!
    But, they are not stable, and there are no basis sets available for them.
*/

 PeriodicTable()
 {
```

```
    int i;

/*
    HANDLE (err_deref_transient_pointer)
      if (this != NULL)
        delete this;
    EXCEPTION
      cout << "caught exception: err_deref_transient_pointer\n"
           << "no dictionary to delete\n";
    END_HANDLE
*/
    for ( i = 0; i <= 100; i++ )
    {
      PTable[i] = new(db) PeriodicElement();
    }
  }
 ~PeriodicTable()
 {
   int i;
   for ( i = 0; i <= 100; i++ )
   {
     if (strlen(PTable[i]->atomicSymbol) > 0)
           delete PTable[i]->atomicSymbol;
     if (strlen(PTable[i]->atomicName) > 0)
           delete PTable[i]->atomicSymbol;
         delete PTable[i];
   }
 }
 char* AtomicName(int AN)
 {
   return PTable[AN]->atomicName;
 }
 char* AtomicSymbol(int AN)
 {
   return PTable[AN]->atomicSymbol;
 }
 int   AN(char* Name)
 {
   int len = strlen(Name);
   for (int i=1;i<=100;i++)
     if (strncmp(Name,PTable[i]->atomicName,len) == 0)
       return PTable[i]->atomicNumber;
   cout << "Sorry, atomic Name not found in Periodic Table:  "
        << Name
        << "\n";
 }

 int  SymboltoAN(char* Symbol)
 {
   int len = strlen(Symbol);
   for (int i=1;i<=100;i++)
     if (strncmp(Symbol,PTable[i]->atomicSymbol,len) == 0)
       return PTable[i]->atomicNumber;
```

```
    cout << "Sorry, atomic Symbol not found in Periodic Table:  "
         << Symbol
         << "\n";
    return 0;
  }
};


/*  Below are the "root" objects or entry points into the database.
    For easy access from the browser, for ease in debugging, and
    the convenience of using the ObjectStore queries, we've made
    all interesting objects persitent.
*/
persistent<db> os_Set<Chemist*> Chemist::extent = os_Set<Chemist*>();
persistent<db> os_Set<CompExperiment*> CompExperiment::extent =
                                    os_Set<CompExperiment*>();
persistent<db> os_Set<LabExperiment*> LabExperiment::extent =
                                    os_Set<LabExperiment*>();
persistent<db> os_Set<BasisSet*> BasisSet::extent = os_Set<BasisSet*>();
persistent<db> os_Set<LevelofTheory*> LevelofTheory::extent =
                                    os_Set<LevelofTheory*>();
persistent<db> os_Set<CodePackage*> CodePackage::extent = os_Set<CodePackage*>();
persistent<db> os_Set<Molecule*> Molecule::extent = os_Set<Molecule*>();
persistent<db> os_Set<Property*> Property::extent = os_Set<Property*>();
persistent<db> os_Set<PersonalCE*> PersonalCE::extent = os_Set<PersonalCE*>();

persistent<db> PeriodicTable  PeriodicTable::PT = PeriodicTable();
```

APPENDIX B

The Formatting Function for GAMESS

Below is a listing of the input formatting function fmtBSIGAM, that we wrote for formatting the input for the subset of GAMESS runs supported by the GAMESS computational proxy:

```
void fmtBSIGAM(CompProxie* comprox, FILE* fp)
{
  Atom*  atom;
  atomBS* abs;
  Contraction* cont;
  Primitive*  prim;
  char  symm[20];
  char  lval[2];
  int i, mplevel;


  fprintf(fp,
  " $CONTRL TIMLIM=999.0 MEMORY=2000000 $END \n");
  fprintf( fp,
  " $CONTRL SCFTYP=RHF UNITS=BOHR  $END \n");
  if ( comprox->isProxieFor->runType == NULL )
     fprintf( fp, " $CONTRL RUNTYP=ENERGY  $END \n");
  else
     fprintf( fp, " $CONTRL RUNTYP=%s $END \n", comprox->isProxieFor->runType );
  if ( (comprox->isProxieFor->runType != NULL) &&
          (strcmp(comprox->isProxieFor->runType, "OPTIMIZE" ) == 0 ))
           mplevel = 0;
  else if ( strcmp(comprox->isProxieFor->isTakenTo->name, "MP1" ) == 0 )
          mplevel = 1;
  else if ( strcmp(comprox->isProxieFor->isTakenTo->name, "MP2" )==0 )
          mplevel = 2;
  else
          mplevel = 0;
  fprintf( fp,
  " $CONTRL MPLEVL=%i  $END \n", mplevel );
  fprintf(fp, " $DATA \n");
  fprintf(fp, "%s \n", comprox->isProxieFor->name);
  strcpy( symm, comprox->isProxieFor->hasAsSubject->symmetry);
  if ( strcmp(symm, "C1" ) == 0 )
    fprintf(fp, "%s  \n", symm);
  else if ( strcmp( symm, "C2V") == 0 )

    fprintf( fp, "CNV  2 \n\n" );
  else
    fprintf(fp, "%s \n\n", symm);
```

```
   foreach( atom, comprox->isProxieFor->hasAsSubject->hasAtoms )
   {
    fprintf(fp, "%s   %lf  %lf  %lf  %lf \n",
                      atom->name, atom->charge, atom->x,
                      atom->y, atom->z );
    foreach( abs, comprox->BSI->AtomBSList )
    {
          if ( (strcmp(atom->name,
             PeriodicTable::PT.AtomicName(abs->atomicNumber)) == 0 ) ||
         (strcmp(atom->name,
        PeriodicTable::PT.AtomicSymbol(abs->atomicNumber)) == 0 ) )
             {
//    cout << "!"
//       << PeriodicTable::PT.AtomicName(abs->atomicNumber)
//      << PeriodicTable::PT.PTable[abs->atomicNumber]->atomicName
//          << "    " << abs->description << "\n";
     foreach( cont, abs->contractions )
     {
          if ( cont->L == S )
            strcpy(lval, "S");
      else if (cont->L == P)
            strcpy(lval,"P");
      else if (cont->L == D )
            strcpy(lval, "D");
      else if (cont->L == F )
            strcpy(lval,"F");
      else if (cont->L == G)
            strcpy(lval,"G");
//         cout << lval  << "   "  <<  cont->primitives.cardinality()
//        << "\n";
      fprintf(fp, "  %s    %i \n", lval, cont->primitives.cardinality() );
          i = 1;
      foreach ( prim, cont->primitives )
      {
//            cout << "  " << i
//               <<  "  " << prim->exponent << "        "
//               <<  prim->coefficient << " \n";
       fprintf( fp, "    %i   %lf   %lf \n", i++, prim->exponent,
                                                prim->coefficient );
      }
     }
    }
   }
   fprintf(fp, " \n" );
   }
   fprintf(fp, " $END \n");
   fprintf(fp, " $GUESS  GUESS=MINGUESS $END \n");
   fclose( fp );
}
```

The output parsing PERL script:

```
#!/usr/bin/perl

$pattern = $ARGV[0];
open(FILE, "$ARGV[1]");
@a = ();
$i = 0;
while (<FILE>) {
  if (/$pattern/) {
push(@a, $_);
  }
}
foreach $i  (0 .. $#a ) {
  print "$a[$i]";
}
```

APPENDIX C

CCDB Queries

NAME

expbym

SYNOPSIS

expbym [molecular name | molecular formula] {databasename}

DESCRIPTION

expbym displays all the experiments in the database, either lab or computational, that have been performed on the specified molecule. If a databasename is not specified, the query is performed on the database given by defaultDB (/ccdb/db in the case of the prototype). The molecular name or formula must be specified.

USAGE

expbym H2O
expbym H2O /ccdb/db
expbym water /ccdb/db

OUTPUT

Terse-display experiments which haveAsSubject: C2H4

1 COMP Ethylene DZP Test Case wi DZP (Dunning) MP2 Feller, Dave

8 COMP Optimize C2H4/STO-3G/MP0 STO-3G MP0 Feller, Dave

5 COMP Energy C2H4/DZP + Diffuse DZP + Diffuse (MP2 Feller, Dave

7 COMP Energy C2H4/STO-3G/MP2 STO-3G MP2 Feller, Dave

6 COMP Energy C2H4/DZP + Diffuse DZP + Diffuse (MP0 Feller, Dave

BUGS

Error not trapped if an invalid databasename is specified. Several extensions are possible:

- searching based on date and molecule - i.e. all experiments
- on water performed after 20th June 1991
- searching based on templates/families of molecules
- searching based on subgroups/atoms contained - i.e. all

experiments on molecules containing Zn

NAME

listCE

SYNOPSIS

listCE [chemist's last name | all] {databasename}

DESCRIPTION

listCE gives a terse display of all the computaional experiments performed by the specified chemist. If all is specified, all computational experiments in the database are displayed. If no databasename is specified, the query is performed on the database given by defaultDB (/ccdb/db for the prototype).

USAGE

listCE "Felle"
listCE "Feller" /ccdb/db
listCE all
listCE all /ccdb/db

OUTPUT

ID	EXPERIMENT NAME	MOL.	CODE	BASIS SET	LEVEL	DATE TIME
8	Optimize C2H4/STO-3G/MP0	C2H4	GAMESS	STO-3G	MP0	7/ 3/1992 17:49
7	Energy C2H4/STO-3G/MP2	C2H4	GAMESS	STO-3G	MP2	7/ 3/1992 17:46
6	Energy C2H4/DZP + Diffuse/MP0	C2H4	GAMESS	DZP + Diff	MP0	7/ 3/1992 17:22
5	Energy C2H4/DZP + Diffuse/MP2	C2H4	GAMESS	DZP + Diff	MP2	7/ 3/1992 14:25
1	Ethylene DZP Test Case with no	C2H4	MELDF	DZP (Dunni	MP2	12/12/90 14:30

BUGS

Error not trapped if an invalid databasename is specified.

NAME

CE

SYNOPSIS

CE [exp serial number] {databasename}

DESCRIPTION

CE gives a verbose display of the computational experiment specified by the serial number. If no database name is specified, the query is performed on the database given by defaultDB (/ccdb/db for the prototype).

USAGE

CE 8
CE 8 /ccdb/db

OUTPUT

```
Name:          Optimize C2H4/STO-3G/MP0
Molecule:      C2H4
                  Hydrogen  1.00797  1.00   2.32513356   1.72999310   0.00000000
                  Hydrogen  1.00797  1.00  -2.32513356   1.72999310   0.00000000
                  Hydrogen  1.00797  1.00   2.32513356  -1.72999310   0.00000000
                  Carbon   12.01115  6.00  -1.25666809   0.00000000   0.00000000
                  Carbon   12.01115  6.00   1.25666809   0.00000000   0.00000000
                  Hydrogen  1.00797  1.00  -2.32513356  -1.72999310   0.00000000
CodePackage:     GAMESS
BasisSet:        STO-3G
Level of Theory: MP0
Date Begun:      7/3/1992
Date Completed:  7/3/1992
CPU Time:        0.16 sec.
Elapsed Time:    0 sec.
Performed by:
              Feller Dave
ESCF:          -78.0505
ESDCI:         -78.3281
Properties:
```

BUGS

Error not trapped if an invalid databasename is specified. No check to make sure serial number specified is not > number of experiments in database.

NAME

listExp

SYNOPSIS

listExp {databasename}

DESCRIPTION

listExp gives a terse display of all (computational and lab) experiments in the database. If no database name is specified, the query is performed on the database given by defaultDB (/ccdb/db for the prototype).

USAGE

listExp
listExp /ccdb/db

OUTPUT

ethylene polarizability LAB
Geometry Optimization of H2O/STO-3G COMP Jan Labanowski

Energy - H2O/Dunning (DZP) COMP Jan Labanowski

Energy C2H4/DZP + Diffuse/MP0 COMP Dave Feller

Ethylene DZP Test Case with no compressi COMP Dave Feller

Optimize C2H4/STO-3G/MP0 COMP Dave Feller

Energy C2H4/STO-3G/MP2 COMP Dave Feller

Energy - H2O/STO-2G/MP0 COMP Jan Labanowski

Energy C2H4/DZP + Diffuse/MP2 COMP Dave Feller

BUGS

Error not trapped if an invalid databasename is specified.

NAME

listBS

SYNOPSIS

listBS [all | BasisSet Name] {databasename}

DESCRIPTION

listBS gives a terse display of all the basis sets in the specified database, or the default database, if no database name is given, and the keyword "all" is the first argument. If the first argument is a valid basis set name, listBS gives a verbose display of the specified basis set.

USAGE

listBS
listBS all
listBS STO-3G
listBS STO-3G /ccdb/dv

OUTPUT

listBS all /ccdb/db

BASIS SET NAME	AUTHORED BY
3-21++G	John Pople
DZP + Diffuse (Dunning)	Jim Dunning Patrick Hay
STO-2G	John Pople
6-31G**	John Pople
DZ (Dunning)	Jim Dunning
6-31G	John Pople
STO-3G	John Pople
6-31G*	John Pople
DZP (Dunning)	Jim Dunning Patrick Hay
3-21G	John Pople
3-21G*	John Pople

listBS STO-3G /ccdb/db

Basis Set name : STO-3G
Authors : John Pople

File Name(s) : STO-3G.BAS

Max L : 0 1 1 2
Supports sphericals : 0 0 0 0 2
max contractions/L : 3 3 3 3
0 3 3 3
0 0 0 3
0 0 0 0

Supports Atoms : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

BUGS

Error not trapped if an invalid databasename is specified. No message/warning if invalid database name specified.

NAME

listCP

SYNOPSIS

listCP {databasename}

DESCRIPTION

listCP gives a terse display of all the code packages in the specified database (or in the default database if no database name is specified).

USAGE

listCP
listCP /ccdb/db

OUTPUT

NAME: Gaussian
FORMAT:
COMPUTER:
COMPILER:

NAME: GAMESS
FORMAT: fmtBSIGAM
COMPUTER: Sun4
COMPILER:

BUGS

Error not trapped if an invalid databasename is specified.

NAME

runCE

SYNOPSIS

runCE [databasename]

DESCRIPTION

runCE displays the titles of all the computational chemistry experiments in the database. It then queries which on the user would like to run, and on which machine. It then builds the appropriate input datafiles and forks a process on the specified platform. It does not wait for the forked process to terminate.
runCE requires the following file to execute :
/ogi/students/mrao/scidb/schema/client.port

USAGE

runCE
runCE /ccdb/db

BUGS

runCE should really display only a (small) subset of the experiments in the database.

APPENDIX D

Setting up the Computational Environment for GAMESS

```
#include <sys/time.h>
#include <sys/resource.h>
#include <string.h>
#include <stream.h>
#include <assert.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <netinet/in.h>
#include <netdb.h>



main(int argc, char** argv)
{
  int pid;
  int forkprocess( int, int, char* );

  if ( argc < 4 )
  {
    printf("usage : fparent pipefd portno filename \n");
    exit(1);
  }
 // printf("%d : %d : %s \n", atoi(argv[1]), atoi(argv[2]), argv[3]);
  pid = forkprocess(atoi(argv[1]), atoi(argv[2]), argv[3]);

}




int forkprocess(int pwfd, int portx, char* filename)
{

   int pid;
   wait wstat;
   long x1;
   char* envp[16];
```

```
char  oname[40], buffer[48];
struct rusage pinfo;
void sendmonitor(int, int, char* );


    switch (pid = fork())
    {
        case -1:
          cout << "unable to create process" << endl;
          break;
        case 0:
             envp[0] = malloc(64);
          strcpy( envp[0], "IRCDATA=/tmp/gamess/");
             strcat( envp[0], filename );
             strcat( envp[0], ".irc");

          envp[1] = malloc(64);
      strcpy( envp[1], "INPUT=/tmp/gamess/");
             strcat( envp[1], filename );
             strcat( envp[1], ".F05");

             envp[2] = malloc(64);
             strcpy( envp[2], "PUNCH=/tmp/gamess/");
             strcat( envp[2], filename);
             strcat( envp[2], ".dat");

             envp[3] = malloc(64);
             strcpy( envp[3], "INTGRLS=/tmp/gamess/");
             strcat( envp[3], filename );
             strcat( envp[3], ".F08");

          envp[4] = malloc(64);
             strcpy( envp[4], "ORDINT=/tmp/gamess/");
    strcat( envp[4], filename );
          strcat( envp[4], ".F09");

             envp[5] = malloc(64);
             strcpy( envp[5], "JKFILE=/tmp/gamess/");
             strcat( envp[5], filename );
             strcat( envp[5], ".F09");

       envp[6] = malloc(64);
             strcpy( envp[6], "DICTNRY=/tmp/gamess/");
             strcat( envp[6], filename );
             strcat( envp[6], ".F10" );

          envp[7] = malloc(64);
             strcpy( envp[7], "DRTFILE=/tmp/gamess/");
          strcat( envp[7], filename );
             strcat( envp[7], ".F11");

             envp[8] = malloc(64);
             strcpy( envp[8], "CIVECTR=/tmp/gamess/");
             strcat( envp[8], filename );
```

```
            strcat( envp[8], ".F12");

            envp[9] = malloc(64);
          strcpy( envp[9], "NTNFMLA=/tmp/gamess/");
            strcat( envp[9], filename );
          strcat( envp[9], ".F13");

            envp[10] = malloc(64);
    strcpy( envp[10], "WORK15=/tmp/gamess/");
            strcat( envp[10], filename );
            strcat( envp[10], ".F15");

            envp[11] = malloc(64);
          strcpy( envp[11], "WORK16=/tmp/gamess/");
            strcat( envp[11], filename );
          strcat( envp[11], ".F16");

            envp[12] = malloc(64);
          strcpy( envp[12], "CSFSAFE=/tmp/gamess/");
            strcat( envp[12], filename );
          strcat( envp[12], ".F17");

            envp[13] = malloc(64);
          strcpy( envp[13], "FOCKDER=/tmp/gamess/");
            strcat( envp[13], filename );
          strcat( envp[13], ".F18" );

            envp[14] = malloc(64);
          strcpy( envp[14], "DASORT=/tmp/gamess/");
            strcat( envp[14], filename );
          strcat( envp[14], ".F20" );

          envp[15] = NULL;

            if ( close(0) == -1 )
          {
            printf(" couldn't close fd 0 \n");
            }
            fopen("/dev/null", "r" );
            if ( close(1) == -1 )
              printf("couldn't close fd 1\n");
            strcpy( oname, filename );
            strcat(oname, ".log" );
            if ( fopen(oname,"w" ) == NULL )
            {
              printf("cannot open %s.log for write\n", oname);
              break;
    }
        execle("/ogi/students/cushing/r/cp/gamess/gamess.exe",
         "gamess.exe",
                "basis", NULL, envp );
    printf("could not execute gamess\n");
            exit(1);
default:
```

```
            sprintf(buffer, "%d", pid);
           // printf("piping: %s %i \n", buffer, sizeof(buffer) );
            if ( write(pwfd, buffer, sizeof(buffer)) == -1 )
              perror("writing pipe");
            if ( wait(&wstat) == -1 )
              perror("wait");



            if ( getrusage(RUSAGE_CHILDREN, &pinfo) == 0 )
                    {
              x1 = pinfo.ru_utime.tv_sec*1000 + pinfo.ru_utime.tv_usec/1000 +
                 pinfo.ru_stime.tv_sec*1000 + pinfo.ru_stime.tv_usec/1000;
              printf("total cpu time (msecs) : %ld \n", x1 );
              sprintf(buffer, "%d*%d*%d", pid, x1, wstat.w_status );
              sendmonitor(portx, pid, buffer);

                    }
            else
              perror("getrusage");

            return(pid);

        }

 }

void sendmonitor( int portno, int pid, char* buffer )
{


  int sock;
  struct sockaddr_in server;
  struct hostent *hp;




  /* create socket */      n

  sock = socket( AF_INET, SOCK_STREAM, 0);
  if ( sock < 0 )
  {
     perror("opening stream socket");
     exit(1);
  }

  server.sin_family = AF_INET;
  hp = gethostbyname("smoked");
  if ( hp == 0 )
  {
         printf(" smoked : unknown host", );
         exit(2);
  }
```

```
    bcopy( (char *)hp->h_addr, (char *)&server.sin_addr, hp->h_length );
    server.sin_port = htons(portno);

    if ( connect( sock, (struct sockaddr *) &server,
            sizeof(server) ) < 0 )
    {
      close(sock);
      perror("connecting stream socket");
      exit(1);
    }

    if ( write(sock, buffer, strlen(buffer) ) < 0 )
      perror("1. writing stream socket");


    close(sock);

}
```

APPENDIX E

A General Introduction to Basis Sets

Ab initio quantum chemistry aims to determine the properties of molecules from first principles – namely the Schroedinger equation:

$$\mathbf{H}\psi = E\psi$$

where **H** represents the Hamiltonian operator, ψ is the wavefunction of the system (atom or molecule) under consideration, and E is the energy of the system. An analytic solution to the Schroedinger equation has been achieved only for the Hydrogen molecule. For all other molecules, numerical techniques are used to solve the equation, and these require an initial guess for ψ.

Historically, the quantum calculations for molecules were performed using LCAO MO: Linear Combination of Atomic Orbitals – Molecular Orbitals. Using LCAO-MO the molecular orbitals are represented as some linear combination of atomic orbitals:

$$\psi_i = \Sigma\,(u=1,\ u=n)c_{u,i}\,\phi_u$$

where ψ_i represents the i^{th} molecular orbital, ϕ_u is the u^{th} atomic orbital, n is the number of atomic orbitals and the $c_{u,i}$ are the coefficients of linear combination. Technically, the Atomic Orbitals (AO) themselves are the solutions of the Hartree-Fock equations for the atom, i.e., each AO is a wave function representing a single electron in the atom. There was a lot of ambiguity in the usage of the term AO. To avoid this ambiguity, the term AO now has been reserved for the technical sense above, and the term *basis function*[7] or *contraction* used for functions that are not derived from the atomic Hartree-Fock equations, but are used as the basis of linear combination to generate Molecular Orbitals (MOs). Initially, Slater Type Orbitals (STOs) were used as basis functions because of their similarity to the atomic hydrogen orbitals. However, the STOs are not suitable for fast numerical calculation, and Gaussian Type Orbitals (GTOs) were introduced. The GTO can be represented as:

$$g(\alpha, l,m,n; x,y,z) = N\exp(-\alpha r^2)x^l y^m z^n$$

where $r^2 = x^2 + y^2 + z^2$, N is a normalization constant, and l,m,n are integers non-negative integers, such that l+m+n = L, the angular momentum quantum number (for example, L=0 for s-type, L=1 for p-type orbitals). The GTOs are not truly orbitals, i.e. they do not represent an electronic wave function, and they are more correctly referred to as gaussian primitives.

[7] A Basis Set refers to a collection of basis functions, derived using the same assumptions, and generally optimized for either a type of type of numerical calculatation or a group of atoms. Developing new basis sets is an active area of research in the field of ab initio Computational Chemistry.

For molecular calculations, the gaussian primitives are often *contracted.* A contraction means "a linear combination of gaussian primitives to be used as a basis function". For example, 4 s-type[8] gaussian primitives may be used to represent the 1s atomic hydrogen orbital:

$$\psi(1s) = 0.50907N_1\exp[-0.123317r^2] + 0.47449N_2\exp[-0.453757r^2] + 0.13424N_3\exp[-2.01330r^2] + 0.01906N_4\exp[-13.3615r^2]$$

The $N_{1\text{-}4}$ are called the normalization constants.
The same 4 primitives may be grouped into 2 contractions:

$$\phi(1s,1) = 0.50907N_1\exp[-0.123317r^2]$$

$$\phi(1s,2) = N\{ 0.47449N_2\exp[-0.453757r^2] + 0.13424N_3\exp[-2.01330r^2] + 0.01906N_4\exp[13.3615r^2] \}$$

The contractions $\phi(1s,1)$ and $\phi(1s,2)$ can be used as the basis functions in subsequent molecular calculations. Using basis functions (combinations of primitives) instead of primitives reduces the computational effort in doing molecular calculations, but can introduce possible inaccuracies.
The STO-3G basis set for water is:

Hydrogen

S	2	
	α	coefficient
1	1.309756	0.430128
2	0.233136	0.678914

Oxygen

S	2	
	α	coefficient
1	49.980971	0.430128
2	8.896588	0.678914
S	2	
1	1.945237	0.049472
2	0.493363	0.963782
P	2	
1	1.945237	0.511541
2	0.493363	0.612820

Reading this, we can see that the gaussian primitives:

Hydrogen(1):

$$g(s,1) = 0.430128\exp[-1.309756r^2]$$

$$g(s,2) = 0.678914\exp[-0.233136\, r^2]$$

Hydrogen(2):

---

[8] For an s-type orbital, the angular momentum quantum number is 0, ie. l+m+n = 0.
Thus, for s-type orbitals l=m=n=0.
For a p-type orbital, the angular momentun quantum number is 1, ie. l+m+n = 1.
There are 3 ways the angular momentum qunatum number to be 1:
l = 1 (angular momnetum along the x-axis); m=n=0;
m= 1 (angular momentum along the y-axis); l=n=0;
n=1 (angular momentum along the z-axis); l=m=0;

$g(s,1) = 0.430128\exp[-1.309756\ r^2]$
$g(s,2) = 0.678914\exp[-0.233136\ r^2]$

Oxygen:
$g(s,1) = 0.430128\exp[-49.980971\ r^2]$
$g(s,2) = 0.678914\exp[-8.896588\ r^2]$
$g(s,3) = 0.049472\exp[-1.945237\ r^2]$
$g(s,4) = 0.963782\exp[-0.493363\ r^2]$
$g(p,1) = 0.511541*x*\exp[-1.945237\ r^2]$
$g(p,2) = 0.511541*y*\exp[-1.945237\ r^2]$
$g(p,3) = 0.511541*z*\exp[-1.945237\ r^2]$
$g(p,4) = 0.612820\ *x*\exp[-0.493363\ r^2]$
$g(p,5) = 0.612820\ *y*\exp[-0.493363\ r^2]$
$g(p,6) = 0.612820\ *z*\exp[-0.493363\ r^2]$

The *contractions* or *basis functions* are:

Hydrogen(1):
$\phi(1) = N_1\{\ g(s,1) + g(s,2)\ \}$
$= N_1\{0.430128*\exp[-1.309756\ r^2] + 0.678914*\exp[-0.233136\ r^2]\ \}$

Hydrogen(2):
$\phi(2) = N_1\{\ g(s,1) + g(s,2)\ \}$
$= N_1\{0.430128*\exp[-1.309756\ r^2] + 0.678914*\exp[-0.233136\ r^2]\ \}$

Oxygen:
$\phi(3) = N_2\{\ g(s,1) + g(s,2)\ \}$
$= N_2\{0.430128*\exp[-49.980971\ r^2] + 0.678914*\exp[-8.896588\ r^2]\ \}$
$\phi(4) = N_3\{\ g(s,3) + g(s,4)\ \}$
$= N_3\{0.049472*\exp[-1.945237\ r^2] + 0.963782*\exp[-0.493363\ r^2]\ \}$
$\phi(5,6,7) = N_4\{\ g(p,1) + g(p,2)\ \}$
$\phi(5) = N_4\ x\ *\ \{0.511541*\exp[-1.945237\ r^2] + 0.612820\ *\exp[-0.493363\ r^2]\ \}$ (l =1; m=n=0)
$\phi(6) = N_4\ y\ *\ \{0.511541*\exp[-1.945237\ r^2] + 0.612820\ *\exp[-0.493363\ r^2]\ \}$ (m=1; l=n=0)
$\phi(7) = N_4\ z\ *\ \{0.511541*\exp[-1.945237\ r^2] + 0.612820\ *\exp[-0.493363\ r^2]\ \}$ (n=1; l=m=0)

The $N_{1\text{-}4}$ represent normalization constants.
The wave function for the i^{th} molecular orbital in the water molecule, can now be represented by:

$$\psi_1 = c_{11}\ \phi(1) + c_{21}\phi(2) + c_{31}\phi(3) + c_{41}\phi(4) + c_{51}\phi(5) + c_{61}\phi(6) + c_{71}\phi(7)$$
$$\psi_2 = c_{12}\ \phi(1) + c_{22}\phi(2) + c_{32}\phi(3) + c_{42}\phi(4) + c_{52}\phi(5) + c_{62}\phi(6) + c_{72}\phi(7)$$
$$\psi_3 = c_{13}\ \phi(1) + c_{23}\phi(2) + c_{33}\phi(3) + c_{43}\phi(4) + c_{53}\phi(5) + c_{63}\phi(6) + c_{73}\phi(7)$$
$$\psi_4 = c_{14}\ \phi(1) + c_{24}\phi(2) + c_{34}\phi(3) + c_{44}\phi(4) + c_{54}\phi(5) + c_{64}\phi(6) + c_{74}\phi(7)$$
$$\psi_5 = c_{15}\ \phi(1) + c_{25}\phi(2) + c_{35}\phi(3) + c_{45}\phi(4) + c_{55}\phi(5) + c_{65}\phi(6) + c_{75}\phi(7)$$
$$\psi_6 = c_{16}\ \phi(1) + c_{26}\phi(2) + c_{36}\phi(3) + c_{46}\phi(4) + c_{576}\phi(5) + c_{66}\phi(6) + c_{76}\phi(7)$$
$$\psi_7 = c_{17}\ \phi(1) + c_{27}\phi(2) + c_{37}\phi(3) + c_{47}\phi(4) + c_{57}\phi(5) + c_{67}\phi(6) + c_{77}\phi(7)$$

where the c_{ij}s are the co-efficients of linear combination that are determined by the ab initio computational chemistry application. The molecular wave function for the water molecule is: