Computational Proxies: An Object-based Infrastructure for Computational Science

Judith Bayard Cushing
B.A., The College of William and Mary, 1968
M.A., Brown University, 1970

A dissertation submitted to the faculty of the
Oregon Graduate Institute of Science & Technology
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

April 1995

The dissertation "Computational Proxies: An Object-based Infrastructure for Computational Science" by Judith Bayard Cushing has been examined and approved by the following Examination Committee:

David Maier
Professor
Thesis Research Adviser

David Feller
Research Chemist
Battelle Pacific Northwest Laboratories

Steven Otto
Assistant Professor

Jonathan Walpole
Associate Professor

Dedication

For John, who for this thesis relinquished as many days skiing and hiking trips, and as many nights together, as did I; as for home-cooked meals, he forwent many more.

For my mother, whose company on many Sunday afternoons I missed.

And, for my colleagues and students at The Evergreen State College in Olympia, Washington, to whose teaching of me I am delighted to return.

Acknowledgements

This research reported in this thesis was very much an interdisciplinary effort. As such, it involved perhaps a significantly higher level of collaboration than most other dissertation research, and the first person plural pronoun I used in writing thus holds special significance. I could not have accomplished the work without my principle collaborators: David Maier, David Feller, Meenakshi Rao and Don Abel. Although many of the ideas in this thesis belong to them, the mistakes belong to me alone.

I am most grateful to Professor David Maier of the Oregon Graduate Institute for his unflagging support and guidance, and for his many and significant contributions not only to this thesis but to my research and teaching. Dave has served not only as an advisor but as a mentor and model teacher and researcher. As a result of his tutelage, I am more focused and disciplined than when I came to OGI. Suffering my enthusiasm for new ideas, Dave showed me how to recognize those with promise while weeding out the superfluous. I cannot imagine a better advisor; he can hone a researcher and teacher. After it was over, he even taught me the PH.D. handshake. While I can never repay his wise counsel and many kindnesses, I'll do my best to assure that the care and consideration he gives his students pays off in the next-generation I help spawn.

To my collaborators at the Molecular Science Research Center at Battelle Pacific Northwest Laboratories I owe a great deal as well. They generously funded three years of my research, and provided the "raw data" upon which that research was based. Dr. David Feller engendered in me not only an understanding and passion for ab initio computational applications, but also his vision of how scientists would benefit from these powerful tools were they easier to use. For his many hours of patient and enlightened explanation about his area and our work, and for his willingness to take seriously this computer scientist's view, I thank him. I hope that this work contributes to his vision.

D. Michael DeVaney, also of PNL, contributed much not only by ably guiding research contracts between OGI and PNL, but by arranging for me to meet with PNL scientists, suggesting valuable information sources, and always making me feel "at home" in Richland. Many other PNL scientists contributed significantly as well: computer scientists Karen Schuchardt and Thomas Keller, in particular, worked long hours with me to refine specifications for the Computational Chemistry Database and responsibilities for the Computational Proxy. I enjoyed our collaboration and note that the thesis is far better for their contribution. To the Chemistry Objects Research Architecture (CORA) Working Group, I owe a broader understanding of the chemists' work than I would have had working only with ab initio chemists: Dr. Mark Thompson and Dr. Ray Bair (of PNL), George Purvis, III (of CAChe Scientific), and Dr. Douglas McLean and Dr. Eric Replogle (of IBM Almaden). Finally, I acknowledge my debt to Jim Thomas, who made the initial match and, not unlike a guardian angel, watched over the whole process.

Meenakshi Rao and Don Abel merit considerable recognition as collaborators: they chose to conduct their master's thesis research on the computational proxy. Meenakshi, whose former studies in quantum physics lent additional scientific expertise to the design of the computational chemistry database, very ably designed and programmed the proxy's Compute Monitor and compute daemon programs. Don designed and programmed the proxy's Parser-Converter-Loader program. Without their feedback in defining the proxy's specification and their considerable technical expertise, this work would not have been possible. Without their good humor, and fellowship, it would have been considerably less enjoyable.

I applaud the Department of Computer Science and Enginnering and the Oregon Graduate Institute of Science & Engineering for fostering high quality research and learning environments, each of which engenders cooperation among students and is highly supportive of women students. In particular, I thank my own dissertation committee. Prof. Steve Otto's expertise as both an active researcher in portable software environments for parallel processors and a (former) physicist provided an excellent sounding board for the proxy's potential value to the computational scientist. Prof. Jon Walpole, as an expert in distributed operating systems and high speed networks, worked with me to develop a clearer picture of the proxy's network service component. Steve and Jon's excellent suggestions greatly improved the final product, particularly with respect to a vision of future distributed system services. Their enthusiasm for and encouragement of my work is most appreciated.

Other OGI faculty and staff deserve more than the minor mention I can make here: Prof. Lois Delcambre's careful reading my initial thesis outline, and excellent observations about database design helped refine both thesis and database. Profs. Dick Kieburtz, James Hook, David Novick, Todd Leen, Françoise Belgarde and Ron Cole each offered encouragement and example through excellent teaching and research. Prof. Michael Wolfe, who in addition piqued my intellectual interest in distributed and parallel scientific applications, holds a special place in this regard: his many acts of energetic kindness and good teaching are much appreciated. The day to day work of an outstanding staff provides the environment that renders possible excellent research at OGI. I thank: Phil Barrett, who helped with numerous proposals and reports; an outstanding systems staff, who keeps computers and software running (Nike Horton, Mark Morissey, Nora Auseklis, Marion Hakanson, Bruce Jerrick); and Jo Ann Binkerd, who makes the Data Intensive Systems Center run. I also thank the OGI librarians, especially Julianne Williams, for unflagging support of student research and for making the library warm and welcoming.

I acknowledge gratefully the significant role played by my fellow students' intellectual curiosity and collegiality. In particular Bennet Vance, Laura McKinney, Harini Srinivasan, Ravi Konuru, Moira Mallison, Jonathon Inouye, Scott Daniels, Jenny Orr, Leo Fegaras, and Rich Staehli (who closed my own personal loop between teaching and research) proffered a collegiality that will last longer than our professional lives.

I was welcomed to the world of research by numerous other researchers — too numerous to mention by name — who responded generously to questions and requests. I acknowledge the scientific database research community for its openness and dedication to helping computer science serve science and putting scientists' needs to work pushing computer science research. Dr. Maria Zemankova of the National Science Foundation deserves special recognition for her tireless work on behalf of those conducting scientific database research. For particular intellectual generosity, I thank: Lougie Anderson, Jean Bell, Jim Diederich, Denise Ecklund, Earl Ecklund, Brian Ehlmann, Thom Dunning, Ken Fasman, Tom Ferrin, James French, Hans Hinterberger, Malcolm Gleser, Keri Anderson Healy, Yannis Ioannidis, Dick Kouzes, Roger Lea, Meg Lewis, Tom Marr, David J. Mishelevich, Hitomi Ohkawa, Susan Owicki, Rob Pecherer, Dick Peskin, Larry Rothenberg, Duri Schmidt, Jean Scholtz, Linda Shapiro, Arie Shoshanni, Barbara Simmons, Tedd Sparr, George Stemple, George Struble, Sandy Walther, André Weinand, Dave Woods. I shall endeavor to repay these debts by ministering to new researchers.

Finally, I thank my colleagues at The Evergreen State College who erased disciplinary boundaries and taught me the joys of collaboration; who not only granted me years of leave to conduct this research, but who also offered unflagging encouragement and covered for me while I was at OGI, either on leave or writing. John Cushing, George Dimitroff, Al Leisenring, John Marvin, Neal Nelson, David Paulsen, Sherri Shulman, and Masao Sugiyama, my colleagues who teach computing when they don't teach something else, bore the brunt of this burden. The women faculty who continue an active researcher agenda in the face of heavy teaching responsibility taken very seriously served as models for me: Jovana Brown, Sally Cloninger, Carolyn Dobbs, Susan Fiksdal, Susan Finkel, Marilyn Frasca, Virginia Hill, Linda Kahan, Elizabeth Kutter, Jean Mandeberg, Nalini Nadkarni, and Janet Ott, to name a few. Librarians Lee Lyttle and Ernestine Kimbro generously took time from their busy schedules to help confirm some references; Member of the Faculty Oscar Soule contributed similarly. Special thanks go to Barbara Smith who, as Senior Academic Dean in 1988, helped me procure from Boeing particularly generous funding for my first year, and to Byron Youtz who, with Barbara, argued persuasively to College officials to "let me take it and go". Finally, I thank Thad Curtz, who, alone among those who didn't have to, read the whole thing in its embryonic state and took the time to understand and offer many helpful comments.

This work was supported financially by the U.S. Department of Energy's Pacific Northwest Laboratory, U.S. National Science Foundation Grants IRI-911-70-08 and IRI-922-47-26, and the Oregon Advanced Computing Institute (OACIS). Software grants from Object Design, Inc., and Servio Logic Corporation supplied the software for the ObjectStore and GemStone implementations, respectively. The Boeing Company, Seattle Washington, through a generous grant to The Evergreen State College, provided the fellowship that started it all.

Pacific Northwest Laboratory is operated for the U.S. Department of Energy by Battelle Memorial Institute under contract DE-AC06-76RLO. Chem3D is a registered trademark of Cambridge Scientific Computing, Inc.; CAChe of CAChe Scientific; Gaussian of Gaussian, Inc.; GemStone and Geode of Servio Corporation; ObjectStore of Object Design, Inc.; and SparcStation of Sun Microsystems.

Contents

Dedication iii

Acknowledgements iv

Abstract xv

1 Introduction 1
- 1.1 Building an Experiment Management Infrastructure 3
  - 1.1.1 Developing a Domain-Level Conceptual Model 5
  - 1.1.2 Developing a Model for Computational Services 6
- 1.2 Research Design 8
- 1.3 Major Contributions of the Research 10
  - 1.3.1 Additional Potential of Proxies 11
- 1.4 Organization of the Thesis 12

2 Background and Related Work 13
- 2.1 Scientific Computing 14
  - 2.1.1 Computational Chemistry Applications 15
  - 2.1.2 Tools For Computational Scientists 15
  - 2.1.3 Scientific Data Management 17
- 2.2 Application Integration Efforts 22
  - 2.2.1 Integration via the User Interface 22
  - 2.2.2 Software Systems 23
- 2.3 Database Research 26
  - 2.3.1 Data Conversion Research 26
  - 2.3.2 Object-Oriented Databases 26

3 Providing Data Services: The CCDB 29
- 3.1 Ab Initio Computational Chemistry 34
  - 3.1.1 User Scenario for Ab Initio Applications 44
- 3.2 A Conceptual Model for Ab Initio Computational Chemistry 53

3.2.1 Chemist and Experiment 55
3.2.2 Aggregations of Experiment: Investigations, Suites and Personal Sets 56
3.2.3 Molecule 56
3.2.4 Computational Chemistry Application 58
3.2.5 Basis Set 60
3.2.6 Basis Set Library 60
3.2.7 Level of Theory 61
3.2.8 Observable Property 61
3.3 The Computational Chemistry Information Model 62
3.3.1 An Entity-Relationship Model for the Database 63
3.3.2 Information Modeling Issues 67
3.4 Logical and Physical Design 70
3.4.1 Choice of Target Database Management System 71
3.4.2 The Logical Design — an Object-Oriented Database Schema 74
3.4.3 Computational Chemistry Database — Physical Design 77
3.4.4 Loading the Database 78
3.4.5 Summary: Data Services Alone are Inadequate 80

4 Providing Computation Services: Proxies 82
4.1 Design Issues and Alternatives 83
4.1.1 Strategy for Connecting the Applications to the Database 84
4.1.2 Challenges in Describing Applications Declaratively 85
4.2 Computational Proxy: Definition and Functional Specification 88
4.2.1 Computational Proxy: Conceptual Model 92
4.2.2 Database Support for the Proxy: Registering Applications 95
4.2.3 Computational Proxy: Behavior 96
4.3 Functional Components of the Computational Proxy 97
4.3.1 Input and Output Templates 98
4.3.2 Application Registration 100
4.3.3 Data Input: Generating Input Files from the Database 104
4.3.4 Data Capture: Loading Experimental Results into the Database . . 115
4.3.5 Mission Control 134
4.4 Computational Proxy Architecture 136
4.4.1 System-level Requirements — Network Services 138
4.4.2 User-Level Requirements — The User Interface 146
4.5 Chapter Conclusion 147

5 Implementing the Database and Proxy 149
5.1 Implementation Constraints and Choices . . . 150
5.2 Implementation of Database Services . . . 153
5.2.1 Populating the Database . . . 155
5.2.2 Implementation of Queries and Database Operations . . . 156
5.2.3 Summary of Database Implementation Experience . . . 157
5.3 Implementation of the Computational Proxy . . . 159
5.3.1 Generating Input Files and Launching Experiments . . . 160
5.3.2 Output File Parsing . . . 161
5.3.3 Summary and Conclusions of Proxy Implementation Experience . . . 175
5.4 Application Registration . . . 176
5.5 Implementation of Network Services: The Compute Monitor . . . 178
5.5.1 Design Objectives . . . 179
5.5.2 Implementation Alternatives . . . 180
5.5.3 Physical Design and Implementation . . . 183
5.5.4 Summary of the Network Services Implementation Experience . . . 186
5.6 Summary of the Implementation Experience . . . 187

6 Validating the Model and the Proxy 190
6.1 The Infrastructure as a Whole: A User Scenario . . . 191
6.2 Data Services: The Computational Chemistry Model . . . 197
6.3 Computational Services: the Proxy . . . 201
6.3.1 Adding New Applications to the Infrastructure. . . . 202
6.3.2 Profile of Applications Readily Supported . . . 206
6.3.3 Portability of the Proxy System . . . 208
6.3.4 Concluding Remarks on Evaluating the Computational Services . . . 208
6.4 Summary of the Validation . . . 209

7 Contributions, Lessons-Learned and Future Work 210
7.1 The Need for an Infrastructure for Computational Experimentation . . . 211
7.2 Computational Proxies: An Infrastructure for Experiment Management . . . 213
7.3 Research Contributions . . . 215
7.4 Lessons Learned . . . 219
7.5 Follow-on Work . . . 221
7.5.1 Engineering Changes to the Applications and Infrastructure . . . 222
7.5.2 Engineering Changes to Object-Oriented Tools . . . 223
7.5.3 Follow-on Research Opportunities . . . 224
7.6 Future Work . . . 226

7.7 Final Remarks 227

Bibliography **229**

List of Figures

1.1 A data-centered infrastructure for experiment management. 5

3.1 Inputs and outputs for computational chemistry codes. 37
3.2 Transition of hydrogen and oxygen to water. 38
3.3 Molecular structure representations. 40
3.4 Syntactical complexity of computational chemistry applications. 49
3.5 Application taxonomy. 60
3.6 Computational chemistry information model. 64
3.7 Computational chemistry instance diagram. 65
3.8 Basis set. 66
3.9 Molecular orbital. 67
3.10 Functional relationships between experiment and property. 68
3.11 Persistent names for the CCDB logical model. 76

4.1 A data-centered framework for computational science. 89
4.2 Computational proxy functions for managing ongoing experiments. 90
4.3 Hiding syntactic complexity of computational codes. 91
4.4 Information model for computational proxy. 93
4.5 Conceptual model for template. 99
4.6 Application registration functions. 101
4.7 Application input functions needed by the computational proxy. 105
4.8 Sample input file for a GAMESS experiment on hydrogen. 107
4.9 Input template for optimize experiment type, GAMESS application. 114
4.10 Application output capture functions needed by the computational proxy. . 117
4.11 Reformatting GAMESS molecular orbital text for easier parsing. 122
4.12 The computational proxy's output template. 125
4.13 Instances of OutputTemplate for energy and iterations. 131
4.14 Output template for parsing a molecular orbital. 132
4.15 Mission control functions needed by the computational proxy. 135
4.16 Compute Server architecture. 137
4.17 A typical scenario for use of the proxy mechanism. 139

4.18 Event trace illustrating the proxy's use of network services. 145

5.1 Implemented Computational Chemistry Database 154
5.2 Ternary relationship among Template, Experiment Type and Application. . 173
5.3 Implemented ternary relationship. 174
5.4 Using DCE to provide Network Services. 182
5.5 Implementation of the Computation Services Architecture. 185

6.1 Support for managing computational experiments — Currently, with Database and with Proxy plus Database. 196
6.2 The CORA Working Group Chemistry Model 199

Abstract

Computational Proxies:
An Object-based Infrastructure
for
Computational Science

Judith Bayard Cushing, Ph.D.
Oregon Graduate Institute of Science & Technology, 1995

Supervising Professor: David Maier

Scientific computing's rich legacy of data and programs contains its own major disadvantage: lack of interoperability at the user level. Even within a single subfield, scientists are faced with a plethora of potentially useful programs that run on a number of different computers. When these programs do not "interoperate", such simple but desirable tasks as using the output of one program as the input of another become major obstacles to "doing science". Accessing data and running programs in heterogeneous computing environments further compounds problems of interoperability because of considerable differences in data representation, file transfer and program control protocols among computer platforms. In addition, during a single investigation, a typical scientist might name and keep track of hundreds of files on several different computers.

This dissertation addresses problems of data management and of program and data interoperability among computational science applications. We postulate that a database of experiment data would alleviate problems of interoperability and file management, but that connecting a database to existing applications is neither straightforward nor adequate for computational applications. We propose a middleware solution built within a database tradition, and describe the functionality needed for computational experiment management. We believe that a data-centered solution to interoperabilty problems - one that makes current versions of data available to cooperating user applications and system services - shows particular promise. Our solution consists of a domain-specific information model and an object-oriented persistent structure that supports computation as well as data management. This abstract data structure, dubbed "computational proxy", models within an object database scientific programs and processes.

Proxies maintain persistent local records of on-going computational experiments, and provide a consistent view of different applications executing on multiple processors. They provide for launching and monitoring experiments; generating data input to the experiment from the database; and capturing experimental results. The computational proxy mechanism also provides ways to declaratively define the database interface to computational applications. The infrastructure we propose can be used in a migration path from stand-alone legacy applications to distributed database services and adapted for newer object-based applications. A prototype of the computational proxy infrastructure has been implemented in C++ and ObjectStore on a Sun Sparcstation for applications in ab initio computational chemistry.

Chapter 1

Introduction

Physical scientists have been intensive users of modern digital computers since the late 1940's and early 1950's, and the world of scientific computation carries forward a rich legacy of data and programs [69]. Indeed, the many and varied scientific applications constitute a major economic investment, and new scientific applications often stretch the state of the art in computing technology. While the pull of scientific applications greatly affects computing technology, computers also transform the ways in which many scientists work [96] and scientists have come to depend heavily upon them. Indeed, computing plays a central role in the daily activity of scientists today, and many use a range of computers and platforms — from personal workstations, through mid-sized computers attached to a local network, to massively parallel machines available over a wide-area network.

Unfortunately, this legacy of the many scientific applications developed over the past forty years entails a major disadvantage: lack of interoperability at the user level. Even within a single sub-discipline, scientists confront a plethora of potentially useful but often incompatible programs. While different programs may be scientifically applicable to a single investigation, outputs from two different programs are rarely comparable and output from one program cannot easily be used as input to another — especially if those programs run on different platforms. Levels of incompatibility range from differences in physical formats (at the lowest level), to divergence in the logical structure and grouping of information, to disagreement at the conceptual level (sometimes dangerously implicit) about the actual meaning of terms. In addition to dealing with data and program interoperability, a scientist may need to name and keep track of hundreds of files on several computers during one investigation. Because most scientists use more than one computer, problems of program incompatibility and file management are compounded by the lack of interoperability of computer platforms and operating systems.

Problems of data management and of program and data interoperability can significantly decrease the amount of time spent on science per se. Some scientists are so discouraged by complex computing environments that they do not use computers at all, or only for word processing [128]. Other scientists have become highly trained users or expert programmers and deal effectively with this range of problems, though at some considerable cost. Computational scientists, who investigate scientific problems by performing computer simulations of physical systems, often use five or six programs and two or three computers during a typical day. Their programs are computationally intensive and one invocation may run for several hours, or even days or weeks. Improvements in computing power and software over the past ten years have greatly enhanced the computational scientist's "laboratory" but have exacerbated data management and interoperability problems. As further increases in computing power make computational science methods useful to a wider range of scientists, however, we expect an increase in both casual and expert use of computational applications. These new users will be less willing or able to cope with the current complexity of the computing environment. In addition to help with data management and interoperability, they will undoubtedly need better software to set up, run, and interpret computational experiments. Such help, we believe, will also be welcomed by expert users, who would prefer to focus on scientific, rather than data management, problems.

This dissertation addresses problems of data management and interoperability for computational science applications. Of course, computing problems other than data management face the computational scientist, such as scheduling multiple related experiments and scheduling and migrating experiments across a network. We chose to address problems of data and file management first because we believe that solving the scientists' data management problems will provide both the infrastructure and experience for approaching other problems.

Initial efforts for this research focused on building a database repository of past experiments for the domain of computational chemistry. Aimed at helping non-experts run computational experiments, the database repository work addressed data management issues alone. Our collaborators from Pacific Northwest Laboratories had observed that users have considerable difficulty setting up and interpreting computational experiments and believed that a database of past experiments could help.

We considered the database and programming language alternatives available to us, and chose object-oriented technology as our implementation vehicle. We then built an object-oriented database — called the "Computational Chemistry Database" or CCDB — to maintain experiment data (inputs, parameters, and outputs) from different applications in comparable formats. Initial results indicated that the CCDB alleviated some problems facing computational scientists. However, we became convinced that lack of program interoperability precluded the effective maintenance of our repository. Loading experiment data remained a problem because there was no direct connection between the application programs and the database. We also realized that computational scientists needed help not only setting up but also running their experiments, given the wide range of computers on which their applications are installed. In short, we found that providing data services in the absence of ways to load experiment data and to manage long-lived computations was an inadequate solution to the data management problems facing computational scientists. We then proposed and prototyped an infrastructure for managing computational experiments.

1.1 Building an Experiment Management Infrastructure

The major barrier to building an infrastructure for computational experiment management is the lack of interoperability among programs. Analogous problems of interoperability and shared file management in the realm of business data processing are being solved through common data models and distributed databases. Unfortunately, current record-oriented database technology does not support scientific applications well. While object-oriented systems will likely provide the flexibility for modeling complex scientific data structures that the relational model lacks [60, 91, 164], neither relational nor object-oriented database systems provide the support for managing long-lived processes[1] needed

[1]By *process* we mean a program in execution, as per the standard operating system definition[176]. For our purposes, a process is always an activated computational application program, i.e., a computational

for doing computational science. This dissertation describes the additional functionality needed for computational experiment management and proposes an object-oriented infrastructure to meet those needs. An effective solution to managing experiment processes should provide a consistent and persistent view of both experiment data and ongoing experiments. Hence, our infrastructure combines database services and computation services, and integrates loading experiment data with experiment management.

Data services (a repository of past experiments) are provided through a domain-specific information model implemented in an object-oriented database system. Computation services (on-line connections between application programs and the database) are provided by "computational proxies". Computational proxies model executions of applications as database objects and directly store application inputs and outputs in the domain database. We call our infrastructure "data-centered" because the object database, containing both domain data and the proxy representation of ongoing experiments, is the mediator of experimental activity between the user interface and computational applications. (See Figure 1.1.) The proxy uses network services to make input data available to applications and experimental results available to users. We faced two major issues in developing this infrastructure:

1. Defining and implementing a conceptual model general enough to cover the inputs and outputs of computational applications within a particular domain, yet intuitive and acceptable to end users.

2. Defining and implementing a data structure powerful enough to model remote invocations of the computational applications in the database.

These two issues are discussed below in Sections 1.1.1 and 1.1.2. We contend that the domain model and the computational proxy together provide an infrastructure for managing computational experiments effectively.

experiment in execution phase.

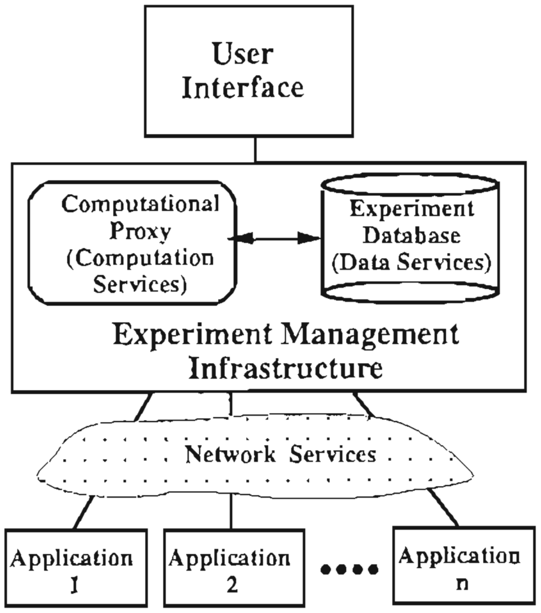

User Interface
Computational Proxy (Computation Services)
Experiment Database (Data Services)
Experiment Management Infrastructure
Network Services
Application 1
Application 2
Application n

Figure 1.1: A data-centered infrastructure for experiment management.

1.1.1 Developing a Domain-Level Conceptual Model

Developing a conceptual model general enough to cover the major inputs and outputs of the application programs of interest is critical for three reasons. First, scientists using a number of applications must themselves have a unified conceptual view of the domain before they can effectively navigate among applications. Secondly, implicit conceptual views of individual researchers may be in conflict either with each other or with those of an application. Without agreement at the conceptual level, mapping inputs from one program to another is difficult at best, error prone or impossible at worst. Thirdly, a common model into which application inputs and outputs can be translated is necessary before experimental results can be rendered comparable.

While a common conceptual view is often implicitly held across researchers and applications, it is rarely written down — except as data structures internal to the application programs. The implicit common model must become explicit before any mapping of output from one application to the input of another can be automated. The domain-level conceptual model explicitly documents the common view that makes such mapping possible.

Developing a common conceptual model is difficult because users must agree on and write down such basic definitions as *experiment* and *subject of experiment*. What seem trivial tasks attainable by simply referring to a freshman text are soon hotly debated. Is an experiment one single run? Is it a series of runs that produces an array of values, say, a potential energy surface? Is the subject of a computational chemistry experiment a molecule, or a particular geometrical structure, or a chemical mixture undergoing a reaction?

Once the conceptual model has been developed it must be rendered into a logical database design. This requires casting the conceptual model first into an information model and then into an existing (logical) data model. This existing data model must also be implemented as an operational and reliable database management system product. Here also lay challenges resulting from the inherent complexity of the data: How should ternary relationships or attributed binary relationships be represented if the chosen data model does not support them? What intermediate abstract data structures would simplify the design and facilitate understanding? Such data types might be general to several computational domains, or specific to one, and might involve further modeling at the conceptual or information level.

1.1.2 Developing a Model for Computational Services

Our model for computational services consisted of an abstract data type, dubbed "computational proxy". The computational proxy is the locus of experiment control in our experiment management infrastructure. A computational proxy stands-in, within the database, for an active process (usually remote) that is running a scientific application (usually computationally intensive and long-lived). The proxy provides the user with consistent and persistent views of different applications executing on distributed processors. The proxy can be used for specifying computational experiments, generating input to experiments from the database, launching and monitoring experiments, and loading experimental results into the database. In effect, proxies and related objects model scientific programs and processes.

Our primary goal is to provide computational services while hiding syntactic differences between applications and environments from the end user. Proxies also reduce the number of explicit user actions — converting and transferring files, logging on to remote computers, and so forth — required to run a single experiment. To make the proxy's use feasible, we have attempted to provide tools that allow users to create proxies for new applications without writing special-purpose programs. Thus, the computational proxy mechanism provides ways to register an application and define its database interface.

A proxy maintains a persistent record of an on-going computational experiment and makes this information available to users locally. Because proxies model not only the application program, but also the life of a particular invocation of that program (i.e., the process), they constitute a convenient means for querying or recording the status of computations. Thus they are especially useful for computationally intense applications that run for extended periods of time.

Computational proxies offer functionality not now provided in database systems, fulfilling key requirements for database use by computational scientists. Challenges faced in developing the proxy mechanism include:

- Defining an information model to represent input and output formats for the chosen computational applications. These representations should be easy for scientists to understand.

- Building generators and interpreters so that input files to applications can be automatically generated and output files parsed — without programming specific application interfaces.

- Defining an interface between the proxy and the network services.

- Extending the information model to represent information about active (ongoing) application processes. (The domain-level model does not address processes per se, only application programs and their inputs and outputs.)

1.2 Research Design

To render the research goals into realizable objectives, we decided to address one particular computational science in depth. To that end, we selected ab initio computational chemistry as a model domain science and established a collaboration with the Molecular Science Research Center at Battelle's Pacific Northwest Laboratory in Richland, Washington. We worked with Dr. David Feller, a computational chemist who not only uses a wide range of applications in his own research but also develops computational applications and tools for users of those applications. Drawing extensively upon the domain knowledge and vision of Dr. Feller [44, 51, 53, 54], we built a domain model and established requirements for an experiment-management database for computational chemistry. We corroborated our understanding of the computational chemists' needs by interviewing other developers of software for computational chemistry [98, 123, 147, 161], by using standard domain references [145] and by listening in on electronic conversations among members of the computational chemistry community [103].

Focusing the research problem and issues on a single domain adjusted the research issues identified above in Section 1.1.2 to realistic objectives as follows:

- Define a conceptual model covering the data objects of importance to computational chemists when they are using the three most commonly employed computational applications. Where it was not possible to model all objects in use, we chose representative samples.

- Define a computational proxy data structure that adequately modeled the kinds of experiments usually performed. We chose as representative five experiment types ranging over the three applications of interest.

- Define associated computational proxy structures and mechanisms to generate input files and parse output files for the chosen subsets of applications and experiment types.

- Define a facility to register new applications declaratively rather than by writing programs specific to each application.

- Define an application program interface between the proxy and the distributed operating system services, i.e., the network services.

Frequent feedback from our collaborators and other scientific database researchers helped us validate the conceptual model [39, 40, 43, 118]. After translating the conceptual model to an information model and designing the computational proxy data structures, we implemented a prototype system by extending an existing object-oriented database management system to meet our requirements. The implementation was tested with a representative subset of experiment data. Finally, in order to determine if the infrastructure met user needs, we analyzed three experiment management scenarios from the user's point of view — experiment management as now done, experiment management with a repository of past experiments to use as reference in setting up new experiments, and experiment management with both the repository and proxies.

We thus validated the proxy mechanism by testing the clarity and completeness of the logical design and demonstrating its feasibility through a prototype implementation. We developed three distinct user scenarios corresponding to (1) current experiment management, (2) experiment management with a database available, and (3) experiment management with a database plus proxy; we subsequently verified that the experiment management system we designed met user needs and solved user problems identified in the first two scenarios. We illustrated the utility of the proxy by (1) comparing the programming required to connect computational applications to a database both with and without the proxy, and (2) determining the degree to which the proxy is amenable to automated construction or whether customized code must be written for each application interface. We also sought evidence that the data model and computational proxy concept are generalizable beyond the domain of ab initio computational chemistry and implementable in any (object-oriented) database system.

1.3 Major Contributions of the Research

The major contribution of this research is the idea of computational proxies as a construct for managing and modeling computational processes in the database — for effectively solving problems of program interoperability and for improving the accessibility of computational applications. We demonstrate the feasibility of the proxy through a prototype implementation. This research also contributes a conceptual data model for ab initio computational chemistry and mechanisms for constructing connections to applications declaratively rather than programmatically. While it is probably not possible to automate the construction of proxies for every parameter of every application in a given domain, we have automatically generated interfaces for a useful subset of experiment types from our domain. This work automating the construction of proxies contributes indirectly to research in database integration, and in data loading and automatic report generation for object-oriented databases.

In addition to the explicit research results we offer, the prototype database itself constitutes a contribution. The proxy mechanism will enable building a database of past computational experiments. Such a database could serve as an empirical basis upon which a "smart" user interface could suggest parameters for new experiments. It could also be used for estimating resource needs when scheduling application processes.

More generally, this thesis offers evidence of the value of a unifying conceptual model as a first step towards solving problems of program interoperability in the computational sciences. An effective long-term solution to the interoperability problem can be achieved by rewriting legacy applications to work against a common database. However, it is unlikely that program authors will simply agree to convert their respective applications in concert, if at all. It is certainly infeasible for them to do so without experimentation to determine which information models and which database and integration technology to use in the long term. The combination of conceptual and information models and the computational proxy model that we develop here provides a middle ground between continuing with individual, isolated applications and completely rewriting application programs to interface with a common database. Thus our experiment management infrastructure provides a viable migration path from the present complications of stand-alone heterogeneous applications to the achievement of shared distributed database environments for the various computational sciences.

1.3.1 Additional Potential of Proxies

While we have been primarily interested in using the proxy mechanism as the interface between the database and a single invocation of a computational application, we believe that the proxy idea holds additional potential.

Proxies could be used to group several experiments into a chain of experiments, where the invocation of some experiments is dependent upon the successful completion of some previous experiments. In effect, the proxy could link several runs of the same or different applications into one, more complex, computational experiment. Proxies could also be used to schedule a set of experiments, where an input parameter to members of that set varies over some range. Assume, for example, that a scientist wishes to compute a potential energy surface, where the Cartesian coordinates of some of the atoms in the input molecule vary by .5 angstrom increments over a certain range. Such an investigation might require the scheduling, monitoring and file management of hundreds of almost identical experiments, a tedious task even with our computational proxy framework. To use proxies to specify such a set of experiments by varying the input parameters according to some formula would involve a logical and straightforward extension of our mechanism.

A more complex extension to proxies would involve interfacing interactive applications such as such as molecular editors, scientific visualization programs or analysis packages to the experiment database. For example, several recent scientific visualization packages such as Chem3D [24] and CAChe Scientific [23] help scientists build molecular structures to use as input to computational applications and provide a means for viewing results. However, interfacing such visualization packages to applications is still done on an ad hoc basis. The so-called "data flow" solutions [184] to connecting visualization, computational, and analysis packages do help interoperability but are largely process-oriented as opposed to data-oriented. As such, they do not explicitly require that the data to be shared conform to common semantics; rather, the user writes a procedure that passes the data along a pipeline (of sorts) in the expected syntactic form. Data-flow solutions provide short run interoperability between two programs, but scaling up to provide interoperability among numerous programs carries the attendant technical hassles of writing many pair-wise data conversion programs and the risks of error due to differences in the implicit understanding of underlying conceptual entities. The proxy idea could perhaps be extended to provide a common data-oriented interface between a wider variety of applications than we address here.

1.4 Organization of the Thesis

After describing previous research related to scientific data management in Chapter 2, we go on in Chapter 3 to illuminate the need for a computational experiment management infrastructure by describing the current computing environment available to computational scientists. We also establish ab initio computational chemistry as a viable domain in which to search for a general solution to current problems, and define functional requirements for experiment management within this particular domain. The domain-level conceptual model prerequisite to our solution is presented, and the database design and requirements for implementing it are elucidated. Chapter 4 first describes the computational proxy concept, functions, data structures and architecture. We then articulate the proxy's role with respect to the network services and the user interface. We pay particular attention to the need for describing the input and output of computational applications to the proxy in a non-procedural manner, to avoid requiring special-purpose programming for each application. Chapter 5 addresses the prototype implementation of the database and proxy, and our evaluation of the infrastructure is presented in Chapter 6. Chapter 7 concludes the thesis. There, we identify our research contribution and state the research conclusions. We also summarize the lessons we learned that might be applicable to other efforts to integrate applications and outline follow-on work suggested by our research.

Chapter 2

Background and Related Work

While modeling remote, distributed programs and computations within an object database constitutes the major contribution of our work, we view its interdisciplinary aspects as important secondary contributions. Building an infrastructure for computational experiment management has necessarily been an interdisciplinary effort, involving domain scientists as well as computer scientists. Because our infrastructure relates not only to scientific computing, but to several sub-disciplines of computer science as well, we have drawn on previous work in application integration, software systems and databases, as well as current research in scientific computing.

This chapter recounts the research context in which we worked. Within the realm of scientific computing, we describe current applications in computational chemistry and tools used by computational chemists. We also discuss how our research relates to other work in computation management, scientific data management, and experiment management. With respect to current work in application integration, we describe (1) efforts to integrate applications by integrating the user interfaces of those applications and (2) efforts in distributed system support such as domain-specific software architectures and middleware. Finally, we place our work in the context of other research in database systems.

We hope this thesis will encourage researchers in the related fields to address the interdisciplinary issues we raise. We also hope data collected in the course of use of an experiment infrastructure such as ours will yield empirical information about experiments that could aid researchers in intelligent user interfaces, network services, and parallel computing.

2.1 Scientific Computing

Research in scientific computing has traditionally fallen into two categories: algorithm development and the quest for high-performance [96]. Early work concentrated on the problems of finding numerical solutions to continuous equations and of making the resulting computationally intense applications run faster.

As computers themselves became more powerful and as the performance of scientific programs improved, other research efforts began to address the accessibility of scientific computing [66, 109]. The organization of research on important scientific problems into big national projects, such as the Human Genome Project [141] and the Earth Observing System [48], also has pushed scientific applications towards better support of collaborative research. As a result, new areas of research in scientific computing have emerged, such as data management, visualization, and data standards for information exchange. In addition, researchers and practitioners are extending the use of computing not only beyond traditional research and engineering in the physical sciences but also to other fields such as biology. botany, and ecology that have traditionally shunned computational modeling. Our own research in computational science aims to increase both the usability and accessibility of computational tools previously available only to scientists with specialized training and local access to high-performance computing.

Our work draws on areas of scientific computing, in particular computational science applications[1] and scientific data management. Scientific visualization systems and file interchange standards are relevant inasmuch as they are used as tools by computational scientists. We focus the following discussion on current uses of such computing in the molecular sciences, since they are particularly relevant to our work in the domain of computational chemistry.

[1]We define computational applications as those that model physical phenomena and typically run on super computers for extended periods.

2.1.1 Computational Chemistry Applications

Commonly used programs that perform ab initio chemistry computations include Gaussian (which includes a data browser) [62], the General Atomic and Molecular Electronic Structure System (GAMESS) [64], HONDO [50], and MELDF [55].[2] Continued development efforts are improving the performance and accuracy of these applications, and many are being converted to parallel implementations. Our work builds upon this research, which has produced the applications that our infrastructure models. McLean, Replogle and other researchers at IBM Almaden Research Center are pursuing a particularly novel approach to computational chemistry in an effort to alleviate some of the problems our research addresses. They are working on ways to break up the typical ab initio functions as single components so that experiments can be more easily organized into user-specified sequences than currently is the case. To that end, they are defining a file interface and scripting language so that applications contributed to common libraries can easily be used in series [123]. McLean and Replogle's work is file-based, not database-oriented (as is ours). In addition, their proposed system requires the rewriting of applications into functional units that fit their file structures, whereas ours is designed to interface with existing computational applications.

2.1.2 Tools For Computational Scientists

The commercial and public domain programs commonly used by computational scientists define the data modeling entities with which research such as ours must cope when integrating these programs. Thus, those programs are one empirical basis for determining an information model for program integration. The most popular of these programs are candidates for interfacing to the scientist's data repository, whether directly (by being modified), indirectly (by some kind of encapsulation or interface mechanism), or (eventually perhaps) via standard data-interchange structures. In addition to the computational

[2]"HONDO" is not an acronym; HONDO the program was named for John "Hondo" Havlicek, a basketball player for the Boston Celtics in the 1960's. MELDF was originally named for the first initials of the scientists who wrote it: McMurchie, Elbert, Langhoff, Davidson, and Feller; the term has also been known to stand for "Many ELectron Description" [52].

programs themselves, related areas include visualization and computational tools, and physical interchange standards.

Visualization Tools

Numerous molecular editors and visualization tools are available both commercially and in the public domain. Many chemists use graphical molecular display and editing packages such as CAChe Scientific [23] and Chem3D [24] to prepare molecular structure input to computational programs. Specialized toolkits such as Daylight Tools [45] and AVS Chemistry subsystems [184] that allow visual rendering of molecular structures are also available to program developers.

Some efforts to provide persistent stores for visual data are similar to ours in that they rely upon a common conceptual model. Shapiro and Tanimoto's database facility for graphics objects (developed specifically for computer vision researchers) [165], Jirak's Aurora Dataserver [92], and Chu and Cardenas' query system for radiological data [31] all rely upon a common conceptual model in their effort to integrate data from several sources and support scientific work with data that is visualized. These efforts differ from ours in that the objects supported are primarily spatial and relatively static (ours are ongoing computational processes) and their focus is on viewing or querying rather than computation.

Computational Tools

The work of Peskin and Walther at Rutgers [187] offers an excellent example of recent efforts to integrate computational services with a scientific visualization facility. The Scientific Computing Environment for Numerical Experimentation (SCENE) system provides a way to visualize the output of remote high performance physical dynamics calculations so that the scientist can adjust experimental parameters. The SCENE information model is built on a single (vector-like) data structure that can be used to represent physical-dynamics objects. Peskin and Walther have focused on aiding the set-up and analysis of computational experiments, but are currently extending SCENE to allow users to store experimental inputs and results across sessions, and to share data with other users. Both our work and theirs involve computation management. Aside from the obvious difference in application domain, the SCENE effort differs from our work in that its major thrust has been experiment setup and analysis for a particular domain rather than developing a model for application and data integration that could be employed independently of the particular domain as a general solution for computational scientists.

Physical Interchange Standardization

Scientists recognize that exchanging data is critical to their work, and have been active in efforts to develop physical interchange standards. Unfortunately, most development in commercial scientific systems today seems directed towards *reading* other applications' file formats, but not towards *writing* them. The netCDF project (primarily for atmospheric data) and the Hierarchical Data Format (HDF) [63], the Crystallographic Interchange Formats (CIF) [21], and the Flexible Image Transport System (FITS) [134] are example standards to address the problems of physical interchange.

Recently, some physical interchange formats have evolved into database systems of sorts — with capabilities to search, display, catalog, and modify files. While convenient in the short term, evolving physical interchange formats into database systems will prove problematic in the long term unless the standards are based upon agreement at the conceptual level. Most work in physical interchange standards (though critical for low-level data exchange) differs from ours in that our focus is at the conceptual, not the physical, level, and in that our system provides computational support. Some recent efforts in computerized chemical data standards do address the conceptual level, for example the 1993 ASTM Symposium on Computerized Chemical Data Standards [114].

2.1.3 Scientific Data Management

Past work by computer scientists concerning scientific databases has included general characterizations of scientific databases [135, 136, 166], studies of scientific knowledge and data structures [11, 12, 47, 97], and specifications of future data-intensive scientific application systems, such as the Earth Observing System [29, 48]. Other database research areas, such as geographical information systems [191], temporal data structures [74, 162], and statistical databases [68, 100, 149, 150] exhibit important similarities to scientific databases.

The Invitational NSF Workshop on Scientific Database Management brought about forty computer scientists and domain scientists together in March of 1990 to address data management problems facing scientific researchers. Their report corroborates other research on scientific databases: most scientists still manage their information with programs that read and write flat files. Almost every scientific domain has an investment in programs (usually in FORTRAN) that use flat files and have evolved over many years. While database management systems could ultimately improve the reliability, availability, and programmability of scientific applications (just as in other application areas), some scientists now attempting to use databases find that current technology does not match their needs. Even if current technology were adequate, changing from flat file access to database access would involve retraining programmers as well as extensive conversion of existing programs and files to database representations [60, 61].

To our knowledge, until quite recently no efforts have been made to interface database management systems to computational applications.

Data Management for the Individual Scientist

Scientific data management systems are typically aimed either towards supporting the individual scientist or towards making databases publicly available. Important database projects oriented toward the individual scientist's work in the laboratory include three protein structure databases (Compo-OWL [16], BIPED [180], and P/FDM [71]), each representing similar structures, but using different methods. These systems differ from ours in that they involve biological, not chemical, structures and that they do not involve computational applications.

The Molecular Interactive Display and Simulation (MIDAS) system [56, 57, 58, 87] integrates display, storage and manipulation of large macromolecular models. MIDAS database structures take advantage of the considerable redundancy in these molecules to save disk storage space and to provide for efficient real time access. The main focus of MIDAS is the graphical display of large molecules (proteins and nucleic acids), and the special-purpose MIDAS storage manager was developed specifically to make such displays fast enough for real time use. In contrast to MIDAS, our Computational Chemistry Database (CCDB) project uses a commercial database system and targets the computational (not graphical) manipulation of significantly smaller molecules. For viewing molecular structures stored in the CCDB, we propose an interface to existing specialized graphical display systems, such as CACHE, Chem3D, or MIDAS.

Experiment management systems are closer to our own interests, but have traditionally focused on providing laboratory automation facilities or data collection for laboratory apparatus [158]. Commercial systems such as Labview [160] that directly support the chemist's work in the traditional laboratory (at the bench) are now available. Our work, on the other hand, is aimed at supporting the use of computational tools by theoretical and bench chemists, and other molecular scientists.

More recent related work aimed specifically to support experiment management includes Ioannidis, Weiner and Naughton's work modeling complex inputs to a scientific simulation program. Their MOOSE system [77] has dealt with modeling the complex inputs to a scientific simulation program. Our research differs from theirs in that they are building a database management system specialized for experiment management; we are building our tools using a commercial database system. More importantly, our domain model covers a class of applications, while they build a new database schema specifically for each particular application.

Sparr and his associates are developing an experiment management system that can be applied to multiple domains [172]. The system is designed to support general scientific inquiry, and is a long term effort to explore how scientists reason about information. In particular, Sparr aims to develop tools that allow ad hoc queries across different experiments and subdisciplines. A major objective of his work is to discover "new" knowledge by making inferences and connections across experiments and subdisciplines. Our infrastructure, on the other hand, is specifically designed to support the individual scientist in running computational experiments. Hachem's work with temporal data for global change research supports individual researchers as does ours, but focuses on query optimization techniques for data gathered by satellite [148]. We concentrate on support for the scientist performing experiments rather than querying data gathered elsewhere.

Research on personal databases and laboratory notebooks addresses the information about their experiments that scientists need to record [15, 113, 190], and is tangentially relevant to ours. To be of use to the scientist in this way, the CCDB would have to record experiment annotation. More importantly, laboratory notebook research has raised issues of personal privacy and data validation. Ultimately, solutions to these problems will need to be implemented in the CCDB as part of our long term goal of making components of individual scientists' private experiment data available in laboratory-wide or public databases.

Data Management of Public Scientific Repositories

While our research deals directly with experiment data management for the single user, the considerable existing work to support public scientific repositories is indirectly relevant. First, we expect that computational scientists will want to import data sets from public repositories to use as experiment input or as corroboration of computational results. Thus, the data formats supported by public repositories are of concern to us; our data types should be general enough so that we can read from or write to files using those public formats that are in wide use. Secondly, as computational methods become more widely used, scientific communities may wish to make computational results generally available. At that time we will find valuable the current curators' experiences verifying, validating and distributing scientific data[3]. Thirdly, as computational scientists begin to make their data publicly available, bibliographic references to that data will be required, and should be accessible along with it. Thus, we will want to consider how to connect experimental data effectively with bibliographic data. Because we may also consider connecting experimental data with property data, the public repositories of chemical, biological and materials science data are relevant to our work with computational chemistry.

Currently available public repositories of chemical data are either bibliographically-oriented (for example, Chemical Abstract Service) or substance-oriented. A few combine

[3] "Curators" are those responsible for gathering data for public repositories and determining whether a specific data item should be added to the repository.

bibliographic and substance information. Substance-oriented databases may contain property data only (for example, Beilstein [83] or Gmelin [130]), reaction characteristics (the Reaction Access System, REACCS [195]), or aid with structure analysis and elucidation (the Cambridge Crystallographic Database [4, 115] and the Mass Spectral Search System (MSSS) in the NIH/EPA Chemical Information System [82]). Some research has also attempted to integrate bibliographic services into substance-oriented databases [95]. Innovative data structures and methods for molecule representation and for molecular substructure searching developed by public data repositories are valuable models for developing data browsers of laboratory and experiment chemical databases such as ours [35, 36, 37, 65, 195]. In particular, Beilstein's connectivity tables and Lawson numbers [108] provide simple and effective means for building indices based on general representations of molecular formulae.

Biological scientists have been among the first to use public databases, perhaps because genome sequence data is (at least superficially) easily represented as ASCII character strings [18, 22, 59]. While the past four years have seen "only" a 7-fold increase in the number of nucleotides in the centralized DNA databases (from 3 million to 21 million), and the data is accumulating at "only" 7 million nucleotides per year, automated sequencing methods promise to increase this rate by an order of magnitude [188]. Of particular interest is the Human Genome Project [110]; work spurred by the genome project in protein sequencing has seen the development of innovative data structures for memory storage (for use with sequence and structure algorithms) [59, 85, 107, 140]. While many of these systems provide bibliographic references, Futrelle's work in biological text processing endeavors to make textual material itself directly accessible to and manipulable by the biologist [7].

Another discipline where public repositories have facilitated scientific inquiry is materials science [84]. Here, numeric or property databases abound, though their proliferation has raised issues in database interoperability analogous to the issues in program interoperability that we address. Hansen, Maier and Stanley's work addresses issues of database heterogeneity in materials science and parallels our own work in program heterogeneity [79, 80].

2.2 Application Integration Efforts

Of course, problems of data and program interoperability are not unique to scientific computing; they plague users across many domains. As computing becomes an integral part of many business environments, serious efforts to provide application integration architectures are emerging. Current research and development in application integration focuses on both (1) efforts to integrate applications by integrating their user interfaces and (2) software system support such as domain-specific software architectures and middleware.

2.2.1 Integration via the User Interface

The visualization of scientific information is an active research area not only for computer scientists and domain scientists interested in facilitating the work of individual researchers, but also for those working to increase the accessibility of scientific information [91, 155]. Some of these systems are relevant to our own work in that they attempt user-friendly interfaces to programs that hitherto have been accessible only to specialists. Soloway's work in providing a common gateway to scientific applications is an attempt to integrate scientific applications by integrating the graphical interfaces to those programs into one interface [170]. He intends to develop a "digital workbench" to support key activities of computational scientists working in nuclear engineering. The workbench focuses on application integration at the user interface level, providing a uniform interface to a number of commercial tools by "wrapping" the applications. His work differs from ours in that his initial emphasis is consistency at the user interface (ours is on the underlying data). In his system, applications continue to use individualized file structures for storing and viewing data across sessions and experiments; our work has focused on data integration as a first step towards application integration.

The Application Visualization System (AVS) [184] and IBM's EXPLOR [89] are of interest because they provide a facility for linking together programs to perform a series of data transformations or computations. These so-called "data flow" solutions to the program interoperability problem integrate applications at the program interface level by providing a means of visually connecting applications with each other and with data reformatting tools. They do not address the underlying conceptual information or integrate data from applications into a single source. Our work has focused on first providing a common conceptual model of the underlying data and programs, and then working towards generating application interfaces from a database repository.

2.2.2 Software Systems

The software systems research relevant to our own falls into two categories — domain-specific architecture and middleware. Both aim to provide productivity tools for building new applications and for systems integration of new applications. Our efforts are intended to support legacy applications in addition to new development, and focus on solutions specifically for computational science.

Domain Specific Architectures

Current research in domain-specific software architectures (DSSAs) aims to create building blocks for system construction that can be configured by application engineers using domain-specific languages for stating system specifications. DSSAs attempt to make software development more efficient and effective by raising the level of abstraction of programming new systems from that of algorithms to that of domain-specific problems [178]. One emphasis among DSSA researchers is to determine appropriate mechanisms for connecting large granularity modules.

The problems addressed by DSSA efforts are similar to those we address in that both are domain-specific. However, our work is aimed primarily at supporting the data management needs of computational scientists and increasing the interoperability of existing programs, rather than increasing the productivity of development programmers. Two DSSA efforts, Coglianese, Batori and others' in Avionics [34] and Baum, Balzer and others' in Command and Control [10], exhibit an additional similarity to ours in that critical aspects of the domain are represented in a logical model; the logical model yields an architecture that describes a family of solutions. Our project defines a domain model for application programs, inputs, parameters and results; from that model we defined an architecture for experiment management. In addition, Baum's project includes application generators that allow software developers to work in a higher-level, domain-specific language — not unlike our effort to provide a declarative mechanism for generating interfaces to existing computational programs. While DSSA efforts incidentally often enhance the usability and interoperability of heterogeneous systems, that is not their primary objective.

In sum, the primary difference between the DSSAs and our work is that the DSSAs aim to support *developers of new programs*. Our CCDB supports *users of existing (legacy) applications*.

Middleware

The term *middleware* has been used by Bernstein to refer to a system architecture component that fits between the user interface and the application or other system services [14]. Middleware integrates application or system services into a consistent view. Middleware can be geared either to the application level (as a domain-specific information model or application program interface) or to the system level. The client programs for application middleware are user interfaces; the intended client programs for system-level middleware are application middleware programs. Of course, if there is no application-specific middleware, clients of system-level middleware can be user-interface programs.

While domain-specific software architectures address primarily the *software engineering* of applications, research in middleware addresses system integration primarily through the *reusability* and development of applications running on networked heterogeneous computers [14, 67, 72, 86, 131, 154].

Some middleware systems provide low-level support for coordinating remote processes via UNIX system calls (Remote Procedure Call or RPC) [154] or UNIX message passing [67]. Our experiment management infrastructure is similar to system-level middleware in that it provides an application program interface that hides heterogeneity; we differ from those efforts in that our infrastructure operates at a higher conceptual level — that of the computational science domain area — rather than at the algorithm or program level for any application area. In short, our effort is domain-area-specific and could be used in conjunction with middleware systems, to specialize those system services for computational science.

Such middleware as the Parallel Virtual Machine (PVM) [67] and Distributed Computing Environment (DCE) [154] are important first steps towards providing the general-purpose network services that domain-specific infrastructures such as ours might use. However, general-purpose services such as PVM and DCE require efforts such as our infrastructure to provide domain-specific information models for sharing data across programs. The information model facilitates greatly the definition of a domain-specific application program interface so that a single user interface can address numerous programs. Lower-level services such as PVM provide system-level integration services such as network file service and message passing to domain-specific integration efforts. Our infrastructure provides an additional level of support for long-lived computational science processes between the application domain level and the network services.

The current industry-wide effort to solve problems of platform heterogeneity by defining protocols for message passing is a key area of software systems work. Object Management Group's Common Object Request Broker (CORBA), for example, defines the standard protocols for building "Object Request Brokers" (ORBs) to pass messages among applications running in a distributed environment. Vendors are currently implementing ORBs to run on their respective architectures [93, 126, 189]. These future software products will provide message passing among objects residing on different systems and are important to our work in that they will provide the distributed system foundation upon which to implement our proxy infrastructure. They will not replace the proxy infrastucture, which provides not only a domain-specific model of data common to the applications of interest, but also a model of computational applications and processes. The ORB specification does not explicitly address long-lived application processes. In addition, using the ORB, for example, will require the rewriting of all applications as "objects". It is impossible to predict when, if ever, computational science applications will be rewritten to iterface with a common protocol. In the meantime, our encapsulation of these legacy applications provides a migration strategy.

2.3 Database Research

Because our infrastructure for experiment management is data-centered, and has as critical components a domain-specific database and a persistent representation of computational applications and processes, we draw upon current database research for our work. In this section, after briefly addressing efforts at providing migration paths for data and applications from flat files to relational systems, we place our work in the context of other object-oriented database research.

2.3.1 Data Conversion Research

Our task of database support for a class of scientific applications still using flat files was not unlike that facing researchers and developers considering migration paths to relational systems in the late 1970's. We drew heavily upon Shu, Housel, and others' research in this regard: their Data EXtraction, Processing, and REStructuring System (EXPRESS) was an experimental prototype that could access data in a wide variety of formats and restructure it for new uses. Driven by two high-level nonprocedural languages, DEFINE for data description and CONVERT for data restructuring, EXPRESS used program generation and cooperating process techniques to achieve efficient operation. The system's modular structure permitted extensions or adaptation to another environment.

EXPRESS is similar to our work in that we also define high-level nonprocedural languages for the textual data written by application programs. Both EXPRESS and our infrastructure attempt to provide a practical migration path towards database use for legacy applications. The major differences between their work and ours is that theirs was designed to support batch conversion of data from files to relational databases, and ours is designed to provide a way to continue using existing legacy applications by providing an on-line interface to an object-oriented database.

2.3.2 Object-Oriented Databases

Object-oriented database management systems (OODBMS) have grown out of two goals: providing persistence to programming languages [112, 120], and meeting applications needs in computer-aided design, document processing, and multimedia applications that were not well served by record-oriented database systems [3, 75, 119]. Our connections to object-oriented database research are at three points: First, we are among a number of scientific application developers using object-oriented databases to test the modeling power of OODBMS. Second, we are among those identifying migration paths towards OODBMS for existing applications. Third, we are among those extending an object-oriented database system to provide new capabilities, in our case, support for computation services and a data model for programs and processes.

Surveys of data structures used in scientific applications [11, 12] indicate that scientific applications exhibit characteristics similar to the application areas listed in the previous paragraph as "unserved" by traditional relational database technology. Object-oriented databases seem appropriate vehicles for representing the complex data structures of scientific computing. The 1990 NSF workshop on scientific databases [60, 61] showed object-oriented approaches used for protein-structure data, medical research, macromolecules, global change data and scientific visualization. MOOSE [77] uses object-oriented structures to model the complex inputs to a scientific simulation program. Other notable research using object-oriented databases in domains related to ours includes Marr's Genome Topographer for integrating and browsing genomic databases [122], Goodman's laboratory data management system for genome sequencing [70], Shapiro and Tanimoto's computer graphic database [165], and Hansen, Maier and Stanley's system integrating heterogeneous materials science databases [79, 80]. Bourne and Pu's Protein Data Bank Tool (PDBTool) [146] uses object-oriented languages and tools to manipulate complex molecular structures, and is primarily designed to provide support for the new macro-molecular file interchange standard [21].

We know of a few efforts to provide support for converting legacy applications and data to OODBMS, in particular automated tools for loading data into object-oriented database systems. Weiner and Naughton's work introduces algorithms for loading large quantities of legacy data into an OODB; it differs from ours in that their focus is on *data* not *programs* and that the quantities of data with which Weiner deals are orders of magnitude larger than what is needed for our computational applications. Paton and

Gray [138] and Orenstein [137] have also written bulk loaders for OODBMS. The work of Weiner, Paton, and Orenstein is similar to ours in that we, too, must provide a description of the data to be loaded, a tool to parse data according to that description, and a tool to load data according to the OODBMS schema. These efforts differ from ours in that their focus is on data generated in bulk and existing in files, rather than data generated in the context of an ongoing experiment.

We are not aware of any previous efforts to extend an OODBMS to support computational objects, but we hope that other researchers will follow our lead.

Chapter 3

Providing Data Services: The CCDB

Ab initio computational chemistry applications compute chemical properties from first principles alone, using the Schrödinger equation. No empirical data are input to ab initio computational applications, although empirical data are often used to validate results. Both casual and expert users of ab initio applications frequently have difficulty selecting and formatting program inputs. The proper selection of input parameters determines whether a particular invocation of an application runs efficiently or terminates at all. More critically, an incorrect input parameter can engender plausible but incorrect results.

Our collaborators wanted to build a graphical front end to help users select experimental parameters, and believed that a database of successful past experiments would provide useful examples of input parameters for user reference. In addition, since a successful investigation may involve hundreds of runs and many more files, a database that stored experimental inputs and outputs would help users manage their data files. This chapter presents the functional requirements and design for a computational chemistry database (the CCDB) that meets these needs.

Eliciting requirements for database applications and the subsequent database design typically involve the following steps:

1. Develop a conceptual model, i.e., requirements. In the analysis phase, the systems analyst familiarizes herself with the application domain and develops a clear written statement of the application problem to be addressed. To these ends, she may interview key users of the proposed system, read documentation on existing applications, and consult standard references. The analyst prepares written statements

of the goals and objectives of the proposed system, and narratives describing the real world objects about which information will be stored. The narrative describing the domain objects of interest is sometimes called a *conceptual model.* Capturing behavioral aspects of the domain objects was particularly important to us, since we wanted to automate them where possible. The systems analyst often writes down user scenarios for the current *modus operandi* and for how a database user might interact with the proposed system. The problem statement, conceptual model and user scenario(s) provide the mechanism through which the developer and end user decide on what the proposed system should do.

These preliminary narratives are written in the language of the application domain, not in a formal mathematical or computer language. Describing the conceptual model in the domain language is important so that the client can easily verify the analyst's understanding of the system requirements [171]. A complicating factor in writing these narratives arises when more than one domain language is relevant, as would be the case if a database is to be used by both ab initio and bench chemists, or by both chemists and materials scientists. Thus, for example, narratives might describe a given molecule in two different ways (say, in terms of atoms and bonds versus the symmetrical distribution of a space group). One can apply the same conceptual operation to either representation, though the algorithmic details of each might differ. In such cases it is preferable, though not always practical, to choose a single language to describe concepts.

In conceptual modeling, one captures current usage, rather than proposing new scientific paradigms. At times, however, the analyst is faced with terms that have multiple senses and must manufacture new terms to eliminate ambiguities. For example, is an "atomic element" just an atomic number, or does it mean a particular isotope, or some distribution of isotopes? Such a distinction matters in defining what the "atomic weight" operation should yield when applied to an atomic element. Also, we would like to express our models so they can be specialized and adapted for particular applications and sub-domains [114].

2. Develop an information model. During this phase the designer begins to impose a formal structure on the conceptual model prepared in the analysis phase. The resulting information model, sometimes cast into an informal structure such as an entity-relationship (ER) model [28, 169] or an object model [19, 27, 157, 193], organizes information about the entities, attributes, relationships and behaviors that are likely to be implemented in the database. Both the information modeling language and the model itself are independent of any particular database management system. Developing an adequate information model is a crucial, and non-trivial, first step towards building a database. The information model is often used to determine what database technology is needed to implement the system, or whether a particular technology is adequate. If a feature used in the information model is not supported in the data description language of the database management system chosen, then one must encode those features to the database management system. In our case, we determined from the information model that the ObjectStore database system [104] would adequately support our needs, with very little additional encoding.

3. Develop a logical model. The information model is the basis for the logical database design [182], which is typically a database schema. Since we decided to build our database in ObjectStore, our logical model consists of an ObjectStore schema that depicts database classes, the relationships between those classes, and the signatures of database methods within the classes. Note that the logical model is still declarative in nature, and simply specifies more formally **what** the database system is to accomplish. The logical model is the view that programmers have of the database, and it is independent of the data's physical storage characteristics.

4. Develop a physical design. To render the logical model into a physical design, the designer specifies **how** the database will perform its functions. Some aspects of this task are automated; for example, an ObjectStore design tool translates the logical database design (schema) into C++ classes, generating additional classes and appropriate object pointers as required to implement binary relationships. Other

aspects of this task, such as designating indices for accessing particular classes directly, are represented by the designer in the database programming language. Still other aspects of the database design, such as program-level design for methods, may be represented as pseudocode.

At each step in the design process, the design is validated by reviews with typical end users and with the client. Note that we distinguish between *user* and *client*. By *user*, we mean a person who will actually be using the system. By *client*, we mean the person or organization who commissions the system. The user and the client may of course be the same person, or the client may be one of many users.

Our major domain problems were that applications were unusable without extensive specialized training, that even experienced computational chemists were spending too much effort running programs and formatting data, and that experimental results of different programs were not directly comparable. We needed to solve problems of data and program incompatibility, and we felt that efforts to reconcile differences at the lowest (physical) level would be ineffective without agreement at the information model and conceptual levels. There is no point in discussing physical compatibility of data if there is fundamental disagreement on the meaning or interpretation of that data. The applications sometimes use different names for the same information, or the same names for different information; thus, it was not always easy to identify semantically identical data elements across applications, nor to determine whether one data structure was "better than" another, nor to find appropriate mappings from one to another. Because of the critical importance of achieving general agreement among users and programs at the conceptual level, we emphasized the conceptual modeling phase. Thus, we attempted to reconcile incompatibilities of molecular information top-down through the conceptual, information, logical and physical levels.

This chapter describes the design of our database. We first define the conceptual structures of ab initio computational chemistry and outline the functional requirements of a database for computational chemists. After casting conceptual structures into an information model, we identify challenges inherent in reifying computational chemistry information structures as a logical model and a physical database. While details of the physical design are too numerous to include *in toto*, we sketch salient aspects of the physical design and conclude the chapter with lessons learned in building the database. The physical design is treated in greater detail in Chapter 5 where we describe our prototype implementation of the database.

Section 3.1 provides background on the domain of interest along with an initial problem statement. We describe the field of ab initio computational chemistry and explore how ab initio computational chemists use computational applications. We then enumerate the shortcomings of the current computational environment with respect to program interoperability, file management and network services. For novice or casual users, the complexity of input parameters and interpretation of results are also problematical. To illustrate the current typical use of computational chemistry programs we present a user scenario, and suggest how a database of past experiments might solve the problems identified, especially for nonspecialists.

Section 3.2 contains the conceptual model for the database. Section 3.3 describes our refinement that conceptual model to an information model using a notation similar to Chen's entity-relationship modeling [28]. We also identify modeling challenges inherent in the domain. Section 3.4 describes the logical and physical design of the database. We discuss our choice of an object-oriented data model and then show how the information model was translated into a logical design, i.e., an object schema plus the persistent roots of the database. An outline of the physical design of the database follows, and we end the chapter with a brief description of our implementation of a prototype database — an experience that led us to conclude that database services alone will not solve the crucial problems of program interoperability.

Our understanding of the domain (i.e., our conceptual model and requirements analysis) is largely based upon interviews and close collaboration with computational chemist Dr. David Feller. The property data and folklore of the academic subdiscipline were also important sources of data for us. The term *property data* is often used by scientists to refer to data values that are generally accepted within a discipline or subdiscipline. Property data are so well understood that, when cited, they are often not referenced, or, if cited, the reference is a standard textbook. Property data often measure or describe a property of some real-world object; for example, an atom has properties of mass, number of protons, a set of valid electron configurations, etc. Properties are generally thought of as immutable, and a property datum as an established and accepted fact [78, 158].

In addition to collaboration with Dr. Feller and study of the domain's property data and folklore, we also drew upon our own experience reading the user's manuals of computational chemistry packages and running those programs [50, 55, 62, 64]. Informal conversations with other chemists [8, 20, 142, 147, 152, 175, 179], standard texts and papers [102, 111, 127, 174, 168], and non-research oriented accounts of quantum theory [73, 125, 159] supplemented our primary sources. The computational chemistry bulletin board and network, run by Dr. Jan K. Labanowski of the Ohio Supercomputer Center [103], corroborated our understanding of the computational challenges facing chemists. Computational chemists at PNL have reviewed drafts of the conceptual and information models, and our work with the chemists from PNL, CAChe Scientific, and IBM Almaden suggests that our computational chemistry model is an appropriate subset of a more general model of chemistry experiments [114]. Any errors or misunderstanding in this rendition of computational chemistry for computer scientists are of course those of the author.

3.1 Ab Initio Computational Chemistry

The computational sciences bring applied mathematicians and computer scientists together with scientists from application domains to use computers in modeling physical phenomena. Typical computational science domains include environmental science, biology, chemistry and physics. The term *computational science* is often narrowly construed to denote computational algorithms and high performance computing. We believe, however, that the computational sciences have in common *not only* the need for increasing the speed and precision of computation, *but also* the need for promoting sharing of scientific data and better supporting the individual scientist's research activities. Such support includes help in managing an increasingly high volume of data, providing visualization and analysis facilities, and easily-used computer program libraries. Our work has focused on support for scientists' research, exploring how object databases can promote better access to computational science applications and more facile sharing of experiment results from these programs. We address the requirements of one domain of computational science in depth, with an aim to generalize eventually to other computational sciences.

Ab initio computational chemistry uses computerized molecular-orbital methods to calculate chemical properties. Semi-empirical computational chemistry uses (at least some) laboratory observations as input to programs, while ab initio computational chemistry, or *chemistry from first principles*, uses none. We chose ab initio computational chemistry because we believe its data management problems are representative of computational science in general, and because data files are manageable in size and complexity. An equally important factor was the availability of willing and able collaborators.

Ab initio molecular-orbital methods apply quantum-mechanical techniques to molecular structure and energetics, solving the Schrödinger equation to various levels of approximation. From the resulting wave function and associated electron density map, certain observable molecular properties such as vibrational frequencies or electrostatic moments (dipoles, quadrupoles, etc.) can be computed. As early as 1929, scientists realized that quantum chemistry calculations could, in theory, predict molecular structure and chemical properties from first principles, but most believed that calculations precise enough for scientific investigations would be impossible. By the 1950's, approximate methods exhibiting adequate precision had been developed, but these were impractical for molecules of any complexity. Because of this limitation, ab initio methods have traditionally been of interest primarily to theoretical chemists, who use them to determine molecular properties and structure for relatively small molecules. Only semi-empirical methods, which are generally less accurate, can be used for molecules larger than 50-100 atoms. Recent improvements in algorithms and rapid increases in computing power, however, should soon make ab initio methods applicable to larger molecules, including those of interest to molecular biologists. Thus ab initio computational chemistry, once the arcane purview of a relatively small group of theorists, is emerging as a useful tool for bench chemists[1], pharmaceutical

[1]The term "bench chemist" refers to an experimental chemist (i.e., non-theoretician) involved in synthetic work or analysis, such as spectroscopy.

researchers and molecular biologists, as well as theoretical chemists. Just the possibility of dealing with larger molecules, however, hardly makes ab initio methods usable to the non-specialist, who will be loathe to invest the time now required for learning how to use these applications [25, 111, 145, 159].

Ab initio computational chemistry[2] applications run on a variety of platforms, such as IBM RS6000s, Sun SparcStations, and Cray supercomputers. For an ab initio computational experiment to take several days or weeks even on a supercomputer is not unusual, and we characterize executions of this application class as computationally long-lived. Experiments typically create scratch files of one or more gigabytes and results files that are one to two megabytes in size. Current computational chemistry applications are stand-alone packages, typically written in FORTRAN, that each perform a variety of functions. One or more input files define the subject molecule and starting conditions, and specify computational functions and control. Input files are highly structured, and each application has its own command formats. Experimental results are written to output files, again in formats relatively idiosyncratic to the particular application.

Principal inputs to an ab initio application include the atomic components of a molecule, an initial guess at their molecular structure (most often expressed as the location in three-space of the atomic components), and a basis set of functions on which the first iteration of the computation is based. These inputs provide a starting point for an iterative solution to the Schrödinger equation. Many other input parameters can also be specified, depending on the particular application; these usually include the level of approximation to which to take the calculation (level of theory), some maximum number of iterations, and the choice of a particular algorithm. The major outputs of an application include an optimized molecular structure, a total energy value corresponding to that structure, and the corresponding wave function (with its associated molecular orbitals and electron density function or molecular orbitals). By optimizing a structure, we mean adjusting the initial molecular structure with respect to the wave function. From the wave function

[2]From this point on in the thesis, for the sake of brevity, we shall use the term *computational chemistry* interchangeably with *ab initio computational chemistry*, though the former term covers a much broader spectrum of applications.

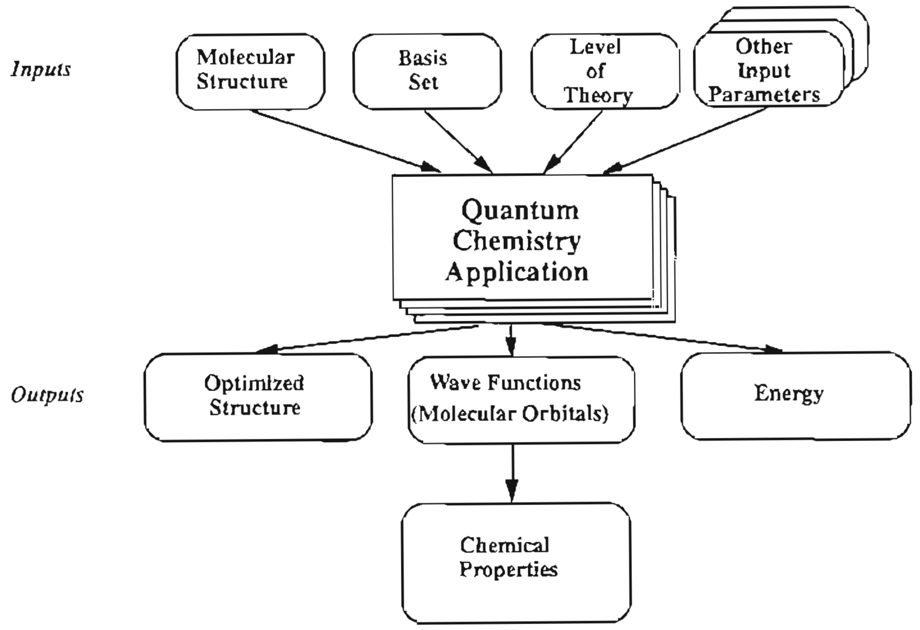


Figure 3.1: Inputs and outputs for computational chemistry codes.

itself are calculated the outputs of interest to non-theorists: chemical properties such as electrostatic moments or hydrophobicity. Thus, for example, given a starting geometry for water, a chemist might calculate its dipole moment. Figure 3.1 gives an overview of these inputs and outputs for computational chemistry applications. We call a single invocation of a quantum chemistry application a *computational experiment*.

During the course of a scientific investigation into a given molecular substance, a chemist usually performs many computational experiments before that substance's structure is adequately determined. A chemist may also study several conformations of the same molecular substance during the course of a single investigation. (Two *conformations* of the same molecular substance would consist of different molecular structures but would involve the same atoms.) For example, in studying transition states from hydrogen and oxygen to water, four computational experiments would be needed to optimize structures for the three different conformations of hydrogen and oxygen atoms. In Figure 3.2 we have plotted the total energy values for: (1) two initial experiments that model the stable states of hydrogen and oxygen molecules, (2) an intermediate experiment that models the unstable state of these elements at the energy level required for the transition, and

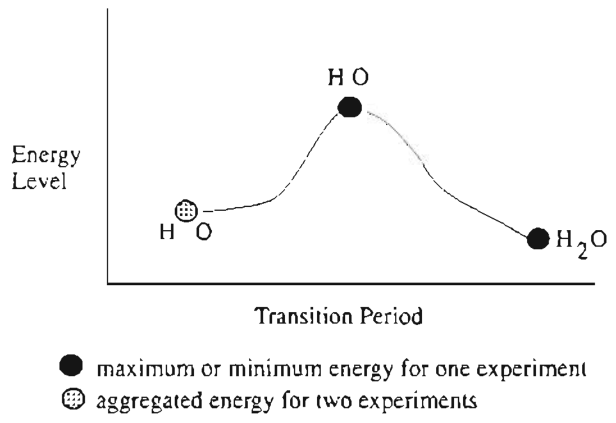

H O
Energy Level
H O
H₂O
Transition Period
● maximum or minimum energy for one experiment
aggregated energy for two experiments

Figure 3.2: Transition of hydrogen and oxygen to water.

(3) a fourth experiment that models the final and stable state of the water molecule. These computational experiments predict molecular properties of the hydrogen and oxygen atoms at the points where the energy values, computed and stored as results of the computational experiment, are minimal (at the three stable states), and maximal (at the unstable state). Note that energy curves are typically not so smooth as those depicted in Figure 3.2, and that an improper or careless determination of molecular structure or some other parameter can cause the computation to converge, deceptively, only to a local minimum or local maximum.

The following sections define the most important inputs and outputs in greater detail.

Molecular Structure

Molecular structure is both an input and an output of computational applications. As an input it is the chemist's initial guess as to the molecule's structure; as an output it is the application's optimization of that initial structure. Computational chemistry applications represent molecular structure in three different ways:

1. Three-dimensional Cartesian coordinates, also called spatial structure. Here, Cartesian coordinates, atomic mass and charge are specified for each atom in the molecule. Cartesian coordinates are the preferred way for communicating molecular structure.

2. Partial structure with symmetries. Molecular structures, as they occur in nature, often exhibit symmetry about the x-, y-, or z-axis. Especially for large molecules, considerable space and computation can be saved by using symmetry conventions to specify the locations of symmetry-unique atoms of the structure. The locations of the symmetric atoms are then calculated using conventional symmetry rules. Specifying just the symmetry-unique atoms can be viewed as a short-hand representation of the molecular geometry. By utilizing the available symmetry, an application can effectively reduce the amount of computer time by a factor of two or more.

3. Internal coordinates, sometimes called "z-matrix format". Here, the molecular geometry is specified using bond lengths and angles instead of Cartesian coordinates. There is no unique set of internal coordinates corresponding to a given set of Cartesians, but by conforming to any one of a number of conventions that fix the positions of the first few atoms in the molecule with respect to a fixed Cartesian axis system, one can define the remainder of the molecule in terms of a variety of internal coordinates [52]. (Because of the difficulties inherent in using internal coordinates, the GAMESS manual refers to these as *infernal coordinates*. If present trends continue, internal coordinates will be rarely used as input to future applications [52].)

Cartesian coordinates in 3-space are acceptable as input to most applications, but some programs require different representations. Sometimes the same program may even require different representations depending on the property to be calculated. Figure 3.3 indicates the extent to which automatic translation among the three representations is possible. Converting between Cartesians and partial structures and from internals to Cartesians or partial structures is straightforward, but arriving at appropriate internal coordinates from Cartesians or partial structures usually requires some human judgement as well as calculation.

Below is a sample textual representation of the molecular structure of water, as found in an output file for the GAMESS program:

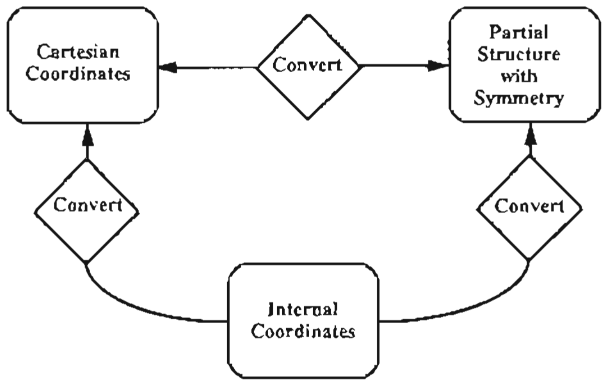


Figure 3.3: Molecular structure representations.

```
ATOM       ATOMIC                       COORDINATES (BOHR)
           CHARGE        X                     Y                      Z
OXYGEN       8.0     0.0000000000         0.0000000000          -0.1239074000
HYDROGEN     1.0     0.0000000000        -1.4304294000           0.9832501000
HYDROGEN     1.0     0.0000000000         1.4304294000           0.9832501000
```

Basis Sets

Selecting a *basis set* for a computational experiment is one of the most exacting tasks for users of ab initio applications. A basis set is a set of real functions over three-dimensional space, and is used for describing the electron density about the molecule. The term *basis function* refers to one of the functions in a basis set.

Basis sets are in effect artifacts used as starting points for the iterative computation that approximates solutions to the Schrödinger equation:

$$H\Psi = E\Psi,$$

where H represents the Hamiltonian operator, Ψ is the wave function of the system (atom or molecule), and E is the energy of the system. Basis functions are used as the basis of linear combination of atomic orbitals to generate molecular orbitals (the wave function for the molecule). A basis set (a set of basis functions) can be thought of as a starting point for the calculation leading to a wave function, but it is not itself a wave function.

Although the solution of the Schrödinger equation does not explicitly require their use, the overwhelming majority of quantum chemistry techniques are formulated assuming that a suitable basis set is available for the particular computation [52]. The heart of the ab initio calculation will be to determine the co-efficients of linear combination (the $c(i,j)$'s below) for the wave function for a particular molecule: $\Psi = N*\Psi(1)*\Psi(2)*\Psi(3)*...*\Psi(n)$, where N is a normalization constant, and

$$\Psi(1) = c(1,1)*\Phi(1) + c(2,1)*\Phi(2) + c(3,1)*\Phi(3) + ... + c(n,1)*\Phi(n),$$
$$\Psi(2) = c(1,2)*\Phi(1) + c(2,2)*\Phi(2) + c(3,2)*\Phi(3) + ... + c(n,2)*\Phi(n), ...,$$
$$\Psi(n) = c(1,n)*\Phi(1) + c(2,7)*\Phi(2) + c(3,7)*\Phi(3) + ... + c(n,n)*\Phi(n).$$

The $\Phi(x)$'s are the basis functions for the molecule in question, and are given as the basis set input to the ab initio application;

$$\Phi(x) = N(x) * coefficient * exp[exponentr^2] + coefficient * exp[exponentr^2] + ...,$$

and $N(x)$ is a normalization constant. Once Ψ, the wave function of the system (atom or molecule), is calculated, then E, the energy of the system, can be calculated using the Hamiltonian operator H.

A large number of basis sets are in popular use, and these can be categorized according to the families of molecules for which they render effective and efficient solutions. Determining which basis set to use as input to an experiment is a complicated process, even for theoretical chemists. For a new investigation, it is possible that no known basis set is appropriate and the chemist will need to develop his or her own.

A number of standard basis sets are usually furnished with an application, and a user can specify by name which one to use in an experiment. Different applications might have basis sets by the same name, but the fact that two basis sets go by the same name is not a guarantee that they are identical. Thus, some research sites prefer their own basis sets, and all applications (to our knowledge) allow the option of including explicit basis sets. Battelle Pacific Northwest Laboratory maintains an electronic *basis set library*, along with programs that generate basis sets for particular computational experiments (i.e., particular molecules and applications).

The basis functions comprising the STO-3 basis set for water, can be constructed from the STO-3G basis functions for oxygen and hydrogen, using the initial molecular structure. Water has 4 shells (1, 2, 3 and 4). The basis functions corresponding to shells 1, 3 and 4 can each be thought of as a single contracted basis function having three s-type Gaussians. The basis function for shell 1 is $\Phi(1) = N(1) * 4.251943 * exp[130.709320r^2] + 4.112294 * exp[23.80886r^2] + 1.281623 * exp[6.443608r^2]$. The second shell is of type L, which is jargon for a group of (s,p) functions with shared exponents. The p functions in the L shell expand into three basis functions, each combining with one of the Cartesian coordinates x, y, and z:

$$\Phi(p1) = N(x) * x * coefficient * exp[exponentr^2] + ...,$$

$$\Phi(p2) = N(x) * y * coefficient * exp[exponentr^2] + ...,$$

$$\Phi(p3) = N(x) * z * coefficient * exp[exponentr^2] + ...,$$

where the normalization coefficient N(x) is the same as that used for the s function in the L shell. Thus, for water, there are seven basis functions.

Once a basis set has been constructed for a given molecule, it must also be formatted for a particular ab initio application before it can be used. Below is a sample textual representation of the STO-3G basis set for water, as found in an output file for the GAMESS application; note that the basis set for hydrogen is given only once; this is because of the symmetry of the hydrogen atoms in water.

```
SHELL TYPE PRIM  EXPONENT        CONTRACTION COEFFICIENTS
OXYGEN
  1    S    1  130.709320   4.251943 (  0.154329)
  1    S    2   23.808861   4.112294 (  0.535328)
  1    S    3    6.443608   1.281623 (  0.444635)
  2    L    4    5.033151  -0.239413 ( -0.099967)  1.675450 ( 0.155916)
  2    L    5    1.169596   0.320234 (  0.399513)  1.053568 ( 0.607684)
  2    L    6    0.380389   0.241686 (  0.700115)  0.166903 ( 0.391957)
HYDROGEN
  4    S    7    3.425251   0.276934 (  0.154329)
  4    S    8    0.623914   0.267839 (  0.535328)
  4    S    9    0.168855   0.083474 (  0.444635)
```

In this example, there are three explicit shells (1, 2, and 4). Shell 3 is the shell for the first hydrogen atom and is implicit because of the symmetry of the two hydrogen atoms in water. Note that for the shell of type L, there are two columns of contraction coefficients; the left corresponds to the s functions and the right to p functions. (The numbers in parentheses can be ignored for this purpose.)

A short general introduction to basis sets can be found in the Appendix of M. Rao's thesis [151], and a more extensive treatment in a paper by E. Davidson and D. Feller [44].

Level of Theory

The chemist uses a level-of-theory parameter to specify the degree of accuracy with which an application is to "describe" the motion of the electrons around a molecule. Higher levels of theory usually result in better agreement with a laboratory measurement, but can cost much more in terms of computer time. Computational chemistry experiments generally become increasingly accurate with "better" basis sets and "higher" levels of theory. Paradoxically, the benefits of a "better" basis set can be nullified unless an appropriate level of theory is selected; with an inappropriate level of theory for a particular basis set one may experience a diminishing degree of accuracy. Choices for level of theory and basis set are thus not independent: an experiment run with a lower level of theory and more primitive basis set can give more accurate results than one run with a higher level of theory and better basis sets if respective discrepancies in the former case cancel each other out. Levels of theory can be paired with basis sets that "work" for a family of molecules, and the pairings can be arranged in an order according to accuracy.

Molecular Orbitals

Molecular orbitals constitute one of the major outputs of many computational applications, and constitute the major result of the solution to the Schrödinger equation. From molecular orbitals applications calculate observable properties. Molecular orbitals are typically represented as a matrix of floating point numbers, organized by atom and shell. The number of orbitals (and hence the size of the matrix) can be calculated from the size of the basis set and the atomic constituents of the molecule.

3.1.1 User Scenario for Ab Initio Applications

The laboratory of the ab initio computational chemist is a computer (more recently, a network of heterogeneous computers) where he or she performs numerical experiments using programs based on quantum-theoretical models. Results consist of chemical properties (structure, dynamics and molecular properties) for a molecule under investigation. Since each application program offers slightly different capabilities, experienced users may employ more than one for a given investigation. We use *investigation* to refer to the collection of computational experiments that a chemist conducts during a scientific inquiry on a particular molecule. The chemist constructs an investigation in an iterative manner, rerunning experiments while adjusting and tuning parameters for the molecule under investigation [52]. A particular experiment fails if it does not converge in the time allotted or if calculated properties do not agree with empirically-measured properties. An investigation is considered successful and complete when the chemist achieves one or more experiments that adequately model the molecule in question. Our research addresses the applications used to compute chemical properties using ab initio methods, but other programs such as molecular editors are important components of the ab initio chemists' laboratory.

The semantic complexity of computational chemistry applications lies primarily in selecting input parameters appropriate for the subject molecule and desired properties, and in correctly interpreting experimental results. A single run may require hundreds of numbers as input; getting even a few of these "wrong" can result in many lost CPU hours or, worse, a plausible but incorrect result. Because applications depend on the same physical theory, they are semantically similar. Input and output conventions, however, vary considerably. This syntactic complexity lies primarily in the relatively diverse and arcane formats of input files and output files, a result of their independent development over a number of years. Unfortunately, even though one application might be preferable to another for calculating a particular property, casual users and even many theoretical chemists use only one application package, rather than deal with learning the idiosyncrasies of several.

We believe that a chemist should be able to use the output of one program "transparently" as the input to another. "Transparently" here means using (perhaps reformatting) the output of one program as the input to another without conscious action on the part of the user. The current *modus operandi* is unfortunately far from this ideal. At best, the chemist must explicitly run programs that translate data from one file format to another. Sometimes, a chemist may even have to write a special-purpose data conversion program. In the worst case, no algorithm exists to reliably translate one data format to another, and each program-rendered translation must be adjudicated by an experienced user.

As we have seen, the semantic and syntactic complexity of ab initio applications is high. The data volume, however, is not particularly large: a single successful experiment results in long-term storage of only about two megabytes of data. Each successful run, however, may be preceded by hundreds of "unsuccessful" runs, each generating two megabytes of short-term storage. In addition, a single run can generate several gigabytes of intermediate data, written temporarily to disk, and used by the application to solve the problem or by the chemist to restart an interrupted run. Battelle's Molecular Science Research Center now generates about 1.2 gigabytes of data per year that are candidates for permanent archiving. This laboratory-wide reference material should be accessible by casual and off-site users. In addition, data from other laboratories will be imported to this repository.

In order to better understand how computational chemists use ab initio applications and to determine specific problems with their use, we have constructed a user scenario. The following scenario involves a typical investigation of a single state of a single molecule.[3]

1. **Define the subject molecule.** Either by hand or using a molecular structure editor, the chemist defines an initial structure for the molecule. He or she may also perform heuristic structure optimizations, though during the course of the investigation this initial geometry will likely be changed. When the investigation is complete and an optimized structure computed, the chemist will probably not want to save this initial structure.

[3]For purposes of clarity, we chose a simple investigation for this scenario. Chemists frequently perform more complicated investigations involving interacting molecules or different states of the same molecule.

2. **Consult previous runs on similar molecules.** The chemist may consult computational and laboratory experiments on molecules of similar structure and properties to determine likely experimental parameters. Because experimental results are rarely filed in a form that can be easily queried, such consultations are typically limited to one's own recent runs or those of a close colleague.

3. **Annotate records of previous runs.** While consulting previous runs, the chemist often annotates his or her copy of previous experimental results with information from the literature [49, 166]. Such annotations, often consisting of references to property data, are in effect pointers to laboratory experiments and can be thought of as evidence confirming the computational experiment.

4. **Choose input parameters for the current run and prepare an input file.** Drawing on his or her experience with the application, or the help of expert users, the chemist considers the particular molecular properties of interest and chooses an appropriate application and associated input parameters. Selecting input parameters is a highly specialized and critical activity. The chemist then prepares an input file for the application.

5. **Perform the experiment.** Many activities comprise performing a computational chemistry experiment:

    (a) Based on initial input parameters and the choice of application, the chemist selects a target machine on which to run the experiment.

    (b) To determine if the experiment is feasible, the chemist usually estimates resource requirements, e.g., how long a single experiment will run and how much CPU time is required. The chemist may then further optimize the molecular structure or modify parameters before scheduling the experiment.

    (c) If the experiment is to be run remotely, the chemist must first transfer the input file to the remote machine.

    (d) The chemist then invokes the application, i.e., "runs the experiment". If the experiment is run remotely, this step also requires logging on to that machine, which may be of a different type than his or her own workstation.

(e) If the experiment takes longer than an hour or so, the chemist usually monitors its execution. which involves inspecting a file into which the application places preliminary results on an iteration-by-iteration basis. The file is produced on the machine on which the experiment is executing.

(f) Once a remote experiment has completed, the chemist transfers results back to the local machine. Alternatively, the chemist might move the files for remote or local experiments to a different area on the same machine.

6. **Analyze results, adjust parameters and rerun the experiment, until it runs successfully.** The chemist will likely invoke the application many times during the course of a single investigation before being satisfied with the results, repeating items 5 and 6. It may also be scientifically advisable to run the computational experiment using several applications to validate results. However, such comparative runs are rarely made in practice because of file format differences among applications.

7. **Make public and archive the results of the experiment.** Once the chemist has successfully completed an investigation, he or she publishes the results, either locally to colleagues working on the same or related projects, or formally in a scientific report or journal. Often, the chemist makes input and output files publically accessible to colleagues, so that results can be consulted or corroborated. The chemist must also clean up the "laboratory" after the investigation; here that includes deleting, archiving, or compressing unsuccessful runs.

Difficulties abound in the scenario above, even for highly trained theoretical chemists:

1. Even if a chemist uses only one application on one computer, work on even relatively small molecules is hampered by the number of files he or she can effectively name, store, search, and manage. Running experiments on molecules significantly larger than 50 to 100 atoms (which is expected in the next five years) will exacerbate these data management problems:

   (a) A much larger set of sample experiments will be relevant to the process of choosing input parameters because larger molecules typically have more substructures.

   (b) Larger molecules will likely require more application runs, and hence necessitate managing more input and output files.

   (c) Larger molecules have more complex structures and will likely require larger basis sets, and hence larger input and output files.

2. Consulting previous runs on similar molecules is an *ad hoc* process. Because there is no electronic index to experiments already run, finding successful previous successful experiments relevant to a new study is difficult. Furthermore, because experimental results from different programs are not in consistent format, a chemist often limits consultations to applications using similar formats. Finally, the chemist usually works from copies (on paper or in data files) of the previous runs, rather than consulting the original (and presumed more correct) electronic record of the experiment.

3. Little or no information associating application inputs (molecular structure and parameters) with results is captured from the experimental process in a way directly applicable to later experiments with other molecules.

4. Since at least some of the computational chemistry programs are little more than research prototypes, it may be difficult for a bench chemist to properly prepare input, due to the combination of mathematical sophistication and research orientation of such applications.

5. While much of the *semantics* of one application is transferable to another, the *syntaxes* are not transferable. If a chemist wants to compare outputs from several different applications, he or she must perform tedious data conversions. Figure 3.4 shows the format conversions needed in order to compare a property calculated by computational chemistry application A with properties calculated by applications B and C. Assuming that the chemist understands how to invoke the same calculation across the three applications and specify the same level of theory, at a minimum

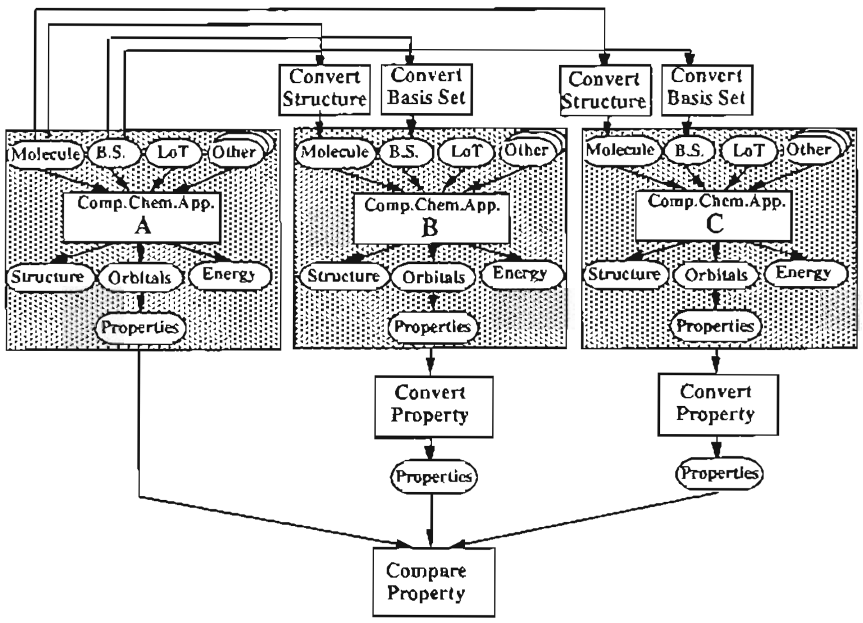


Figure 3.4: Syntactical complexity of computational chemistry applications.

he or she must convert the molecular structure and basis set formats for program inputs, and convert the properties produced by applications B and C to the format of the property value produced by application A.

6. Syntactic differences among applications also make it difficult for chemists to use the inputs from experiments using one application as models for successfully designing experiments using another application.

7. Because computational experiments require highly variable amounts of system resources, the applications run on a range of platforms. A chemist who wishes to choose an appropriate target machine for an experiment must be familiar with the network and operating system idiosyncrasies of that machine, as well as knowing how to access the machine remotely. We call this distributed, heterogeneous characteristic of the computing environment *architectural complexity*.

The above problems fall into categories common to other computational sciences [43, 60, 61, 158]. They include the management by hand of many large files spread over several physical computers; the semantic, syntactic and architectural complexity of the applications themselves; and the lack of data interoperability among the applications. Providing a database to hold input and output files of computational experiments ameliorates some data management problems, though it does not address problems of architectural complexity or interoperability.

A public database of past runs could be used by chemists to consult previous experiments on similar molecules. Of course, if a chemist wanted to use previous successful experiments on, say, all alkanes to determine which application and basis set to use for determining a particular property for a specific alkane, output files would have to be indexed by molecular substructure. To allow searching for molecules similar to the one under study, the database could hold templates of atomic substructures, each of which defines a "family of chemicals" and could be used as an index into molecular structures. Once defined, a new molecule or template could be placed within a chemist's personal database and reused.

To set up a new experiment, one would want to know the specific machine on which the previous samples were run, with which version of which application, when it was run, and by whom. Even with the information above, determining which method was used to perform the calculation usually requires an understanding of the application. Such metadata, while necessary for correctly interpretating results or applying previous work to a new investigation, is not necessarily explicitly contained in either the input or output files.

While such a properly indexed database *repository of input and output files* as described above would be useful, a *database of experiment data* that renders the *data* contained in input and output files into a common format would be even more helpful. Such a database could support the needs for public data as well as provide working data to individuals, and the same entities (molecules, basis sets, experiments) can participate in multiple collections to support varied searches and personalized subsets of data.

Such a database could also hold experimental inputs and results in canonical form, or contain functions to display in comparable formats data that are stored in different formats, thus alleviating some syntactic incompatibility of applications. Outputs from one application could be more easily compared to outputs of another application, or used as inputs to new experiments. Equally important, the data in a database could also be used by applications that assist in selecting previous experiments relevant to a new experiment, and to develop initial inputs and parameters to new experiments based on them.

A database that can display inputs and outputs of ongoing experiments and past runs in comparable form simplifies the user scenario given above as follows:

1. **Define the subject molecule.** Instead of defining a molecular structure from scratch, the chemist can consult the database to find a similar molecule to use or modify as an initial structure. During an investigation, the initial geometry will likely change radically, and the database can keep track of versions of molecular structure, without requiring the chemist to think up and remember a series of appropriate file names. The advantages of using a database separate from the file-based databases supplied with most molecular editors are that scientists can more easily share structures if they prefer to use different editors, the database can integrate experimental results with molecular structure, and optimized molecular structures can be more easily annotated with metadata that indicate how they were derived.

2. **Consult previous runs on similar molecules.** The chemist may now query the database for previous successful computational and laboratory experiments on molecules with similar structure and their calculated properties. Consulting a database for similar runs is a considerably easier task than gathering electronic files or printouts. Because the chemist may not be familiar with all of the computational programs or apparatus on which the selected experiments were run, associated information (metadata) can be available to help interpret input parameters and results. Data may be physically stored in the database, archived at the site, or resident at another location, but the chemist can be presented with a consistent view of data irrespective of physical location or the format or machine on which the data were generated or stored.

3. **Annotate records of previous runs.** A chemist's annotations on an experiment

may include additional information, such as "property data" [49, 166]. With a shared database of experiments, annotations can be made publicly available upon release from the author. Without a shared database, such annotations are made on *copies* of the experiment and are usually lost to other chemists who might later also use those experiments as models (even if the copies are electronic copies).

4. **Choose input parameters for the current run.** Using the experiments retrieved above as examples, the chemist can better determine input parameters for the new experiment.

5. **Run the experiment.** In addition to containing experimental data, i.e., application inputs and outputs, a database of past experiments should contain metadata about the experiment, such as the application invoked, the machine on which the experiment was run, and information about the resource use. Experiments on molecules of similar structure and input parameters are likely to require similar resources. Thus, sampling previous experiments could help the chemist select an appropriate target machine on which to run the experiment and estimate how long the computational experiment will run and (if relevant) how much it will cost.

6. **Analyze results, adjust parameters and rerun the experiment.** Since the database must provide capabilities to represent results from different programs and to view experimental results of past (confirmed) experiments, it could also be used to view results of completed but not yet confirmed experiments. Because results are captured in comparable form, results of ongoing experiments can be directly compared to each other and to results of previous successful experiments, thus helping users analyze results.

7. **Publish and archive the results of the experiment.** The database can more easily effectuate consistent change in status between *private* and *public* data than can the current *ad hoc* file system.

Such a database would be greatly enhanced by an expert system to help build input files to computational experiments. An intelligent front end would use heuristics and data from previous experiments to guide the chemist in his or her input parameter selection. Such a system is under development at Battelle Pacific Northwest Laboratories by D. Feller [51]. The database and expert system complement each other; the interface is an expert system, with the experiment database forming a portion of its knowledge base. Rather than accessing the database directly, we envision the chemist presenting experimental requirements at a relatively high level to the expert system, which itself would use a combination of heuristics and empirical data (from the database) to help set up an experiment.

The scenario above assumes that a person enters experiments into the database. Simply having a database available does not guarantee that users will put past runs into it. In fact, because of format conversions, entering past runs may be a non-trivial task. The remaining sections of this chapter describe our experience designing, building and populating a database of computational experiments.

3.2 A Conceptual Model for Ab Initio Computational Chemistry

In Section 3.1 we observed that some of the difficulty of using computational chemistry applications can be alleviated with a repository of past experiments. In this section we give a conceptual model for that database, identifying and describing the entities of interest. A conceptual model, though not always explicitly articulated, is the first step towards constructing a database and provides the basis for further design. We couch our conceptual model in the language of the domain of interest, avoiding database and computer science terms where possible, so that users and database specialists can contribute equally to the effort. Recall that the conceptual is the first of three increasingly formal models in our design process: it is followed by information and logical models. The physical model, which is the fourth and final will concretize the logical model in a particular database management system.

In the conceptual modeling phase, the aim is to capture domain concepts, including entities, relationships between entities, and functions performed by and on the entities. In classic application software engineering, the conceptual modeling phase of database design proceeds concurrently with the requirements analysis phase of system design [17, 171, 192]. In our conceptual modeling, we tried to capture and formalize current usage, rather than propose new terms. Unfortunately, as noted earlier, we were faced with situations where a given term has multiple senses, and we sometimes had to manufacture additional terms to distinguish those senses. An "atomic element" may be just an atomic number, or a particular isotope, or a distribution of isotopes. Such distinctions matter in defining what the "atomic weight" operation should yield when applied to an atomic element and must be clarified and made explicit.

An additional goal we set was that the model should fit with more general chemistry models so that it can later be specialized for future applications and subdomains, or generalized to include other domains of chemistry such as semi-empirical computational chemistry and laboratory chemistry.

In spite of the difficulties in developing and expressing a conceptual model, we believe that an acceptable and generally understood model is a critical first step towards resolving data incompatibilities. Efforts to reconcile differences at the physical-format level will be ineffective without agreement at the conceptual level. There is no point in discussing physical compatibility of data where there is fundamental disagreement on the meaning or interpretation of that data. Of secondary interest is the role of a conceptual model in helping resolve semantic ambiguities when using data and programs across larger (interdisciplinary) boundaries. For example, our computational chemistry conceptual model could be of help to molecular biologists using computational chemistry programs or as a documentation tool to software engineers developing software for computational chemists.

Below, we describe objects of interest for the computational chemistry database. Those objects of interest that we chose to include as database objects are *italicized* on first use in each subsection. Important relationships between objects are expressed in **boldface**, and attributes and behavior of objects are <u>underlined</u>.

3.2.1 Chemist and Experiment

The model includes simple identifying information about each *chemist* whose *experiments* or *basis sets* are included in the database, so that scientists can contact each other.

Experiments **have as subject** a particular *molecule*, and are arranged in a type hierarchy. An experiment, perhaps the collaborative effort of more than one chemist, is either a *laboratory experiment* or a *computational chemistry experiment*. The following information about each experiment is included in the database: a user-defined textual experiment name, (a run title or other textual annotation), date-begun, date-completed, site where the experiment was conducted, a list of citations for the data used as a source of the experiment, and a list of publications where results of the experiment were reported. A chemist usually organizes his or her experiments by subject and investigation, and within subject and investigation, by the date begun.

Computational Experiment

A *chemist* **performs** a *computational (chemistry) experiment* on a given *molecule* **using** an *application program*, specifying **parameters** such as *molecular structure*, *basis set* and *level of theory*. In the course of computationally determining a molecule's structure, a chemist performs several experiments on that *molecule*. Only a few (perhaps one or even none) of these experiments will ultimately be archived as successful. Until deemed successful, each computational experiment should be identifiable as part of a single *investigation* and marked as private and available only to the performing chemist(s).

Among the numbers that appear in the output are some that correspond to physically observable properties of a molecule, and some that are simply artifacts of the equations that were solved. The most important output value is the total energy of the molecule. An important artifact of solving the equation is the set of *molecular orbitals* (sometimes called the electron density), an array of numbers whose length varies approximately with the square of the number of atoms in the molecule.

Molecular properties are generated directly from the detailed *molecular structure* and molecular orbitals derived in an experiment. Thus, a computational experiment **predicts** *molecular properties.* An experiment is successful when these computed properties "agree with" properties measured in *laboratory experiments*, e.g., by x-ray crystallographic methods. This agreement is a matter of human judgment, not a determination that can be made automatically by the computer. Where such agreement is noted, we say that a computational chemistry experiment is confirmed by a laboratory experiment.

One measure of the accuracy of the computational experiment is the amount of numerical error. The experiment should include some measure of the cumulative error introduced by each state of the calculation, from which a rudimentary stability analysis can be performed upon request.

Laboratory Experiment

A *laboratory experiment* is conducted in a traditional chemistry laboratory by a bench chemist, using *laboratory apparatus* such as a mass spectrometer or a cloud chamber, and producing a value or values for a specific *observable property.*

3.2.2 Aggregations of Experiment: Investigations, Suites and Personal Sets

An *investigation* is an aggregation of *experiments* that groups a series of computational experiments together for convenience of analysis.

During the course of a single scientific investigation a chemist may study several conformations of the same *molecular substance*; a conformation is a particular *molecular structure.* A *suite of experiments* is an aggregation of *computational experiments* that groups conformations of the same molecular substance into an ordered collection. Another name for a conformation is *chemical state.*

3.2.3 Molecule

A *molecule* is the subject of one or more *laboratory* and *computational experiments*, and can be identified by name, chemical formula or through its corresponding *experiment.* Some intrinsic properties of a molecule, such as name and chemical formula, are independent of any particular experiment. Some properties, such as *molecular structure*, are highly dependent on a particular experiment; other properties are functions of the atoms or isotopes comprising the molecule, and can be deduced by looking at the molecular structure. For our purposes in ab initio computational chemistry, a molecule is determined by its molecular structure; we consider two molecules with the same name and molecular formula, but different molecular structure, to be different molecules. Because such molecules are thought to be "the same" both in common parlance and for some scientific purposes, we say that they are the same *molecular substance*.

Some of these intrinsic molecular properties, such as molecule name and chemical formula, are not needed for the actual computation, but are important for the chemist's use in retrieving data on molecules.

Molecular Structure

For computational experiments, the most important property of a *molecule* (indeed, its defining characteristic) is its *molecular structure*. Molecular structure specifies the molecule to the application program, and consists of a list of the component atoms, along with the location of those atoms in 3-space. Thus molecular structure is a list of *atoms*. An atom has molecular weight, molecular charge, and Cartesian coordinates x, y, and z.

A symmetry indicator denotes whether the molecular structure is fully represented, or partially represented with symmetries. If the latter, then only the symmetry-unique atoms of the structure are specified, and symmetry functions calculate the locations of the symmetric atoms using conventional symmetry rules.

There are no alternative formats in our conceptual model for internal coordinates (sometimes called "z-matrix format") or for representing molecular structure as bonds between atoms. The latter would be useful for graphically displaying molecules.

Conversion operators Cartesians_to_partials and partials_to_Cartesians convert a molecular structure given in Cartesian coordinates to partial symmetry representation and vice versa.

A chemist performing a suite of experiments for different conformations of a molecule will create several instances of that molecule, each with a different molecular structure. Thus, as shown in Figure 3.2, each experiment in a suite of experiments looking at different states of a molecule will have as subject a different molecule (and hence molecular structure) even though in common parlance a chemist would say that all those experiments deal with the same molecular substance. The subjects of experiments relating to certain chemical states may not in fact be a molecule in the traditional sense, but simply a collection of atoms grouped together in one structure for the purposes of a particular experiment. The example experiment of Figure 3.2 involving the unstable state of water is a case in point: the subject molecule is in effect merely a collection of hydrogen and oxygen atoms.

Molecular Template

To browse the database for candidate input parameters to an experiment on a particular molecule, a chemist will want to examine previous successful experiments on molecules with a similar structure. We use the term "molecular family" to denote molecules with similar structure. Molecular families are not disjoint; any given molecule can match a number of templates and thus belong to a number of families. *Molecular templates* represent ways in which chemists mentally group *molecules* with similar structure into *molecular families.*

A molecular template is a different kind of entity than any yet described in our conceptual model. We coined the term *molecular template* to refer to the way that chemists specify instances of structure or formula that could be used to search the database for molecules with similar structure or formula. A molecular template specifies a flexible search on molecular substructure; it is itself a structure like a molecule (technically more like a molecular substructure) that can be matched against molecular structures in the database. For example, if a user wishes to find a basis set to use with ethane, he or she can define a template for alkanes such as "C_nH_{2n+2}" that would match similar molecules such as methane containing this substructure.

3.2.4 Computational Chemistry Application

An *application* is an ab initio computational chemistry program or collection of programs. Each computational experiment uses an application; we also say that a computational experiment is run using an application. We sometimes refer to an application as an *application program* or *"code"*. A computational chemist's application program is analogous to the bench chemist's laboratory instrument; each is an *experimental apparatus*.

Computational chemists want to know precisely the nature of the apparatus on which they run an experiment; knowing the application name, application author(s), and authors' address is necessary but not sufficient. The *version of the application* and *computer* used to run an experiment may be necessary to track down resource utilization details or experiments run with an application that contained an implementation error. The chemist may even want to know which version of which programming language and which computer was used to compile the application. Most theoretical chemists want to know the particular algorithms used in the application, and often even further details about the numerical implementations of those algorithms.

Figure 3.5 depicts information that is relevant in some contexts about a particular computation, and shows how three taxonomies relate: applications and application versions, computer platforms and particular computers, and programming languages and versions of programming languages. An *installed version of an application* is a *compiled version of that application* installed on a *particular computer*. A *particular computer* is an instance of a generic *computer platform*. The installed version of the application has been compiled for that *computer platform* using *a version of the target (programming) language*. A version of that programming language has itself been installed on *a particular computer* that is necessarily an instance of the same platform of the particular computer on which the application has been installed.

For example, a given computational experiment might utilize the GAMESS package, release 2.0, compiled under version 3.21 of the Sun 3 FORTRAN 77 compiler. Important meta-information needed at various levels of the taxonomy includes basic formatting and functional capability of the application at the application level, changes in format at the release level, and performance characteristics at the compiler version and platform levels.

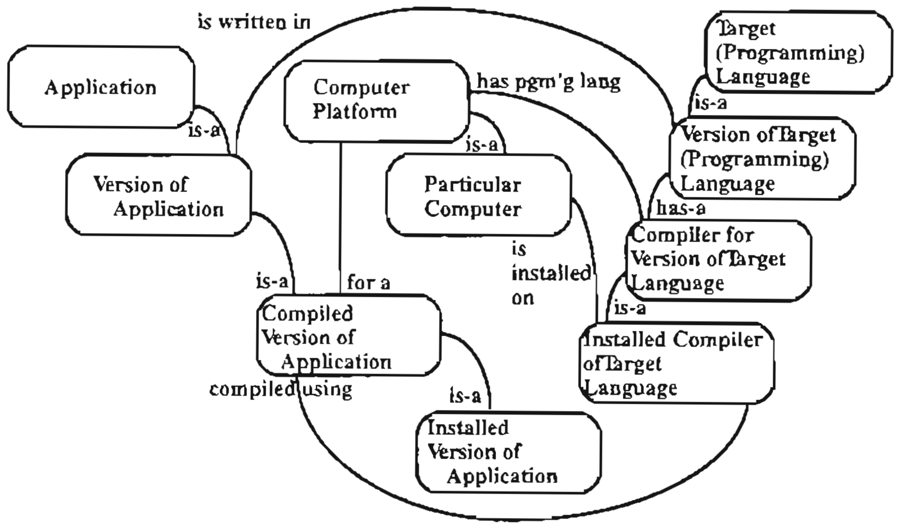

is written in
Application
is-a
Version of Application
Computer Platform
has pgm'g lang
is-a
Particular Computer
is installed on
Target (Programming) Language
is-a
Version of Target (Programming) Language
has-a
Compiler for Version of Target Language
is-a
Installed Compiler of Target Language
is-a
for a
Compiled Version of Application
compiled using
is-a
Installed Version of Application

Figure 3.5: Application taxonomy.

3.2.5 Basis Set

A *basis set* is a set of real functions over three-dimensional space. A *computational experiment* uses a basis set as a starting point for its iterative computation. A user can specify by name which basis set to use in an experiment. Some research sites have their own basis set libraries, and individual basis sets can be retrieved by name. A basis set is usually **authored by** a *chemist* and is indexed for retrieval by author. Basis sets can be **categorized** according to the families of molecules for which they have generated successful computational experiments.

A basis set includes a list of coefficients and exponents for every contraction for each atom that the basis set supports. An operation to generate *a basis set instance* for a particular experiment entails producing a list of the contractions for each atom for the molecule in question, in the format specific to the application chosen.

3.2.6 Basis Set Library

A *basis set library* is collection of *basis sets*. Such a library may be site-specific and hold the basis sets that are to be used at that site, as in the case of the basis set library available to PNL chemists. A library of basis sets may also be made available within an application, as is the case with the Gaussian application [62]. The basis set contains not only basis sets themselves but also information about each basis set that helps determine its applicability to a particular experiment, e.g., for a given molecule, application, experiment type, and level of theory.

3.2.7 Level of Theory

Level of theory corresponds to the degree of accuracy with which the motion of the electrons around a molecule is described in a particular experiment. A level of theory is characterized by its name. Since application packages use different names to designate the same level of theory, a level of theory name conversion operation is required for each application package.

Levels of theory **can be paired** with *basis sets* where these parameters have been used together in successful experiments. A pairing can be **retrieved** by the molecular family associated with the subject of those experiments.

Molecular Orbital

A *molecular orbital*, also called an electron density function or a wave function, **is computed by** a *computational experiment.*

The molecular orbital must be reported with the *molecular structure* and the *basis set.* The molecular orbital can be represented as an n by n matrix, where n is the number of basis functions in the corresponding basis set.

3.2.8 Observable Property

Final energy and *observable property* values are the two major results of interest for computational chemistry applications. Final energy is a scalar value, and may be a minimum energy, or a saddle point.

An *observable property* for a given molecule is a function of an experiment, not the molecule itself, and is said to be **produced by** an experiment. **Measured by** and **predicted by** are synonyms for **produced by** that are often used in the contexts of laboratory and computational experiment, respectively.

Strictly speaking, the observable properties produced by computational experiments are calculated from the wave function, but chemists associate an observable property with the experiment, not the molecular orbital.

A computational chemistry experiment **is confirmed by** a laboratory experiment (or vice versa) when an observable property calculated by a computational experiment **matches** an observable property measured by a laboratory experiment. Whether or not two observable properties match is a matter of judgement — **matches** cannot be automated as a function by the database.

As of this writing, we have modeled each observable property as a separate database entity. A general representation for property would be preferable so that new properties could be added to the database without changing the schema.

3.3 The Computational Chemistry Information Model

The previous section described objects of interest to our database design effort at the conceptual level, in the language of the application domain. In this section, we describe our efforts to extend the conceptual model to an information model — the first step towards formalizing a database design.

Information modeling has been an active area of database research and development, particularly since Chen's classic work in 1976, which introduced the entity-relationship model [28, 124, 169, 181, 182]. Given current needs to extend databases to handle new kinds of applications, other researchers are working to provide more direct transitions between information modeling and database design, and to incorporate an "object orientation" into information modeling constructs [19, 157, 177, 193].

An ideal modeling methodology would provide an effective means of describing candidate entities, their attributes and behavior. Unfortunately, the semantic modeling methodologies that attempt to integrate entity-relationship modeling with dynamic and functional (also known as behavioral) modeling are, in our estimation, immature. Even though capturing behavioral aspects of the conceptual model was particularly important to us, we found no effective means of representing behavior along with data structure at the information-modeling level. Thus, the information model presented in this section contains only a model of data structure; only during the next database design step, the casting of the information model into a database schema, do we explicitly describe the behavior of the objects.

In Section 3.3.1, we present our information model and describe our information modeling conventions. Section 3.3.2 discusses information modeling issues that surfaced during this phase.

3.3.1 An Entity-Relationship Model for the Database

We present here the entities and relationships identified during the conceptual modeling phase. The entity-relationship model in Figure 3.6 depicts the overall structure of our information model for computational chemistry experiments. An entity is represented as a box with rounded corners. Aggregations of entities are depicted as multiple entity boxes, and labeled with the keyword "grouped into". Relationships between entities are represented as labeled lines between boxes representing entities; cardinality of greater than 1 is indicated by a black dot. Is-a relationships (superclass–subclass relationships) are denoted by thick lines. Sometimes a relationship between a superclass and a related class is renamed when one speaks of the relationship between the subclasses and the related class; such synonyms are indicated as dotted lines.

In our diagrams, each entity box represents a type of object and does not require that there be an extent or collection of all instances associated with the type. Where there is an expectation of iterating over instances of an entity type, we have indicated a named *entry point* with a dashed arrow. In mapping from the information model to a database schema, each entry point may be rendered as one or more named collections. For example, each user of the database may have his or her individual collection of experiments, namely a *personal set* of *experiment*.

With the exception of the complex objects representing applications (described above), and Molecular Orbital and Basis Set (described below), each of the entities we identified can be mapped directly onto a logical database model, or schema. The instance diagram in Figure 3.7 reflects the major attributes for each entity.

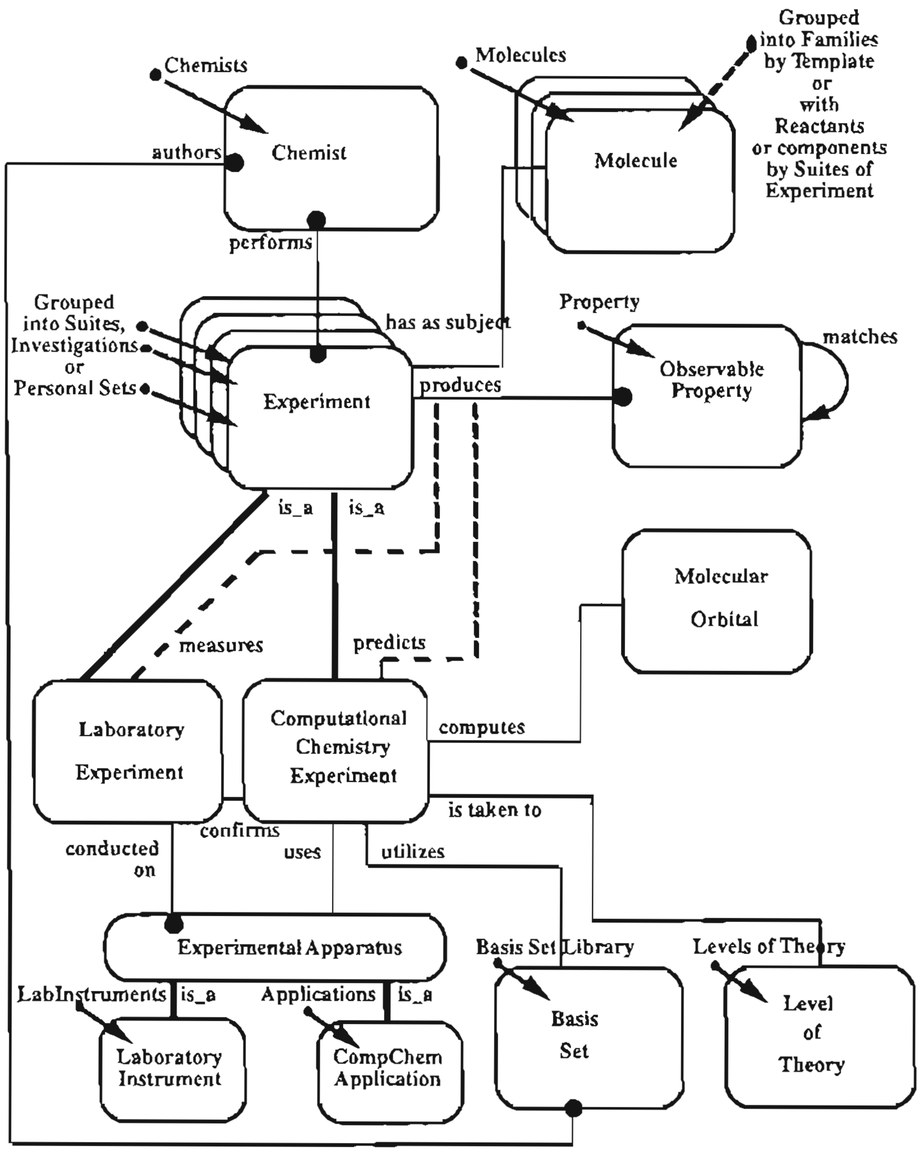


Figure 3.6: Computational chemistry information model.

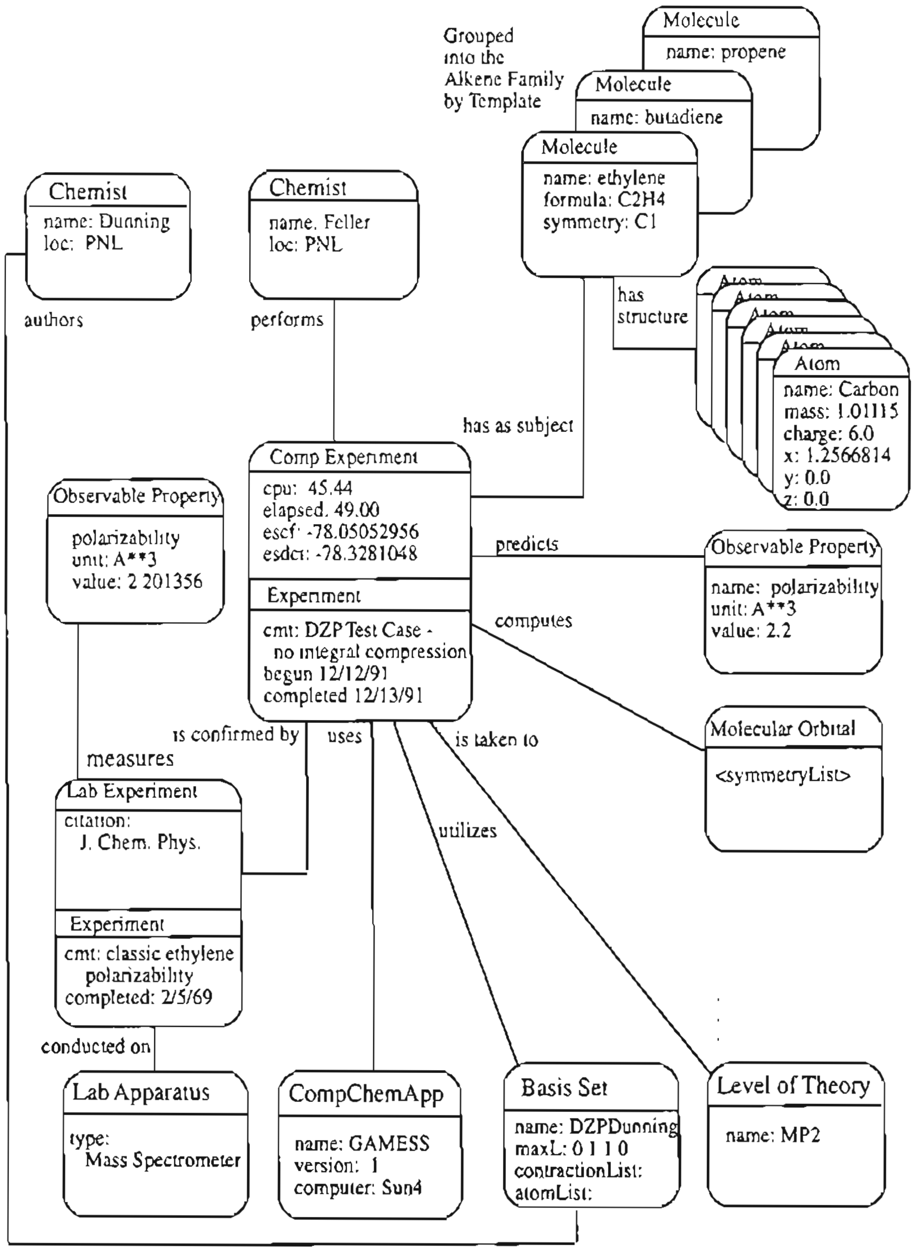


Figure 3.7: Computational chemistry instance diagram.

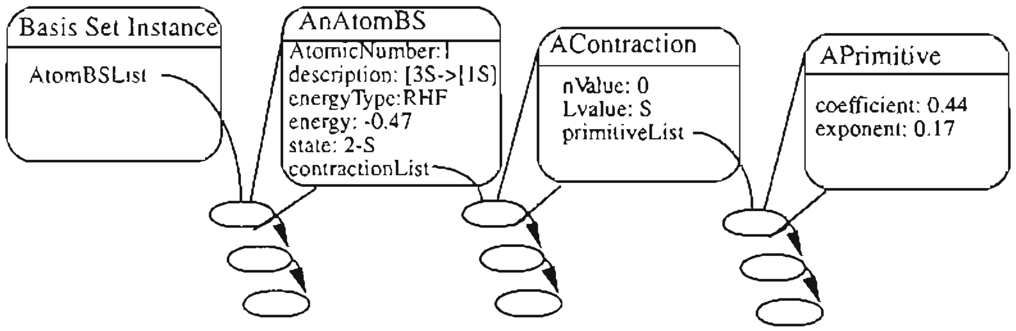


Figure 3.8: Basis set.

Our entity-relationship model above is simplified in that *basis set* and *molecular orbital* entities have been presented as single entities. In fact, each is a complex object (i.e., composed of other objects) and each can be modeled using a series of list structures, as shown in Figures 3.8 and 3.9. In these figures, list structures are shown as a series of ovals connected by arrows; the first element in the list is "expanded" to show the type of the list. Thus, for example, AtomBSList is an attribute of the Basis Set Instance entity; each element in AtomBSList is of type AnAtomBS. (Alternatively, one might represent these structures using a matrix type. Thus, the symmetry values for molecular orbital could be column headers, and molecular orbital coefficients could be matrix data "under" the columns.)

Modeling the relationships between laboratory and computational chemistry experiment also requires a more complicated and interesting structure. Recall that a laboratory experiment **confirms** a computational chemistry experiment when an observable property predicted by the computational experiment **matches** the observable property **measured by** that laboratory experiment. Directionality of the *matches* relationship is critical, since it indicates (in this example) the directionality of the derived relationship **confirms**, i.e., that the computational chemistry experiment confirms the laboratory experiment (and not vice versa). The **measures** and **predicts** relationships are synonyms for the **produces** relationship. Which term use (**measures** or **predicts**) depends on whether the experiment in question is a laboratory or computational experiment. Figure 3.10 shows explicitly which relationships are explicit and which derived: explicit relationships are

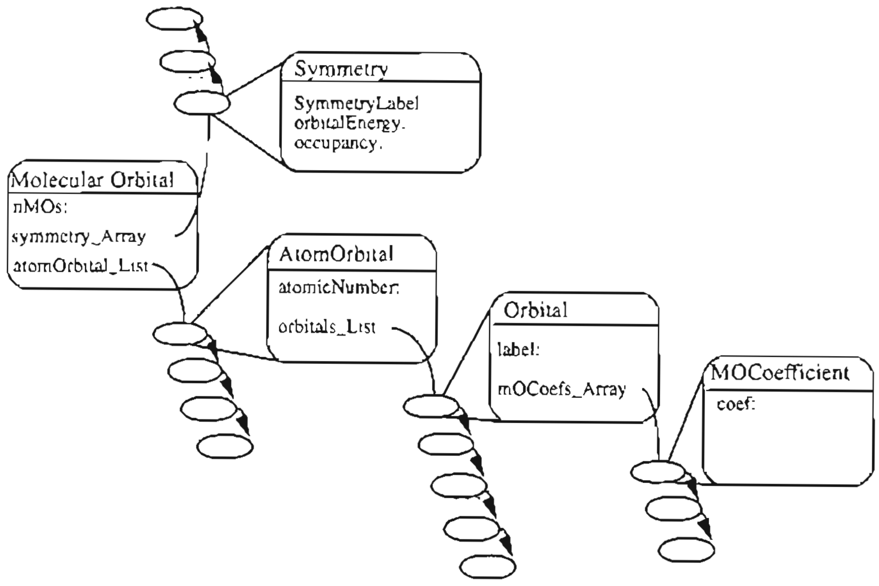


Figure 3.9: Molecular orbital.

shown in diamond-shaped boxes; derived relationships are shown as dotted lines.

3.3.2 Information Modeling Issues

During the course of the conceptual and information modeling stages, we encountered modeling problems of two kinds: complexities in the problem domain that would require more extensive domain modeling than we were prepared to offer and inadequacies in modeling methodologies.

Domain Complexities

The major domain-level modeling issue concerned the many semantically overloaded terms we encountered. These terms can usually be disambiguated within context by a domain specialist, but their meaning is not always clear when taken out of context or even within context to a casual user. "Molecule" and "experiment" are prime examples of this overloading. Sometimes "molecule" means any collection of specific atoms. For example, "Retrieve all experiments on the water molecule" means retrieve all experiments for which the molecule entity has two hydrogen atoms and one oxygen atom. "Molecule" can also mean a particular spatial arrangement of atoms associated with the input of an experiment.

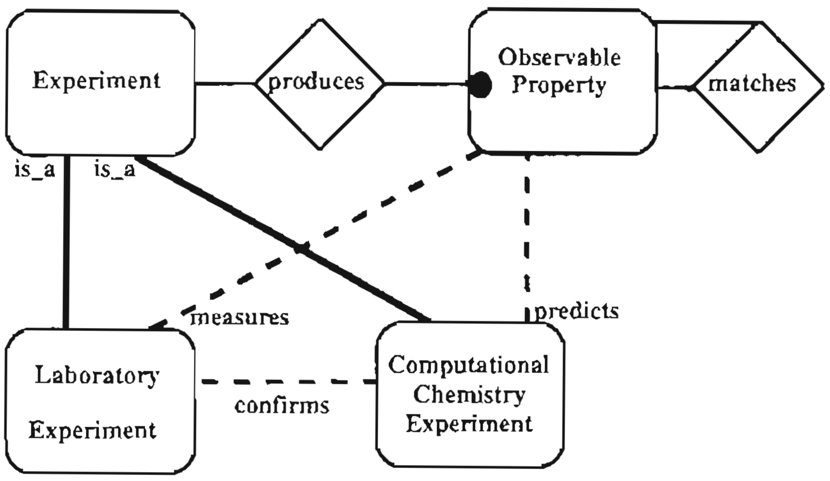


Figure 3.10: Functional relationships between experiment and property.

Sometimes a "molecule" includes molecular orbitals and sometimes not.

"Experiment" sometimes means one run of an application, sometimes a set of runs modeling the states of one molecule, sometimes a number of runs modeling transition from one or more molecules to one or more different molecules. "Experiment" is also used as an aggregation of any of these, in the sense that the chemist groups a number of runs together when doing an experiment focused on solving a molecular structure. We used the terms "experiment", "suite of experiments", and "investigation" to denote those different senses of the term "experiment," but there is still considerable subtlety remaining in how chemists themselves use the word.

While the decisions we made to disambiguate these terms were adequate for our purposes, their usage would need to be further resolved for an an industrial-strength database.

Modeling Methodology Inadequacies

Inadequacies in modeling methodologies for representing the information model were twofold:

1. The difficulty of representing groupings within a domain. A single instance of *molecule*, *basis set*, or *computational experiment* can be grouped into one or more

collections. For example, a molecule can belong to none, one, or several different families of molecules depending on how many molecular templates it matches.

While groupings that involve aggregations of entities such as *suite of experiment* were easily modeled, we could not represent levels of membership within a collection. Thus, for example, we could not easily show that some molecules in *family of molecule* were "more" a member of that family than others.

2. The low degree to which entity-relationship modeling can be integrated with behavioral aspects. We chose to avoid including behavioral aspects in the information model itself and worked from the conceptual model to code behavior at the logical (schema) level.

Some database researchers, particularly those working in scientific databases, have identified representing the degree to which a particular object belongs to a collection [9, 30, 31, 88, 94] as a research issue. Their results are still preliminary.

The lack of integration between entity and behavioral modeling is also a research issue in semantic data modeling [46, 81]. In fact, we had good success using an object-oriented approach to integrating behavior with class definitions and in implementing the required groupings for our domain, even though we found it difficult to represent these integrations at the information-modeling level. The capacity to deal explicitly with behavioral aspects is a particular strength of the object-oriented data models in comparison with current semantic data models, which we feel do not adequately integrate behavioral aspects with the entity-relationship based modeling.

The object-oriented framework also gave us several mechanisms for coping with detail and complexity in our conceptual model, albeit at the logical and physical design levels. First, we could group the attributes of an entity that were of interest into a single object. Second, we could model behavioral interfaces directly, thus capturing object interactions rather than providing indirect specification through separate data flow models or procedure code. Third, we could express the commonalities of similar entity types because object models support an abstraction or generalization hierarchy of classes.

Object models also support several extensibility mechanisms, useful in customizing or refining a domain model to a particular application. Those mechanisms include the creation of new subtypes in the generalization hierarchy, the use of existing object types in the definition of new types, and the ability to specify new operations for a type. Because such extension mechanisms are additive, applications that depend on the original model need not be modified if they do not use the extensions. For example, to incorporate a molecular editor into our framework, we might create subtypes of molecular structure and atom that include information about the graphical display of atoms and bonds between atoms (such as color, size, rendering style). Tools developed for the original definition of molecular structure and atom would continue to work on the new subtypes, as those subtypes would simply extend the existing set of behaviors.

3.4 Logical and Physical Design

This section describes the logical and physical design of the computational chemistry database. Because the choice of a data model determines the form of the logical design, we briefly review our database requirements in Section 3.4.1 and summarize our reasons for choosing an object-oriented system rather than a relational one for implementing the database. Then, in Section 3.4.2, we show the logical design of the database and give its persistent roots. An outline of the physical design of the database follows in Section 3.4.3. Section 3.4.4 relates our experience implementing a prototype database and how this led us to appreciate the inherent difficulty of loading our database and the importance of providing a closer interface between the applications and the database than we had first envisioned. We conclude the section and the chapter in Section 3.4.5 with our realization that database services alone would not solve the crucial problems of program interoperability that face computational chemists. Building a database for this domain requires an experiment management infrastructure that provides both data *and* computation services.

3.4.1 Choice of Target Database Management System

Classic software engineering principles recommend that a logical design be fully independent of the physical implementation [17].

In the case of database management systems, however, choosing between implementing the system as relational or object-oriented can affect the manner in which the logical design is described. The logical design of a database application is not altogether independent of its physical implementation because the logical design is usually couched in data modeling terms, and given as a database schema. The logical design of a database that will be implemented in a relational database management system consists of a logical description of the database couched in the language of the relational model, i.e., normalized relational tables with indications of primary and foreign keys. Because the relational model is programming-language independent, the logical database design is largely independent of the particular relational database management system and target programming language in which the system is implemented. On the other hand, the logical design of an object-oriented database consists of the classes and method signatures for the database; because there is no object-oriented data model *per se* [116], the classes and methods are usually couched in the target programming language or database manipulation language. Thus, before describing the logical model for the Computational Chemistry Database, we recount why we chose to implement the system in an object-oriented database.

An object-oriented database (OODBMS) combines object-oriented language and modeling features (encapsulation, object-identity, subtype and implementation hierarchies, direct representation of complex structures) with data management capabilities. An obvious advantage is that we can map conceptual-level classes, operations and hierarchies directly into counterparts in the database's data definition language (DDL). While some encoding is required (e.g., for taxonomies), class hierarchies and binary relationships (even many-to-many binary relationships) usually require no encoding.

On the implementation side, an OODBMS usually provides a data manipulation language (DML) that allows one to implement many operations without recourse to an external application language. This behavioral capability is also useful for building routines into the system that convert to and from particular formats. Many OODBMSs can run in a distributed fashion on a network of computers of possibly different kinds. OODBMSs thus provide some location transparency and help mask the heterogeneity of computers and operating systems. Finally, some OODBMSs provide gateways to the operating system file system and relational databases, and hence can serve as an integration mechanism for multiple data sources [104, 120, 132, 163].

While an OODBMS certainly makes for a simple mapping from information model to logical model, it is not an absolute requirement. CAChe [23], for example, currently supports an object-oriented information model using a relational-like tabular representation. Our survey of scientific database research showed six alternatives for physical implementation: special-purpose data management facilities [16], extensible tool kits [26], logic databases [139], relational [32, 33] and extended relational systems [76, 90, 156], and OODBMSs [196].

For our purposes, developing a special-purpose data management facility involved too much overhead and would not meet the need to scale up easily to support additional applications or experiment types. We felt that extended relational systems and extensible tool kits are still in the research phases, and logic databases do not in general offer the database capabilities we required. Thus, relational and object-oriented systems were our two viable alternatives.

Relational database systems offer some distinct advantages over object-oriented database systems: they are well-understood, well-grounded in theory, well-documented, and robust. Unfortunately current record-oriented database technology does not support scientific applications well. Relational systems are inadequate for representing spatial information (such as we require for molecules, basis sets and molecular orbitals), for maintaining meta-information as relations, and for adequately representing the inherent structural complexity of most scientific data [47, 60, 61, 120].

Working from our conceptual model and analysis of the computational chemistry database needs, we made the following observations:

1. The schema abounds with highly interrelated data that would require many tables

(and thus joins)[4] in order to bring one experiment into memory. Since an experiment is the unit of work per interactive session, we believed that the number of joins required to set up an experiment would have an adverse effect on performance. While the relational model is flexible enough to implement such structures, albeit indirectly, we felt that representing logical structure as directly as possible in our physical implementation was important for efficiency in both implementation and execution.

2. Normalizing the conceptual data model into relational tables would require breaking down complex, but scientifically meaningful, structures such as molecule and experiment. We believed that the current relational model lacks the representational power for an intuitive rendering of the structures required. Computational chemists are highly skilled programmers, and many write their own programs. Giving expert users such as the authors of computational chemistry applications an intuitive understanding of the physical model is important because they will want to program their own extensions to the system.

3. The application area abounds with the need for writing conversion and display methods, from properties of one type or unit to another, from program parameters from one application format to another, and so forth. Object-oriented systems allow for storing methods with data descriptions, a feature we thought particularly helpful in keeping track of the many small conversion programs.

4. This database application requires collecting a number of entities into sets and lists: collecting experiments into personal collections, suites and investigations, and collecting molecules into families. Object-oriented database management systems provide good built-in collection facilities.

5. Finally, our intuition was that object-oriented database systems would serve this and other scientific applications well, and we wanted to gain experience with this new technology.

[4]Even if molecule and molecular orbital entities were not normalized, we estimated about 20 joins per experiment. If the model were fully normalized, several hundred joins might be required.

From the observations above, we concluded that object-oriented database management systems would be more appropriate for this application than relational systems. We identified a subset of attributes and methods for each major database entity and conducted a feasibility study to determine which object-oriented database systems would be adequate to the purpose [42]. This study led to our choice of ObjectStore, a commercial object-oriented database management system designed for distributed client-server database applications written in C++.

3.4.2 The Logical Design — an Object-Oriented Database Schema

In this section, we briefly describe the logical model for the database, i.e., the computational chemistry database schema.

As noted above, translating an information model into the logical model using the relational data model consists (generally) of casting entities into relational tables, and assigning prmiary keys. To assure that a relationship between entity A and entity B can be effected by relational joins, one assures that (for example) the primary key of entity A is among the attributes of entity B. Once candidate relations and keys have been identified, one assures that the database is normalized by applying well understood tests.

Translating an information model into a logical model for an object-oriented database is initially simpler, since each entity in an entity-relationship diagram is an object-oriented class. The data model in which one represents the schema is the programming language class structure in which the database is to be implemented. The ObjectStore data model consists of C++ classes plus ObjectStore extensions to C++ that allow the specification of binary relationships and collection classes.

Each entity in our information model became a C++ class; each relationship an ObjectStore relationship. Casting behavior into the logical model was more complicated because (as we pointed out in Section 3.3.2) we did not include behavior in our information model. To include behavior in our logical model, we had to go back to the less formal conceptual model and determine which behaviors we would model as C++ methods.

The final stage of our logical modeling was to determine the persistent names for database objects. Persistent names in the logical model correspond to the named entry points we defined in the information model. For example, the persistent name Chemist::extent refers to a set of name and object-identifier pairs for all instances of *chemist*. To find a chemist named "Feller", one traverses this set of all *chemists*, examining the name of each *chemist* and then using the associated object-identifier to navigate to the chemist of interest. An alternate (though less desirable) way of providing this same functionality would be to provide a root variable for each *chemist*; thus, for example, we would directly access the *chemist* instance for Feller via a root variable *Feller*.

The logical model depicted in Figure 3.11 presents our choices for the persistent names in the database. Access to experiments is usually effected through two persistent root variables, one set of all chemists and one set of all molecules. Basis sets and applications are similarly independently accessible. In our database, it happens that each root variable corresponds to a set of instances of a class. Thus, in the figure, each root variable is underlined and an arrow is drawn to the class whose object-identifiers its corresponding set contains. As before, the superclass-subclass relationship is represented as a thick line. One-way relationships, where one can navigate from an object in *Class A* to its corresponding object in *Class B* (but not vice versa), are named at the *Class A* side of the relationship line. Where we need to navigate in both directions the relationship is named at both ends of the line. We made the simplifying assumption that a laboratory experiment can confirm a computational experiment, but not vice versa, and that this relationship is explicitly a relationship between experiments (not properties produced by experiments).

Since a full implementation of the conceptual model was not required for our research prototype, we made a number of simplifications. The *Molecule* entity provides its structure only as Cartesian coordinates. We did not use the superclass *Apparatus* for *Laboratory Instrument* and *Application* entities, nor were the *Suite of Experiment*, *Investigation*, and *Family of Molecule* aggregations implemented as classes. Furthermore, we made no distinction between private and public experiments.

We mapped the computational chemistry information model described above into an ObjectStore schema by using the ObjectStore Schema Designer, and found a very good correlation between our requirements and the modeling power of the ObjectStore Data

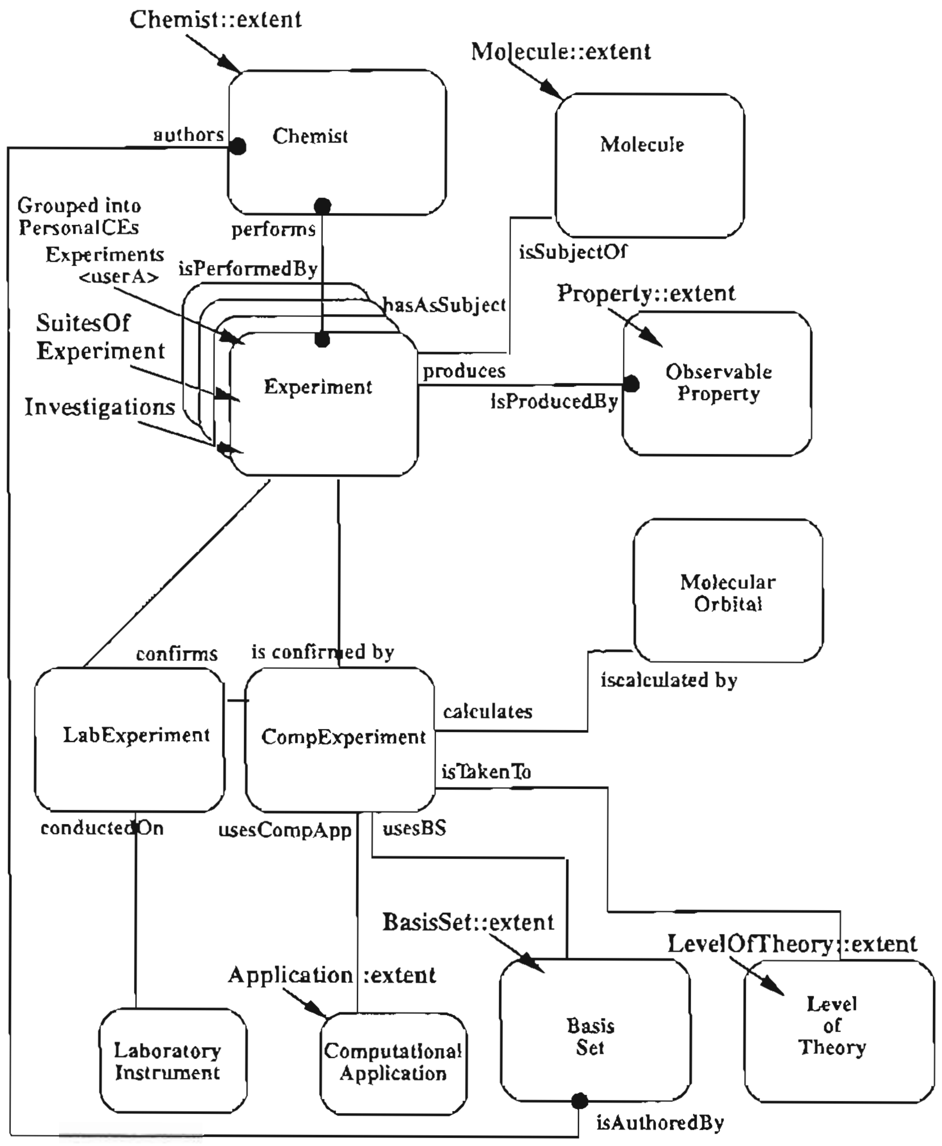


Figure 3.11: Persistent names for the CCDB logical model.

Definition Language. Binary relationships, subtypes, and complex structures for molecule, basis set and molecular orbital mapped directly to ObjectStore. ObjectStore supports binary relationships of cardinality greater than one by generating an additional C++ class to represent the relationship. This intermediate class is transparent to the programmer. Taxonomies for *application*, however, did not map directly; one reason for this is that a ternary relationship among application version, compiler version, and computer platform was required. For the sake of convenience in our prototype, we simplified the taxonomy by representing the application version and compiler as attributes. To implement the taxonomy fully, we would have had to generate by hand an intermediate class for each ternary relationship. (This limitation is discussed further in Section 5.3.2, page 170.)

Our simplification of the application taxonomy seems justified because there are relatively few versions of applications and compilers, and because the number of levels in the taxonomy is fixed, rather than arbitrary.

3.4.3 Computational Chemistry Database — Physical Design

The physical design of a database describes how the data are actually stored in the database, and describes complex low-level data structures in detail [101]. Many modern database management systems map the logical design (i.e., the schema) directly into physical data structures, and the physical design tasks for the developer consist primarily of index selection and directives or pragmas (i.e., hints) to the system that will improve performance. In addition, database designers may specify and implement additional programs to enforce integrity constraints that are not adequately supported by the database management system. As we had expected, given our decision to implement the database in an object-oriented database management system, the physical design of the database was straightforward. ObjectStore automatically enforced integrity constraints for inverse relationships, and we chose not to specify physical optimizations such as clustering chemist and experiment classes or providing information about how to physically store particular collections (e.g., as arrays or lists).

The physical design for our system included the following:

1. Implementing root variables as ObjectStore database roots. Root variables for

Chemists, Molecules, BasisSets, and Applications are the extents of their respective classes, and are each implemented as a collection containing all members of the class. Insert and remove statements for the class extents were explicitly added to constructor and destructor functions of the element classes.

2. Declaring attributes indexable, as appropriate. Where an ObjectStore query specifies a particular path, each element in the path must be declared indexable in the schema; ObjectStore maintains an access method for each data member mentioned in a path. Thus, for example, we declared chemist name indexable because we wished to retrieve chemists by name, in alphabetic order. Other indexable attributes included application name and basis set name.

3. Designing rank functions to control ordering of instances in a class. ObjectStore provides a facility for defining an iteration order where a navigation path ends in the instances of a user-declared class: one need only supply a function called *rank*, whose possible values for any pair of instances of the class are enumerable. For example, since we wished to retrieve experiments for any given chemist in reverse chronological order, we wrote a ranking function so that the order of experiments could be determined, given experiment date and experiment time.

4. Designing class methods. For each class, we wrote constructor and destructor methods, as well as terse (one-line) and verbose (full object) displays. In addition, because the definition of identity differs from class to class, and because having unique instances of these classes mattered, we wrote class identity methods for classes where we had defined an extent (collection of all instances).

3.4.4 Loading the Database

To determine the feasibility of building a database of past experiments, we built a prototype database of computational chemistry experiments, along with a rudimentary browser for examining the data. In the course of this experience, loading the prototype database with input and output files from previous experiments impelled us to make the database a more integral part of the user's interaction with the application.

The prototype implementation consisted of three major activities: building the schema and creating the database, loading the data from 20 experiments into the database, and implementing queries on the database. Of these tasks, loading experimental data was by far the most time-consuming, even though we were working from experiment data that had already been extracted from the application's output file.

We illustrate here the complexity of this task by showing the steps required for loading a simplified "experiment" consisting of experimental data (date, energy) and a performing chemist. To load this experiment we must load instances of two classes: experiment and chemist. Assume in this case that the chemist who performed the experiment is already in the database and that we do not wish to include duplicate instances of *chemist* in the database. Note that ObjectStore distinguishes between temporary and persistent instances of classes; a persistent instance is one that is stored in the database. Unlike relational systems, which are value-based, ObjectStore does not preclude instances with identical attribute values even of persistent classes. Identity in ObjectStore is not value-based; two references to Objectstore class instances refer to the same instance if those two references, i.e., object identifiers, are equal. Whether or not two instances of an Object-Store class are value-equal is a application-level question.

To load the experiment instance in our example, we first read the experiment data and created the persistent instance of *experiment*, leaving reference to *chemist* null. We then read the chemist data, and created a temporary instance of *chemist*; we examined all instances of *chemist* already in the database to see if the chemist referred to in the new experiment was already in the database. If we found that chemist in the database, we placed the object identifier for the new experiment in that chemists' "performs" collection, then went back to the *experiment* instance and stored the object identifier for the *chemist* instance in the *experiment*'s "is performed by" collection. The intermediate collections for the **performs** and **is performed by** relationships implement one-to-many and many-to-one experiments. If we did not have a duplicate chemist, then we made the temporary instance of *chemist* persistent, and (as above) placed its object identifier into the *experiment* instance. What made the loading process so complicated was that we needed the object identifier for the *chemist* instance to complete the *experiment* instance (and vice versa). Furthermore, to determine if we needed to create a new persistent instance of *chemist*, we had to carry with the new experiment data enough information about the chemist to check value identity with existing instances of chemist, and to create a new instance if needed.

For every instance of *experiment* to be loaded, there were eight classes for which "existence" tests like the above were required. These tests made our database loader a complex program. In addition, the loader required that all the necessary disambiguating attributes be included with the data.

A second and more critical problem with a *post facto* database loader is that the loading of experimental data into the database is not an integral part of the running of the experiment, but a separate task for the user. Having the loading as a separate task adds to the chemist's overhead in running experiments, can be inadvertently forgotten, and inevitably would prove error-prone. In addition, having the database separate from the applications did little to simplify the chemists' computing environment.

3.4.5 Summary: Data Services Alone are Inadequate

Experience loading our prototype database led us to conclude that capturing experimental parameters and results must be an integral part of setting up and running experiments. The best way to accomplish this objective, we felt, is to have input parameters originate in the database and be conveyed to the program, rather than originate externally and be transferred into the database after the experiment has run. Recognizing object equivalences after the fact seemed quite hard to automate, while creating inputs from the database allowed simpler equivalence checks. Thus, for example, references to the *chemist* performing the experiment could be directly inferred from the system, since a chemist logs in to the system and establishes his or her identity. Similarly, when loading experimental results we would know exactly which basis set was used because we would have generated the input to that experiment (which included the basis set).

Chapter 4 treats in detail the problem of connecting computational applications to our database, and explores the idea of modeling both applications and invocations of applications as objects — as computational proxy objects. Because the proxy class is part of the database, we think of it as extending the database to provide computation services.

Chapter 4

Providing Computation Services: Proxies

In Chapter 3, we showed how a database of past experiments could help computational chemists make appropriate choices for input parameters to new experiments. We then described the database design and recounted our experience building a database prototype. Problems encountered in loading data into the prototype prompted us to make the capture of experimental parameters and results an integral part of setting up and running computational experiments — generating program inputs from the database and capturing results automatically upon experiment completion. That decided, we were then faced with the problem of how to provide the requisite interface between computational applications and the database. The database-to-application interface must help the user generate program inputs and capture program outputs, assure experimental reproducibility by linking program inputs to outputs, and provide experimental comparability by representing data from different applications in compatible formats. The resulting system should also simplify the chemists' complex computing environment.

We considered and eliminated two traditional alternatives for interfacing applications with the database. Modifying the applications to read from and write to the database is impractical, and naively encapsulating the applications as database objects does not provide adequate flexibility for controlling the running applications. We postulated that structuring additional information and behavior about computational applications and their invocations as objects would provide the needed power and flexibility. So we decided to model programs and computations as complex objects using a structure of our own design dubbed "computational proxy". The idea of modeling programs and computations as objects arose directly from the following considerations:

- The need to draw inputs directly from the database rather than insert them after the fact into the database,
- The realization that a database object representing the input to an application could also represent the invocation and control of that application, and
- The need to capture outputs from different applications in comparable formats to assure experiment reproducibility and comparability.

This thesis offers an improvement over naive encapsulation as a method for interfacing existing applications to databases, and this chapter constitutes the heart of that work — the definition of the computational proxy. Design issues and alternatives are described in Section 4.1, and Section 4.2 defines the computational proxy mechanism and lays out its functional requirements. Section 4.3 provides design detail, including extensions to the conceptual model made to accommodate the proxy mechanism. Section 4.4 describes the infrastructure architecture. There we distinguish "database"(i.e., proxy) responsibilities from system and user responsibilities by specifying system-level services required to support the proxy mechanism and outlining the functions to be provided by a graphical user interface to the computational chemistry applications.

4.1 Design Issues and Alternatives

This section describes design issues for the interface between computational applications and our database. We first recount our initial design decision not to modify the applications to read inputs from and write outputs to the database. Section 4.1.1 argues that neither this alternative nor a naive encapsulation of the application programs is appropriate, and proposes an extension of encapsulation that provides ways to describe and access applications and invocations of applications. Once our major design strategy was set, we needed to determine how to describe applications to the database; challenges identified in describing applications declaratively are laid out in Section 4.1.2.

4.1.1 Strategy for Connecting the Applications to the Database

The need to avoid *post facto* database loading and to make loading experimental data an integral part of running experiments led us to consider the design alternatives described here. An obvious way to meet our objectives was to interface existing computational chemistry programs directly to the database by modifying the programs to access the database for inputs and to write outputs directly into the database. We did not consider this approach viable because the computational chemistry programs in question are complex, very large (100,000 to 300,000 lines of code[1]), and revised at least yearly. Creating and maintaining a direct database interface to these programs would involve, first, finding and replacing all read accesses to inputs and all write statements for outputs in 5 or 6 very large programs, and then repeating this exercise whenever new versions of the applications were released. Such a programming and maintenance task is not feasible.

A second obvious alternative was naive encapsulation. In this case, each application to be interfaced to the database is "wrapped" with a wrapper object that receives and processes requests to run the application. The application would be invoked when a message with the input data as an argument was sent to its wrapper object. A more integrated approach than naive encapsulation was to leave the application program untouched, and encapsulate it as a database object or as a message of a database class. Under this approach, an application would run when its corresponding message was sent to the object representing its input. Either encapsulation approach assumes some way of calling out to the application or linking it in to the database's executable file, as well as constructing conversion routines between data objects and files. While "wrapping" a legacy application may work in some cases, we believe it is too limited an approach in general.

Encapsulating computational chemistry applications, capturing an external application program as a single object or message as above, assumes a very simple input-output model, one where all the inputs are passed in as a unit, the program executes and then outputs are sent out as a unit. Such a simple model is inconvenient and inappropriate for computational chemistry applications for the following reasons. First, we need to

[1]MELDF is considered a "small" package at over 100,000 lines. Gaussian 92 is over 300,000 lines.

separate supplying inputs from scheduling the execution of the program. Second, because the computations are long, the chemist would not want to wait while it executes. We wanted the program to run asynchronously from the database session, freeing the chemist to do other things with the database while the application executes. Thirdly, instead of treating the program execution as an atomic action, we wanted to monitor its progress in order to stop it, checkpoint it or observe intermediate results. Finally, a single object or message gives little help for organizing and gathering inputs or for structuring the translation process.

We eschewed the two design alternatives above. (1) Directly interfacing the applications to the database would provide an easy way to provide input parameters from the database and to capture results into the database, but such modifications would be impractical to maintain in the face of continuing releases of these application programs. (2) Encapsulating the applications as database objects did not provide any ability to monitor and control ongoing experiments. We thus explored the idea of modeling both applications and invocations of applications as objects, calling the structure that modeled computations as objects a "computational proxy" and hypothesizing that the proxy structure and mechanism would provide an interface of computational applications to the database without having to immediately modify the programs themselves. We furthermore believed that the proxy structure could provide the chemist with help setting up and monitoring experiments.

4.1.2 Challenges in Describing Applications Declaratively

Having made our fundamental design decision about connecting applications to the database, we were left with the question of how to structure and engineer the computational proxy. A design goal for the computational proxy was that adding an application or installing an updated version require neither changes to the database schema nor extensive application-specific programming. Because changing the database schema requires database design skill as well as technical knowledge of a specific database management system, we wanted to represent application inputs and outputs declaratively, within the database. Representing this information declaratively, and providing database methods for its interpretation, minimizes application-specific programming. Application-specific programming is minimized because operational differences in running different applications (where automated at all) are often handled by special-purpose programs (which are expensive to write); we reasoned that if the proxy could read declarative information about applications and generate the desired application-specific action, less application-specific code would be needed.

This section describes the challenges in describing the interface to computational applications declaratively. Writing a special-purpose program to interface one version of one application to the database is relatively straightforward. Indeed, we wrote a special-purpose program to do so in an initial feasibility study (described in Section 5.2.1). However, writing and maintaining one special-purpose interface per application quickly becomes a daunting programming effort when faced with yearly releases of several applications. In addition, writing any new program in the context of an object-oriented database system often means creating new classes. Creating new classes violates our design goal of minimizing database schema changes, and a proliferation of classes would prevent us from preserving a common and stable conceptual model that facilitates the user's understanding of the database. Because of these issues, we were committed to finding ways for application registrars to describe applications in a declarative, rather than procedural or programmatic, manner.

While writing descriptions of the applications that one wishes to interface to the database is easier for the non-computer-scientist than writing code and adding classes to the database, providing the mechanisms to interpret those descriptions is far from trivial. Interpreting those descriptions and carrying out the actions so defined requires writing, in essence, an interface to the database general enough to cover the major applications within a specific domain. Writing such an interface is considerably more complex than writing one or even several database interfaces for specific applications. The research issue involved here is whether it is possible to design and implement database objects that can interpret user-defined descriptions of applications and use them, in conjunction with other information in the database, to drive program input, invocation and data capture. Our work has involved designing structures to describe applications and implementing tools that interpret those structures.

For our application descriptor mechanisms to work requires, first, that a single conceptual data model cover the semantic domain so that users can maintain application interface information without making schema changes. Second, this model must be implemented into a particular database system. Third, inputs and outputs of application packages must be *conformable* to the conceptual model. By *conformable* we mean that the conceptual model subsumes the types implicit in application inputs and outputs. Being conformable further implies that database programs can translate database objects to and from application inputs and outputs.

We summarize our research problem and approach as follows: Creating classes and methods is difficult, expensive and error-prone; creating instances of existing classes is much easier. But, can we set up a system so that applications can be attached to the database simply by creating new object instances? One way to accomplish this task is to produce new database objects that describes application input and output files along with their relation to corresponding database objects, and to provide a general interpretation mechanism for these descriptive objects. We can then design a language that a chemist can use to create instances of the descriptive objects. Given a way of defining application inputs and outputs, we can write additional mechanisms for controlling application invocation.

Our approach raises the following challenges:

1. Providing an interpreter for application descriptors that is application-independent. To do this, we must make explicit the division between the part of the proxy that is application-independent and the part that is application-specific (and hence will require customization on a per-application basis).

2. Designing a language so that a chemist can build a model of application inputs and outputs. To meet this challenge, both the conceptual model that describes the semantic content of application inputs and outputs and the language that describes the syntax of those inputs and outputs must be sufficiently general to cover a range of applications in the domain. If the database conceptual model is adequately general,

then application inputs and outputs are simply *syntactic* variants of existing entities in that model.

3. Clearly dividing responsibility for controlling application invocation between the proxy and the network services. The proxy can gather domain-specific information that might affect network-level (policy) decisions, but the network services should provide the mechanism for accessing processors. Thus, for example, the proxy gathers the program inputs, and the network services physically send inputs to the chosen processor along with a message (built by the proxy) to start the application.

In spite of the high degree of semantic and syntactic complexity in the application programs, we believe we can meet the first two challenges because the application programs are based upon common scientific principles, principles that evolve at a relatively slow rate. Thus, our conceptual domain model (described in Chapter 3) describes scientific principles rather than several particular applications. We have approached the challenge of describing syntactic variants of the domain model by designing an input file generation language (in effect a scientific database report writer) and an output file (text) parsing language. These languages cover the major experiment run types in common use, and are described in Section 4.3.

Our efforts to meet the third challenge, described in Section 4.4, have taken into account current distributed operating systems research that takes as precept a client-server (distributed) computing environment clearly separating policy and mechanism [13].

4.2 Computational Proxy: Definition and Functional Specification

Recalling the user scenario for ab initio applications from Section 3.1, we note that the database of past runs helped a chemist to choose input parameters, but not to actually perform the experiment or to place experiment results into the database. The proxy mechanism addresses these needs. With it, the user can start up and control computational processes, as well as capture important information about a given computational experiment. When the user schedules a run, a proxy uses a description of the given application

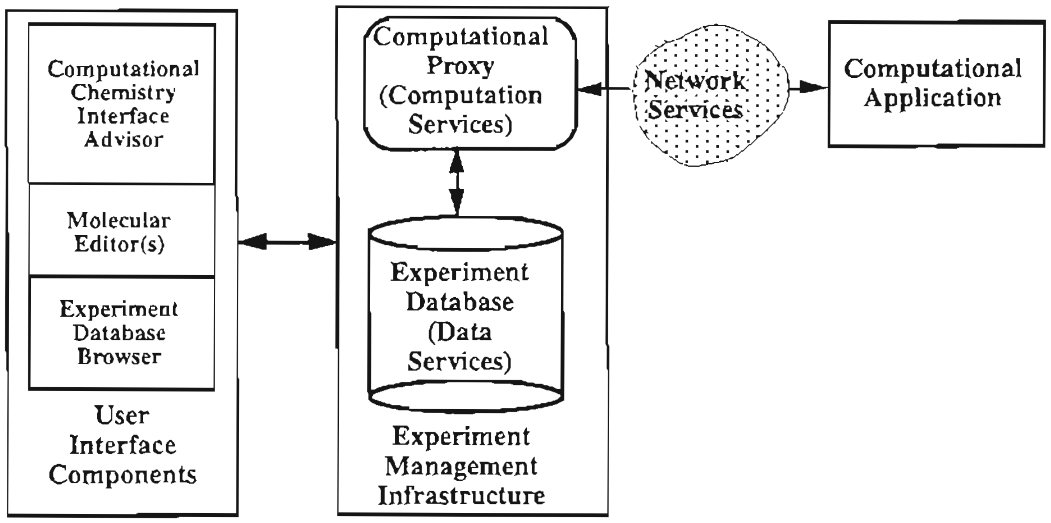

Computational Chemistry Interface Advisor
Molecular Editor(s)
Experiment Database Browser
User Interface Components
Computational Proxy (Computation Services)
Experiment Database (Data Services)
Experiment Management Infrastructure
Network Services
Computational Application

Figure 4.1: A data-centered framework for computational science.

to automatically transform experimental attributes held in the database into appropriate inputs formatted for that application. Once the run has terminated, the proxy again uses the application descriptor, this time to parse the outputs and place experimental results into the database in a common form.

We call our infrastructure "data-centered" because the proxy itself is a database object, because the proxy relies on the database for information about applications, because the proxy takes experiment input from and places results into the database, and because the proxy uses the database to maintain a persistent representation of an ongoing experiment. Figure 4.1 illustrates the role we envision for computational proxies: the point of contact between the user interface and computational applications on the one hand, and between network services and computational applications on the other. The central box in Figure 4.1 encompasses the proxy plus database. The User Interface, shown in the left box, includes three components: the Computational Chemistry Interface Advisor (a user interface to computational applications under development by Dr. David Feller and others at Pacific Northwest Laboratories), one or more molecular editors, and an experiment database browser. The Network Services component provides distributed system services such as transferring files and starting up and stopping processes.

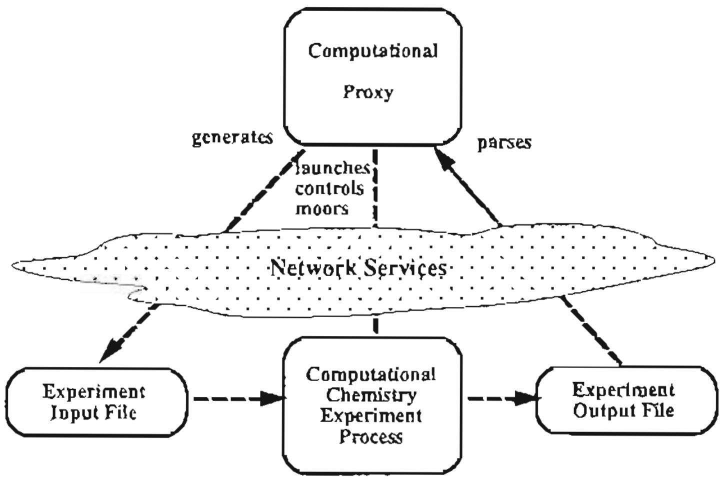


Figure 4.2: Computational proxy functions for managing ongoing experiments.

The key component of our experiment management infrastructure is the computational proxy object. An instance of proxy "stands-in", within the database, for every computational experiment in preparation, currently in process, or recently completed. Methods associated with the proxy class provide an interface to the computational programs that run those experiments. Implemented as persistent, object-oriented classes with data and methods, the proxy mechanism encapsulates syntactic differences among semantically related applications.

Figure 4.2, a simplified view of the proxy and an experiment process, illustrates a functional view of the proxy. The proxy generates an experiment input file. Control of the computational chemistry experiment process consists of launching the process, controlling it and registering the fact that the process has terminated ("mooring" the experiment). Corresponding to each active experiment is a proxy object; the proxy object maintains a persistent record of the experiment that can be viewed by the chemist. Once the experiment has completed, the proxy transforms results into a format common to all applications, and loads them into the database. The *generate*, *launch*, *control*, *moor* and *parse* functions may be invoked from a remote machine, using appropriate network services.

Figure 4.3 shows this conceptual encapsulation of computational chemistry applications. Here, we see two separate experiments on the water molecule, one using the

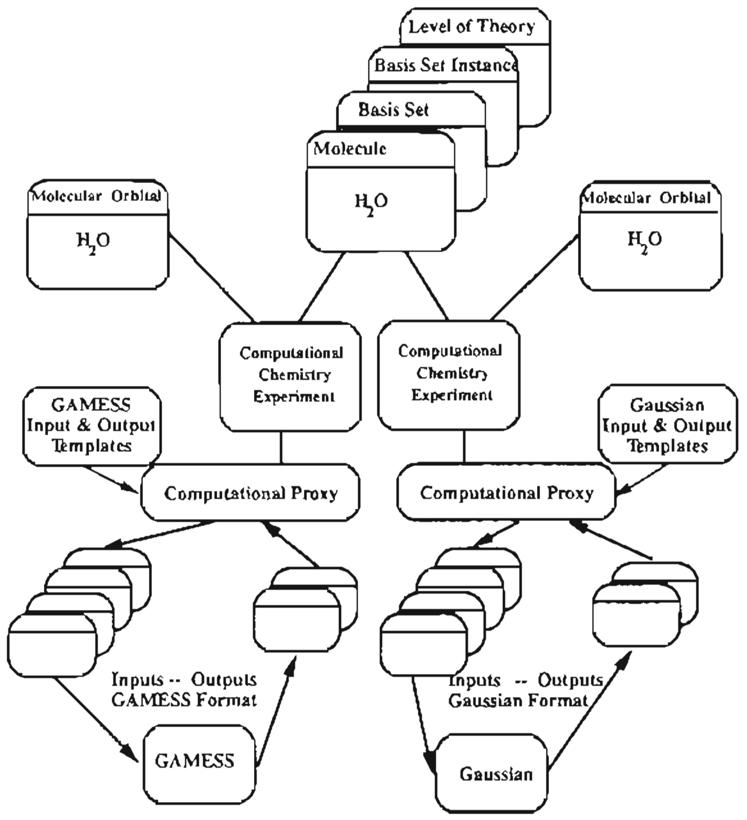

Level of Theory
Basis Set Instance
Basis Set
Molecule
H_2O
Molecular Orbital
H_2O
Molecular Orbital
H_2O
Computational Chemistry Experiment
Computational Chemistry Experiment
GAMESS Input & Output Templates
Gaussian Input & Output Templates
Computational Proxy
Computational Proxy
Inputs -- Outputs GAMESS Format
Inputs -- Outputs Gaussian Format
GAMESS
Gaussian

Figure 4.3: Hiding syntactic complexity of computational codes.

GAMESS application and the other the Gaussian application. Inputs to the experiment are presented in the boxes directly above the experiments themselves, namely molecule, basis set, basis set instance and level of theory; each comes from the database and is translated by a computational proxy (using an input template) into specific formats required by the respective application. The proxy uses output templates to translate application outputs into a common format before placing them in the database. In this case, experiment results of interest consist of two molecular orbital objects, one associated with each experiment.

An instance of a proxy object is created as soon as a chemist begins building an experiment, and holds experiment information that is of an ephemeral nature, i.e., relevant only to the running of that experiment at a particular point in time, on a particular machine. The instantiation of a computational proxy extends from the point when a chemist begins to conduct an experiment to that point in time when the chemist declares that the experiment has successfully completed and output data have been retrieved. A proxy representing an unsuccessful experiment is retained until that experiment is updated and resubmitted, or until the chemist archives or abandons the investigation containing that experiment. Thus, the computational proxy object associated with a given experiment will ultimately be removed from the database or archived. While an experiment is active, the proxy holds information idiosyncratic to the process actually running that experiment, such as the process ID, current resource utilization, and names of working files. The experiment object that represents a successful experiment, as opposed to the proxy object that represents that experiment's computation, endures within the database as the locus of scientific information about that experiment.

4.2.1 Computational Proxy: Conceptual Model

This section describes how our conceptual model for the computational chemistry database was extended to include the computational proxy. After an initial narrative illustrating how the proxy entity relates to other entities, we go on to list the attributes comprising the proxy entity. As in Chapter 3, entities (i.e., database objects) are *italicized*, relationships between entities are expressed in **boldface**, and attributes are <u>underlined</u>. The entity-relationship diagram in Figure 4.4 represents the information model for the proxy and illustrates the narrative description of the conceptual design below. The entities *computational application* and *computational experiment* have already been described in the CCDB conceptual model. (See Figure 3.6.)

A *computational proxy* **represents** a *computational experiment* and **controls** the *computational process* that corresponds to that experiment. A computational experiment **is conducted** as a computational process by the proxy. A computational process **runs on** a *particular computer*, which is connected to the same network service to which the proxy itself is connected. Any particular computer **is an instance of** some generic *computer platform*, e.g., the processor "coho" is an instance of a "Sun4" computer platform.

Any running computational experiment **uses** a *computational application*. In order for an experiment to run on a particular computer, the application it uses must be **available**

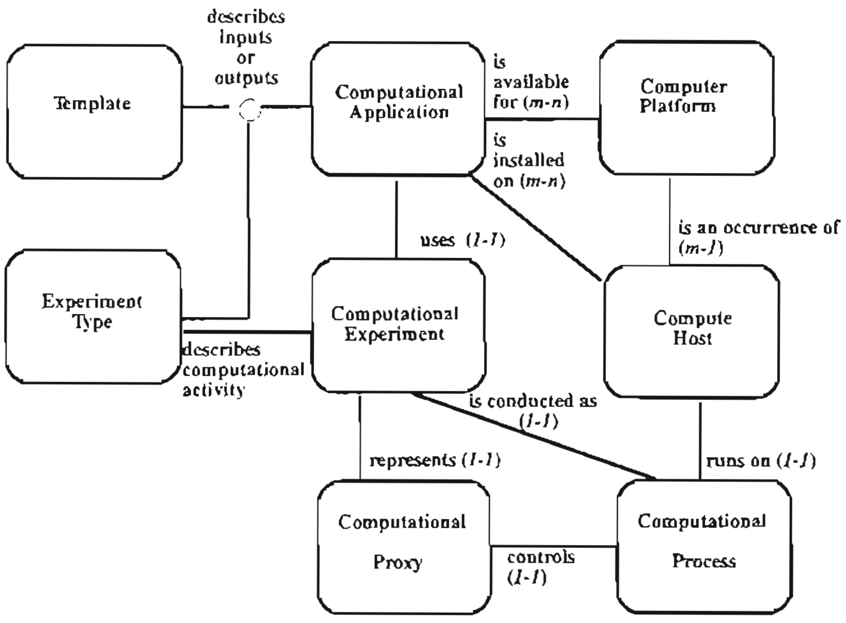


Figure 4.4: Information model for computational proxy.

for the corresponding computer platform and installed on a particular computer of that platform type.

Templates describe the input and output file formats for a computational application. The *experiment type* describes the computational activity for an experiment. Experiment type is a classification of computational activity that cuts across applications; thus, for example, two applications might support the "optimize" computational activity.

Because parameter requirements differ according to the experiment type, experiment type within application serves as a grouping factor for *template* objects. Templates are discussed in Sections 4.3.3 and 4.3.4.

The proxy contains information about the application process that relates to the actual running of the experiment. Defining the *experiment* entity separately from the *proxy* entity allowed us to cleanly separate information of scientific value from information about the running per se of the experiment. Information of scientific value is stored with the experiment (or associated objects such as molecule), while information of operational value is stored with the proxy. Information of scientific value is maintained in the database as long as it has scientific value. Data of an operational nature, i.e., the proxy, is deleted as soon as it is no longer needed — when the computational experiment has completed. The primary clients of the *experiment* object are persons, (i.e., the scientists); clients of the *proxy* object are other database objects.

The proxy object holds the following information, although some attributes may be null for any particular proxy object:

1. Date and time the computational process was last monitored.

2. Cumulative resource usage, in terms of the particular computer on which it is being run: CPU time, disk space utilized, temporary disk space utilized, elapsed time. Resource usage in general terms, e.g., megaflops[2], is stored with the experiment since general measures of resource utilization are relatively independent of the particular computer on which the experiment was run.

3. Process identifier of the corresponding computational process.

4. Current status of the computational process, i.e., not-scheduled, scheduled, running, moored, aborted (by the system) or stopped (by the user).

5. Input file name(s).

6. Output file name(s).

The experiment object holds information sufficient to reproduce the experiment, and the proxy object contains information sufficient to locate and control the experiment's computational process.

In Section 3.2.4, we specified what information about computational applications and particular computers must be available in the database to identify the experimental apparatus on which computational experiments were performed. Adding computation services to the database means that additional information about computational applications and particular computers (which we call "compute hosts") must be made available to invoke and monitor computational processes. The following network-level information must be accessible to the computational proxy:

[2]Millions of floating point operations per second.

1. For each *particular computer* (*compute host*): full network address, subdirectories, and maximum resources available where each application's files must be placed.

2. For each *computer platform*: performance level, e.g., parallel, super, mid-range, micro.

3. For each *application*: invocation signature. By invocation signature, we mean the explicit calling sequence required to invoke the program.

The above information is available to users, the user interface, and the proxy when making decisions regarding the scheduling of experimental processes.

4.2.2 Database Support for the Proxy: Registering Applications

In Section 4.2.1 we described the information structure of the proxy itself. To support computational applications with the computational proxy, we need the ability to register this information about computational applications with the database.

Information about applications required for our needs falls into two categories: information for the scientists' direct use (already described in Section 3.2.4) and information needed by the proxy. Information in the latter category consists of descriptions of application parameters, inputs and outputs. One goal of the proxy design is to allow a computational chemist who is an expert user of an application to register that application with the database. We believe that scientists should be able to add or modify application interfaces for the system without having to understand the technical details of the database design, or having to consult programmers or database administrators when minor changes to an application occur. There is no substitute for having a domain scientist knowledgeable in an application define that application to the system, but the application registrar should not have to be a computer scientist or systems programmer to define his or her application to the database.

We use the term *application registrar* to distinguish the application-expert-user who registers applications from the application-user.[3] The application registrar needs to understand the computational chemistry database schema, but not the particular database management system in which the database has been implemented. To write templates, the registrar needs to understand in detail the experiment types supported by the application and the application formats for input and output, but only in general how the proxy generates input and parses output files. The registrar will also need to know which compute hosts can run the application, and details about those hosts sufficient to start up an application on each.

4.2.3 Computational Proxy: Behavior

In this section, we describe the behavior of the *computational proxy* entity. Because we are describing an entity of our own creation, rather than capturing an existing process, the format we use for describing the conceptual model varies from that used for conceptual models elsewhere in the thesis. Here, to clarify the proxy's behavior, we include a scenario for the proxy interface to computational applications. Below, each behavioral component of the proxy is underlined.

Computational proxies generate application input files using information stored in an experiment object and templates that describe input formats. The proxy launches or starts a computational process, and controls that process as long as the process is active. When the process has terminated, the proxy moors the application process by marking the experiment object complete. The proxy then parses application output files using templates that describe these output files and loads results into the experiment object. The functions above may be performed across heterogeneous compute hosts.

Proxies interface computational applications to the object-oriented database system roughly as follows. When a chemist indicates that he or she intends to conduct a computational experiment, an instance of a computational proxy is generated within the database to represent that experiment. A scientist creates or selects an initial molecular structure,

[3]We did not implement a user interface for application registration. In our prototype, application-specific templates and information about where applications run are loaded into the database from ASCII files.

views previous experiments on similar molecules, and uses those previous experiments as exemplars to build or select appropriate inputs and parameters for a new experiment. The proxy mechanism uses this image of the experiment in conjunction with descriptions of the application syntax known collectively as an *input template* to generate input to the computational process. The proxy then ships the input to the compute host and invokes the application, starting up a computational process according to specifications in the experiment object. The chemist need not log on to the compute host. Thus, the proxy hides both syntactic details of the application and architectural details of the network. The proxy periodically monitors the computational process, and maintains a record that the chemist can access, independently of network details or current availability of a remote machine on the network.

When the experiment has finished, the proxy automatically transfers the output file to the database. Using descriptions of the application's output known as *output templates* the proxy reads the output file and converts results into a common format independent of the application and loads the results into the database experiment object. The proxy provides for the automatic capture not only of outputs of the computational process but also of associated metadata of the experiment such as date, time, and performing chemist. The proxy mechanism stores and possibly transforms these data so that they can later be displayed in a common format regardless of what application generated them. Once the chemist has analyzed results, the experiment can be repeated by replicating the experiment object, changing values as needed, and generating a new proxy.

The above functional specification of the proxy led to the logical design of its functional components: input and output templates, application registration, data input and data capture, and mission control. The following section of the thesis describes our design of these components.

4.3 Functional Components of the Computational Proxy

To provide communication between the database and the computational process, the proxy supplies the following services: *data input* to computational applications, *data capture* from their output, and *control* of the application processes themselves. To perform these functions, the proxy accesses information about each computational application from its templates; Section 4.3.1 contains a conceptual model for input and output templates, and shows how templates relate to other database entities. Section 4.3.2 covers registration of new applications, i.e., how information about applications are entered into the database and templates are specified. Sections 4.3.3 and 4.3.4 show how the proxy mechanism uses templates to build input files and parse output files.

Mission control, addressed in Section 4.3.5, provides for starting or stopping a computational experiment, examining intermediate outputs, and querying the use of resources during execution.

4.3.1 Input and Output Templates

This section defines template entities in conceptual modeling terms. A *template* describes a syntactic variant of one or more domain database entities. Input and output *templates* describe application inputs and outputs, and direct the proxy's mapping of objects from the domain-specific database to application-specific inputs and from application-specific outputs to the database. A template object defines the translation from a particular database object to a textual representation of that object and vice versa. Thus, for example, an output template for the molecule entity for the GAMESS application shows how to transform a textual representation of molecule in the GAMESS output format to a molecule object in the database.

Figure 4.5 shows the relationship between other database entities and the input template. An input template for a particular experiment type for a particular application is generated from a sequence of statements written in the Computational Chemistry Input Language. For each application and experiment type supported by that application, there is one input template, as defined by the ternary relationship (**describes inputs**) between *experiment type*, *application* and *template*.

Output templates are organized somewhat differently. A list of output templates called the master template comprises the definition of the output file for a particular experiment type and application. There is one master template for each particular experiment type

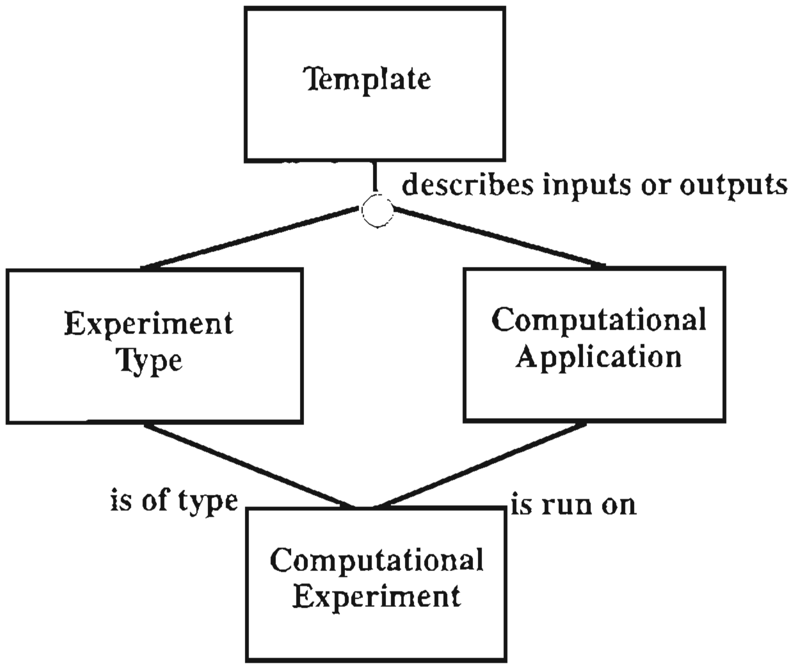


Figure 4.5: Conceptual model for template.

for each application. Any given application will need an output template for each database object to be parsed. The occurrence of an output template for an experiment object in the master template for a given application and experiment type indicates that we wish to load data for that experiment object after running an experiment of that type using that application.

Assume, for example, that the GAMESS application supports two experiment types, ENERGY and OPTIMIZE, and that for ENERGY experiments we will load an energy value, and for OPTIMIZE an energy value and an (optimized) molecular structure. Assume further that output files for the ENERGY and OPTIMIZE experiment types represent energy in the same format, so that only one output template is needed to describe how to load the energy object. Two master templates, i.e., two lists of templates, are required: one list (for ENERGY experiment type) consists of just an energy-value template. The other list (for OPTIMIZE) consists of two templates, the energy-value template and a molecular structure template. There is only one energy-value template for this application, but it appears in two master template lists.

Of course, if application-specific formats for output files differ for a given object depending on the experiment type, there must be an output template for each format.

4.3.2 Application Registration

Before a computational application can be invoked through the database proxy mechanism, it must be registered with the database. In this section, we specify the information about applications that the proxy needs and describe the process by which this information is entered in the database. Recall that the person who registers an application is called the *application registrar* or simply *registrar*. The registrar is a computational chemist who understands the semantic and syntactic profiles of the application being described. The registrar need not be a programmer or have extensive knowledge of database systems software, but he or she must understand the computational experiment schema and be able to map textual elements in the application's input and output files to the corresponding elements in the experiment schema.

To register an application the registrar provides a description about the application and its computing environment sufficient to run an experiment and capture its output. Information about the application (name, allowable experiment types, target processor type, relative location of files, etc.) is loaded into the computational application database object. Information specific to the application's input and output files is placed in input and output templates. Where a template cannot describe data conversions, special-purpose functions to convert database types to or from textual formats for input or output files must be added to the database schema as methods for the appropriate database object. If the textual format cannot be described using existing database types, the registrar must also add a new type to the database. We call such a type a *foreign type* because it cannot be described in terms of the types native to the application schema or the database management system. Even though the registrar is not a programmer, he or she should be able to recognize whether a new application being registered will require conversion functions or new database types. If so, the database administrator (a programmer) must modify the database schema and perhaps write input functions for the new types.

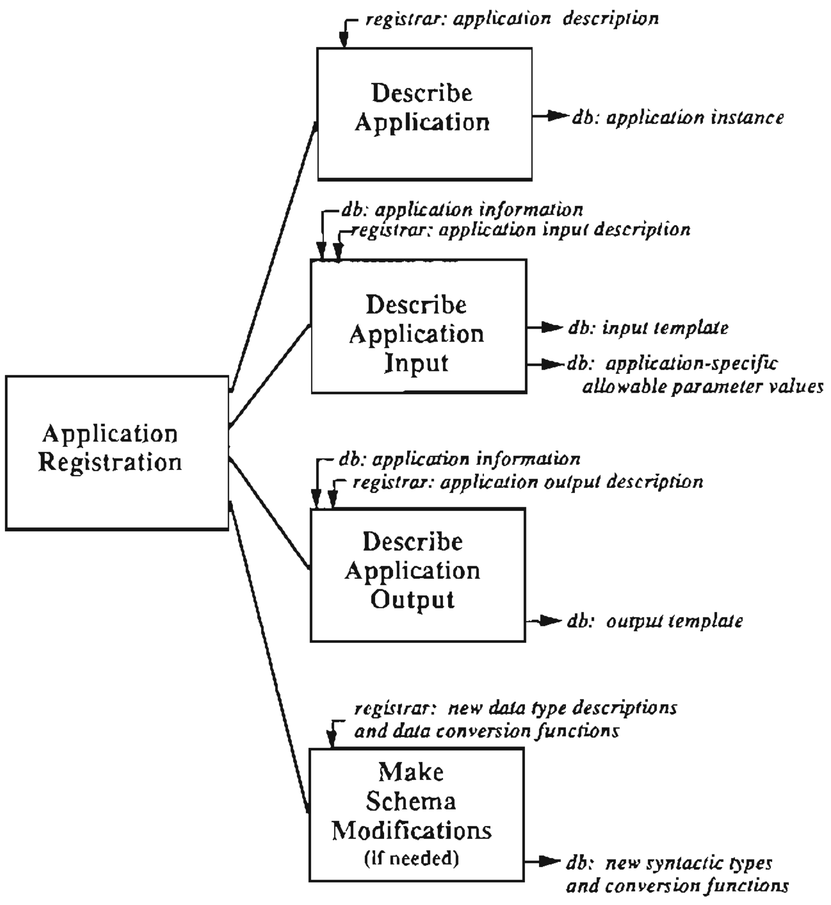

registrar: application description
Describe Application
db: application instance
db: application information
registrar: application input description
Describe Application Input
db: input template
db: application-specific allowable parameter values
Application Registration
db: application information
registrar: application output description
Describe Application Output
db: output template
registrar: new data type descriptions and data conversion functions
Make Schema Modifications (If needed)
db: new syntactic types and conversion functions

Figure 4.6: Application registration functions.

Figure 4.6 depicts application registration functions performed by the registrar. Specific information required for each of these activities is shown at the leading edge of arrows pointing to the box representing each function. Components of the database that are changed as a result of each function are shown at the head of the arrows emanating from the box.

Information about the Application

Information about the application is stored in the *Computational Application* object, which consists of the following descriptive information about an application.

1. Name, author and publisher of the application.

2. Date and time when the application became available.

3. Any remaining details about the application, for example, the maximum "l-value"[4] for a specific basis function that an application supports, the maximum number of Gaussians in a contraction for each l-value, and whether the code supports pure spherical components of the Cartesian Gaussians (e.g., 5 component D's).

In addition to the information above, relationships between the Computational Application instance and other database objects must be instantiated:

1. Programming language and language-version of the application.

2. Experiment types, i.e., computational functions (computational algorithms) performed by that application, such as RHF (restricted Hartree-Fock), UHF (unrestricted Hartree-Fock), SDCI (singles and doubles configuration-interaction).

The registrar must be aware of nuances in the conceptual model, such as distinctions between computational application and version of computational application. As discussed in Section 3.2.4, a single instance of a computational application (from this point on called "application") can be versioned. Database information about applications resides either in the *application* object or in the *version-of-application* object. (Recall Figure 3.5.) When an application is first registered, two objects are created; the application object contains information that we believe is common across versions of a computational application (name, publisher, etc.) and the version-of-application object contains information specific to a particular version such as experiment type and maximum l-value. Application objects and version-of-application objects are related to each other in two ways: (1) an application may have several versions, but (2) each application has only one current version. The publisher's version number of an application is part of the version-of-application object. A particular experiment is run using a version-of-application, though the user need not be aware of the distinction between an application and version-of-application. Thus, for example, a particular installation may register version 3.0 of Gaussian as its first version

[4]The term *l-value* refers to the value of the angular momentum quantum number as represented by the spherical harmonics s, p, d, f, etc.

of Gaussian. The next release of Gaussian, say version 4.0, will be the second version of Gaussian registered. The database differentiates experiments run using version 3.0 from those using version 4.0. Input and output templates are associated with a version-of-application, since file formats may change with releases of the application. Of course, application versions may share input and output templates. (For simplicity's sake, we have in this thesis largely ignored the distinction between running different versions of an application.)

Another example of information that differs among versions of application is programming language version. Version 3.0 of Gaussian may have been written in Fortran 77 v3.0, and version 4.0 in Fortran 88 v1.0. The particular compiler for the application is a function of which platform and machine the application is installed on, and is discussed in the section on the computational proxy architecture (Section 4.4).

Information about Input and Output Files

Computational chemistry applications produce two kinds of files of interest to the proxy, intermediate and output files. Intermediate files are rarely directly examined by the human user and typically contain very large matrices written during the n^{th} iteration of the simulation. These files are read by the application as input to the $(n+1)^{st}$ iteration. Intermediate files were historically used to store matrices too large to fit in their entirety into memory.

The output file contains experiment results. Though normally consulted only when the experiment has completed, the output file is often used to store information about the experiment as it is running. This file can be consulted while the experiment is running to determine if the experiment is converging, and, if so, how close it is to convergence.

In some cases intermediate or output files can be used to restart a computation that has been stopped or aborted. The computational proxy can read intermediate or output files while an experiment is in progress and respond to user queries about how close the calculation is to convergence. Since a badly specified experiment might not terminate, or might run for months, even very rough estimations are useful.

Information about the syntax of the applications' input and output files is stored in input templates and output templates, respectively. As explained in Sections 4.3.3, 4.3.4 and 4.3.5, the proxy uses the templates to perform input file generation, output capture and domain-level experiment monitoring. Below is a description of input and output templates and the process by which they can be added to the database.

1. Input Template. An input template specifies the layout of an application's input file for a particular experiment type. An input template contains information sufficient to generate the input file for that application given an experiment object in the database. As part of application registration, an input template instance, couched in the "Computational Chemistry Input Language" (CCIL), is loaded into the database. The input template is stored in the database as CCIL text (actually a sequence of CCIL statements). This sequence of CCIL statements is input to the proxy's input file generator.

2. Output Template. Analogously, output templates specify the syntax (format) of an application's output file. An output template maps a textual data object to a database object, and defines any required syntactic translation. A given output template is specific to an application, an experiment type, and an object in the experiment database. The ordering of templates into a *master template* list defines the order in which the computational proxy would most efficiently parse the output file. There is one master template list per application and experiment type.

More detailed information about these templates follows.

4.3.3 Data Input: Generating Input Files from the Database

We distinguish information about the subject of experiment and experimental parameters. The subject of the experiment is the molecule itself. Information about the subject of the experiment is added to the database prior to the request to schedule an experiment, via some domain-specific editing facility (e.g., a graphical molecule editor) or by copying from a previous experiment.[5] We have made the simplifying assumption that scientific data

[5]Since computational chemistry experiments often update (optimize) molecular structure and since molecular structure is the defining characteristic the CCDB molecule object, "copying" a molecule instance

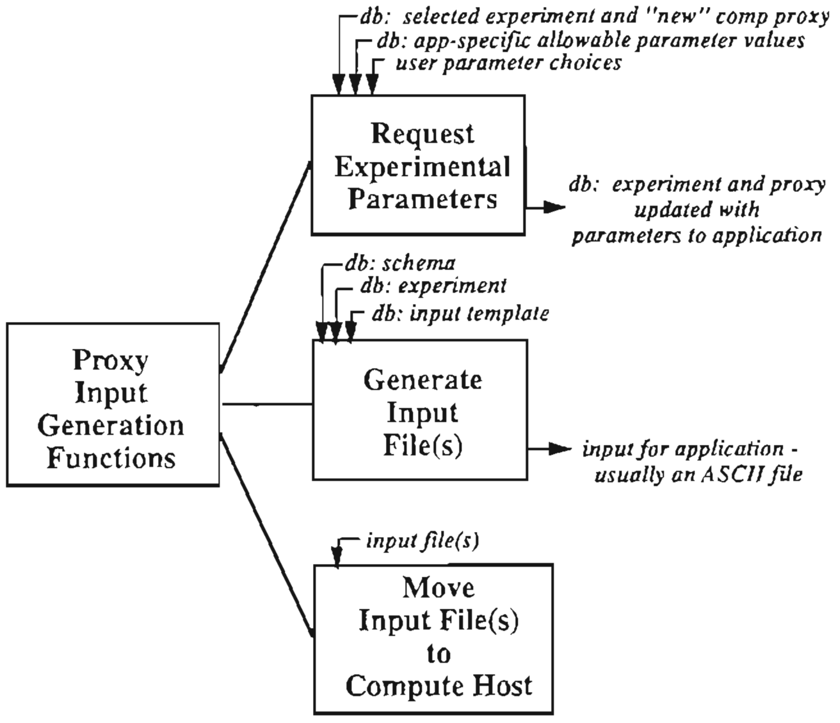

db: selected experiment and "new" comp proxy
db: app-specific allowable parameter values
user parameter choices
Request Experimental Parameters
db: experiment and proxy updated with parameters to application
db: schema
db: experiment
db: input template
Proxy Input Generation Functions
Generate Input File(s)
input for application - usually an ASCII file
input file(s)
Move Input File(s) to Compute Host

Figure 4.7: Application input functions needed by the computational proxy.

describing the subject of the experiment are present at the time the user asks to launch an experiment. Experimental parameters (described below) define the experiment to the application.

The proxy's **data input** component has three functions: requesting experimental parameters from the user, generating the application input file, and moving the input file to the processor (called the compute host) on which the application is to run. Figure 4.7 depicts these functions; inputs to (and outputs from) each function are shown in italics above (and to the right of) the box enclosing the function name. Inputs from or updates to the database are prefixed by **db:**. The three data input functions are further explained below:

1. **Request Experimental Parameters.** Before the input file is generated and the

to set up a new experiment adds a new molecule instance to the database, rather than creating a shared reference to an existing molecule.

experiment launched, missing input parameters (if any) must be supplied.[6] Some parameters, such as experiment type, can be requested in terms general to the domain and later translated into application-specific formats, but other parameters may be application-specific or relevant only to a given site. Examples of experimental parameters generally applicable across several applications include: the self-consistent wave function (SCFTYP) and the Moller-Plesset perturbation level desired; the type of output required (if not obvious from the data item requested) such as minimum energy only, minimum energy plus field gradients, or geometry optimization; and the units in which the application should cast certain outputs. An example of a parameter specific to one application (Gaussian) is the parameter that specifies that a special checkpoint file be saved. This output file is usually deleted after an experiment, but it could be used to restart the application efficiently. An example of site-specific information is the machine on which to run the experiment.

2. **Generate Input File.** When a user request triggers the launching of a computational experiment, the proxy generates an input file for the experiment, formatting data appropriately for the application using an input template. Which parameters are required is a function of experiment type. If an experiment type requires a specific experimental parameter, then one or more CCIL statements in the input template generate the textual form of that parameter. We make the simplifying assumption that input to a computational application consists of a single ASCII data file. Although some applications require more than one file as input for certain experiment types, implementing a multi-file or object-level interface is not conceptually more difficult than generating and shipping a single ASCII file.

3. **Move Input File to Compute Host.** The computational application is rarely run on the same machine as the database itself. Thus, the proxy must take care of sending the input file to the processor on which the application is to be run.

[6]In a full implementation, a user interface will inspect the experiment for completeness and conduct a "conversation" with the user to complete the experiment. The proxy will then check whether the experiment is fully specified before submitting it for execution.

```
$CONTRL TIMLIM=999.0 MEMORY=2000000 $END
$CONTRL SCFTYP=RHF UNITS=BOHR $END
$CONTRL RUNTYP=OPTMIZE $END
$CONTRL MPLEVL=0 $END
$DATA OPTIMIZE CONV RHF H2/STO-3G/MP0
C1
Hydrogen 1.000000 2.325134 1.729993 0.000000
   S 3
1 3.425251 0.154329
2 0.623914 0.535328
3 0.168855 0.444635
Hydrogen 1.000000 -2.325134 1.729993 0.000000
   S 3
1 3.425251 0.154329
2 0.623914 0.535328
3 0.168855 0.444635
$END
$GUESS GUESS=MINGUESS $END
```

Figure 4.8: Sample input file for a GAMESS experiment on hydrogen.

For example, to perform an experiment to optimize the geometry for the hydrogen molecule using the computational application GAMESS, a chemist first tells the system that he or she wishes to run such an experiment. In response, the proxy creates a new experiment object of experiment type "optimize" that uses the GAMESS application. Once the chemist has specified additional input parameters and is ready to run the experiment, the proxy generates an input file that specifies those input parameters.

The input file given in Figure 4.8 controls an invocation of the GAMESS application for a conventional RHF experiment run on hydrogen, i.e., SCFTYP = RHF and RUNTYP = OPTIMIZE. The level of theory specified is 0, and the symmetry of the molecule is assumed to be of type C1 (none). Following the x-y-z coordinates of each atom is the basis set. To generate this input file, the proxy mechanism uses an instance of input template for the "optimize" experiment type for the GAMESS application. The input template object is defined to the database using CCIL. (A praecis of CCIL follows.)

Once the input file is generated, the proxy launches the experiment. While an experiment is running, the proxy monitors the experiment at regular intervals and at user request. Once the experiment process has terminated, the proxy moors the experiment and captures its results.

The Computational Chemistry Input Language

The Computational Chemistry Input Language (CCIL) is a layout language designed to define ASCII text (input) file formats for ab initio computational chemistry applications. The proxy mechanism that generates application input files expands CCIL references to database objects to produce text values for experiment data stored in the database. The CCIL is in effect a special purpose "report writing language", providing for specially formatted reports of experiments from the database.

A sequence of CCIL statments for a particular experiment type and application is called an input template. In the context of a particular experiment, the proxy uses an input template object to generate an input file for that experiment. The sequence of CCIL statements that constitutes an input template object could be written directly by the registrar, but we think of the CCIL as an intermediate language to be generated by the registrar user interface. In either case, a description in CCIL of an application's input file format must exist for each experiment type supported by that application before the proxy can generate input files for experiments with that experiment type and application.

The *CCIL Interpreter* is the proxy method that reads the input template object in the context of a given experiment and produces an input file. The interpreter is always invoked in the context of a particular experiment and expands database references within the input template so that experiment data from the database is included.

The language specification for the CCIL follows below. The reader will find it helpful to refer to the example input file and CCIL code in Figures 4.8 and 4.9, respectively.

- An input template is a sequence of statements written in the CCIL.
- CCIL statements are separated by semi-colons.
- A CCIL statement can consist of an "ITERATE OVER" command. The body of an ITERATE OVER command is a sequence of CCIL statements. One can ITERATE OVER a list of CCIL statements or a database collection. Iterating over a list of CCIL statements generates a sequence of CCIL statements. Iterating over a database collection causes the interpreter to retrieve a collection of database objects

one by one and transform each object in that collection to a textual representation. Examples of these two alternatives follow in the discussion below.

The full syntax of the ITERATE OVER command is:

```
USING <loop-control-variable> ITERATE OVER <some-collection> {
  <a-sequence-of-CCIL-statements>}.
```

- A CCIL "USE...FOR" statement defines a variable that, when later used in a sequence of CCIL statements, is replaced by a literal given in the USE statement. The syntax of the USE statement is:

  ```
  USE <variable-name> FOR <literal>.
  ```

- A CCIL "DEF" statement defines a format, a list of statements, or a sequence of text characters. Formats and lists of statements must be named, and must be defined before reference.

  - A CCIL **format** defines a set of simple transformations, each from a value (in the database) to a textual representation for that application. Each transformation is given as a value pair separated by "`->`". The database value is given first, followed by " `->` ", followed by the textual equivalent. Sequences of value pairs are enclosed in brackets "`{...}`", and each value pair is given on a separate line. The database value may or may not be a character string. The textual representation will most often be given as a character string, but it might alternatively be given as a type for which a string conversion method is known to the CCIL interpreter. Textual representations must be enclosed in quotes; database values must be enclosed in quotes if the type of the corresponding database object is a character or string type.

    ```
    DEF format <name-of-format> {
       database-value -> "textual-representation"
       database-value -> "textual-representation"
       . . .}
    ```

For example, assume that a level of theory value in the database is given as (the character string) "MP1" and that an application requires its corresponding level of theory to be given as (the character string) "1". Statement 3 in Figure 4.9 shows such a CCIL format transformation for this situation.

When a format name is later referenced in a CCIL statement the "name of format" is preceded by "%".

– A CCIL **list of Statement** defines a list of CCIL statements. A defined list of CCIL statements can be used in an ITERATE OVER command to generate a sequence of CCIL statements.

```
DEF list of Statement <name-of-list-of-Statements> {
    CCIL statement;
    CCIL statement;
    ...}
```

In Statement 4 of the CCIL example in Figure 4.9, the list of Statement ControlStatements is defined. In Statement 5, the ITERATE OVER feature is used to generate a sequence of CCIL statements each beginning with "$CONTRL" and each ending with "$END", as in the first four lines of the example input file in Figure 4.8. The loop control variable for this ITERATE OVER statement is ControlStatement; it marks the place in the body of the ITERATE OVER where an element from the collection is placed.

– A CCIL statement not preceded by "DEF" defines a sequence of ASCII characters to be written to a file. The text is generated in the order in which the CCIL statements appear. The interpreter generates a linefeed character after generating text for each statement.

Text to be generated by the interpreter is given as CCIL terms. A character string generated by a CCIL term is called a "textual term". CCIL terms expressed in a single statement are separated by one or more spaces; the interpreter generates a space to separate two textual terms on the same line in the output file. A CCIL term can be:

* An escape sequence. Escape sequences are defined as per the language C [99]; those of interest are newline (`\n`) and tab (`\t`). No spaces are generated after an escape sequence.
* A string literal. A string literal is a sequence of characters (as defined for C) surrounded by quotes, as in "$CONTRL".
* A database object. Database objects are referenced using path expressions. A CCIL path expression consists of a class name followed by one or more data element names. Relationship names are preceded by "->", and attributes of classes by ".". The head of the path expression is the current experiment object in context. The proxy input file generator is called in the context of a particular computational experiment; thus "self" is the current computational experiment in context, an instance of the CCDB class "CompExperiment". All CCIL path expressions are resolved from the binding of "self". The name of a database path expression is enclosed in single quotes in the CCIL.

  Database objects, unless otherwise specified, are displayed by the interpreter using the standard display function for that object.

  For example, 'CompProxy.`timlim`' is a CCIL reference to an ObjectStore database object; "timlim" is an attribute of the CompProxy object. Where 'CompProxy.`timlim`' appears in a CCIL statement, the value of the current instance of that object will be displayed.

  A database object can be converted to an alternate format using one of two format conversions: a format conversion defined in the CCIL program by a format statement or a format conversion defined in the database schema as a method for that database object. Format conversion names and database method names are preceded by "%". Where a format conversion and database method have the same name, the format conversion name takes precedence.

  For example, in Figure 4.9 two format conversions are specified. In Statement 4, the database object 'CompExp->isTakenTo.`name`' is transformed according to the "LevelOfTheoryFormat" format conversion defined in the CCIL program itself. In Line b of Statement 9, the database object `'atom.name'` is formatted according to the database method for the atom class "asAtomicNumber".

* CCIL-provided operations, CARDINALITY and COUNT. The CCIL term "CARDINALITY of <database object>" will output the cardinality of the database object in character format.

  The CCIL term COUNT refers to the implicit counter maintained by the interpreter as it ITERATEs OVER a collection. If the keyword COUNT occurs in a CCIL statement in the context of an ITERATE OVER command, the interpreter prints the value of the counter at that point in the iteration.

  For example, Lines 9e and 9f in Figure 4.9 ITERATE OVER the database collection "BasisSetForAtom". Since there are three contractions in the basis set instance for this atom, invoking COUNT as the first element in each iteration will cause the three sequences of output to begin with "1", "2", and "3". Statement 9 also contains the term "Atom.`COUNT`". `Atom.COUNT` is used to index into the list BasisSetForAtom. Because there are two hydrogen atoms for each hydrogen molecule, there are two basis set lists for each basis set for hydrogen, each corresponding to an atom in the molecule.

The CCIL also provides for specifying default inputs and for constraining maximum values for a database object. Thus, for example, `'CompProxy.timlim'/0//999.0` defines application-specific values for default and maximum for `CompProxy.timlim`. The value that appears after the first "/" is the default value to be written if the value of the database object is null. The value appearing after the second "/" would be the minimum value to be reported. After the third "/" is the maximum value to be reported. If the value of the database object exceeds the maximum value, the maximum value given in the template is printed. (We found no need for minimum values for computational chemistry applications, but include them for completeness.)

Names of CCIL formats and lists are formed as per C++ variable names [173], but the following keywords and symbols are disallowed in CCIL names: "DEF", "format", "list of Statement", "USE", "FOR", "USING", "ITERATE OVER", "->", "%", singlequote, doublequote, and semicolon.

Using The CCIL

To add a new application to the database, one must of course describe the application's input file(s) in the CCIL language, for each experiment type the application supports. This involves writing a sequence of CCIL statements and loading them into the database as the input template for that experiment type and application. To use the CCIL to describe the input files for geometry optimization experiments for GAMESS, for example, one writes the CCIL statements in Figure 4.9, and loads these into the database as an instance of the input template class for experiment type "optimize" for application "GAMESS". Given this input template in the context of a particular experiment to optimize the structure of hydrogen using the GAMESS application, the proxy would generate the input file given in Figure 4.8.

There are three ways in which writing CCIL code alone might not be sufficient for generating input files for an application:

1. The new application's input file may require a format conversion that cannot be expressed in the CCIL. In this case, a new database method to perform the needed format conversion for that experiment object must be added to the database schema.

2. The application schema may not be adequate to represent the data required for input to the new application. Here, the schema must be modified to include the new data type and methods provided (as needed) to transform existing data types to and from the new data type.

3. The CCIL may not be adequate to define the input file. Here, the CCIL grammar must be extended and the CCIL translator modified to cover that extension.

```
 1. USE CompExp  FOR self;
 2. USE Proxy    FOR self->isRepresentedBy;

 3. DEF format LevelOfTheoryFormat (
     "MP1" --> "1";
     "MP2" --> "2";
     "MP3" --> "3"
    );

4.  DEF list of Statement ControlStatements (
     "TIMLIM=" 'CompProxy.timlim'/100//999.0;
     "MEMORY=" 'CompProxy.memory'/100000//20000000;
     "SCFTYP=RHF UNITS=BOHR";
     "RUNTYP="'CompExp.expType'/"ENERGY"//"OPTIMIZE";
     "MPLEVEL="'CompExp->isTakenTo.name'%LevelOfTheoryFormat
    );

 5. USING ControlStatement ITERATE OVER ControlStatements (
     " $CONTRL" ControlStatement "$END"
    );

 6. " $DATA" 'CompExp.name';
 7. 'CompExp->hasAsSubject.symmetry';

 8. USE BasisSetForAtom FOR CompExp->utilizes->atomBSList;

9a. USING Atom ITERATE OVER 'CompExp->hasAsSubject->hasAtoms' (
9b.   \n 'Atom.name' 'Atom.name'%asAtomicWeight 'Atom.x' 'Atom.y' 'Atom.z'
9c.   \n "    "  'BasisSetForAtom(Atom.COUNT).label'
9d.              CARDINALITY of 'BasisSetForAtom(Atom.COUNT)->contractions';
9e.   USING Contract ITERATE OVER 'BasisSetForAtom(Atom.COUNT)->contractions' (
9f.    Contract.COUNT 'Contract->primitives.coefficient' 'Contract->primitives.exponent'
    ));


10. " $END";
11. " $GUESS GUESS=MINGUESS $END";
```

Figure 4.9: Input template for optimize experiment type, GAMESS application.

If, as we believe, both the application database schema and the CCIL are adequate to express textual data for the major experiment types for the ab initio computational chemistry applications in current use, the computational proxy will be able to generate input files for new applications without any of the special-purpose programming above.

4.3.4 Data Capture: Loading Experimental Results into the Database

Data about experiments to be stored in the database fall into three major categories: *inputs to* experiments, *metadata about* experiments[7], and scientific *results of* experiments. Experimental inputs are stored in the database as a consequence of the proxy's central role in building and launching the experiment; the same is true of metadata. Metadata include such information as which chemist performed the experiment, the date and time it was conducted, the application used, the version of that application, the computer type and specific processor on which the experiment was run, and the resources consumed in running the experiment. Metadata are essential for replicability and proper interpretation of experimental results.

Storing metadata is a natural byproduct of using the proxy mechanism for experiment management. Since the database launches the experiment, information such as resources used can be gathered quite easily. Metadata of a scientific nature, such as the specific algorithm used by the application to perform the calculation, can also be automatically gathered during the process of managing the experiment or inferred later from other information in the database.

Loading experimental results is more complicated than gathering metadata because the applications do not interface directly with the database. Output typically resides in one or more text files, and capturing the output of computational chemistry applications involves parsing these files and placing the output into the database. (For simplicity, we have assumed that all data to be captured reside in a single file: capturing data from more than one file is not conceptually different.)

[7]We here use the term *metadata* as it is used in scientific database community: identifying and clarifying information about scientific data. The traditional database sense of *metadata* refers to structural information about the data, i.e., the database schema.

The data capture problem is more difficult than the input file generation problem for two reasons: reading (parsing) text is in general more difficult than generating it, and the particular parsing problem facing us is not simple. That problem is difficult because the text files in question were usually designed for printing on paper or displaying on a screen, and formatted so that human readers would find them easy to understand. In other words, output text was designed for human rather than machine consumption. Furthermore, we could not easily discern the grammars that generated the output files for the applications we studied — whatever implicit grammars may have been used, they are (in our estimation) neither context free nor syntactically unambiguous. Our parsing and capture task is further complicated by a major design goal: to provide a declarative (not procedural) specification of the application interface.

We also found an asymmetry between the amount of documentation available for application input and output. Application manuals are more explicit in their description of input syntax; their authors assume (apparently) that a human computational chemist can easily read and interpret the output relying on an understanding of the context. Thus, we knew less about output text than about input text. Finally, the dependence of the output structure on input parameters is not described explicitly for most applications.

We capture output from computational experiments in a two-step process, as Figure 4.10 depicts. An output file is first moved to the database host; once there it is parsed and data placed into the database.

Our analysis of user needs uncovered two general cases of data capture:

1. **Application process terminates successfully.** In this case, once the proxy is notified of successful termination of a computation process, it parses experimental results and places them into the database. Once captured, the data resides in the database and can be viewed by the user. Data capture should be distinguished from experiment validation. Data capture, an automatic proxy function, precedes data validation, an explicit user action. The user, not the proxy, validates experiments; once the user decides whether an experiment was "successful", the proxy marks the data appropriately.

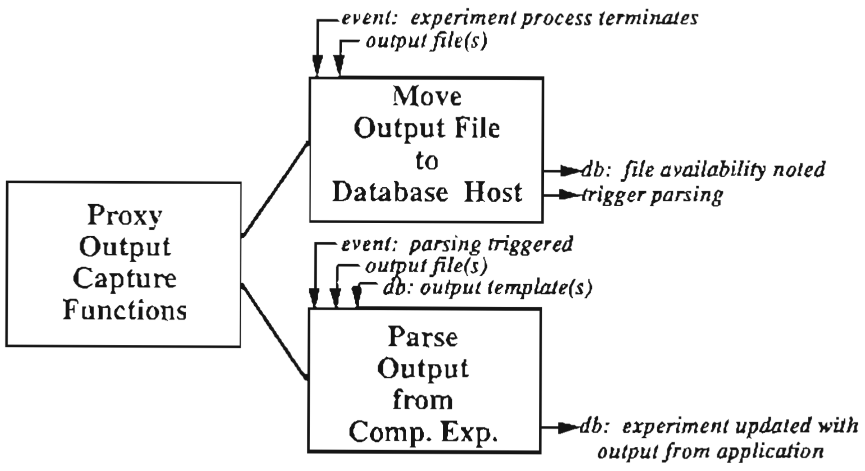

event: experiment process terminates
output file(s)
Move Output File to Database Host
db: file availability noted
trigger parsing
Proxy Output Capture Functions
event: parsing triggered
output file(s)
db: output template(s)
Parse Output from Comp. Exp.
db: experiment updated with output from application

Figure 4.10: Application output capture functions needed by the computational proxy.

2. **Process terminates, but the computation is incomplete.** Data capture after abnormal termination of an experiment is less straightforward. A computation may fail to complete due to a lack of disk space or to the process needing more CPU time than was allocated. In this case, the user may wish to view the output file directly. The user also may wish to parse the output file, place intermediate results into the database, change input parameters, and restart the computation where the aborted process stopped.

Both cases require application-driven parsing of a text file. Loading a single database object from a text file involves:

1. Finding the physical location of the desired datum within the text file. In all cases that we examined, the begin point of the data to be captured can be found by searching on a keyword and then skipping application-specific whitespace or control characters. Because the parsing task may not be context free, however, the end point of the data in the text file might not be easily discernible by the parser. Thus the proxy's parsing mechanism should use a bounded parsing technique, for example by bounding the text to be parsed with both beginning and ending keywords.

2. Identifying the type of the textual data and its physical format. We use the term

textual type to refer to the format of the ASCII data to be parsed. Textual types may be either primitive data types, such as *integer* and *string*, or user-defined types. Our parsing mechanism uses C data types [99] to represent primitive textual types. User-defined textual types can be defined in the database, and are represented by the database schema names; database-supplied input methods are used to read user-defined textual types from the textual file.

3. Identifying the database object into which the parsed data is to be placed, the *target database object*. A path name, as it appears in the application database schema, taken in the context of a particular computational experiment, is used to signify the target database object.

4. Identifying the type of the target database object, the *target type*. The proxy's parsing mechanism determines the target type by reading the database schema; in our case, this is the ObjectStore schema for the Computational Chemistry Database.

5. Determining whether the parsed data must be converted before placing it into the database, and if so, what conversion function(s) to apply. When the type of the textual data and the target type do not match, we have a type clash. Properly handling a type clash requires invoking a conversion function, and can involve units or type conversion.[8] We defined some intermediate collection types, such as *Matrix*, for parsing formatted text into complex objects such as *MolecularOrbitals*. The proxy's parsing mechanism uses both standard C++ compiler-supplied type conversions (e.g., *float* to *int*) and user-defined computational-chemistry-specific type conversions, such as *atomicNumberToAtomicName*.

6. Performing a type conversion, if necessary, by invoking the appropriate function and passing it the object to be converted.

7. Placing the data into the database.

[8]For simplicity's sake we have ignored unit conversions. Mechanisms used for handling type conversions can be applied to unit conversions.

Of course, loading the output for most applications requires parsing more than one database object from a text file. The scenario above is followed for each object to be loaded, and there is one output template describing each object to be loaded. For loading more than one object from a file, the proxy parser-loader uses the master template (list of output templates).

When adding a new application to the database, one first verifies that textual types in that application's output file(s) can be represented in the data types already present in the application schema. If so, then no schema modification is required. Syntactic differences in layout, such as a matrix represented in row-column order as opposed to column-row order, do not constitute a type difference. An example of a type difference is a molecular structure represented as a internal coordinates as opposed to Cartesian coordinates; accommodating such a type difference in our system would require a schema modification, namely adding a new type to the schema, with an input function for it and a conversion function from the new type to the target type.

The PCL: Proxy Methods for Loading Output

The proxy methods for loading output data have been dubbed the Parser-Converter-Loader (PCL). The PCL is modeled after EXPRESS (Data EXtraction, Processing, and REStructuring System), an experimental prototype data translation system designed to extract a wide variety of data from files and restructure it for inclusion in a database. EXPRESS was driven by two very high-level nonprocedural languages: DEFINE for data description and CONVERT for data restructuring. LOAD, a third component, loaded the converted data into the new database[167]. EXPRESS inspired us to organize the proxy's data capture tasks into three phases: define, convert and load. (Below, the term *textual object* refers to the string in the text file that we want to load into the database.)

1. During the define phase, the parser processes the text file, picks out a textual object of interest, and performs syntactic reformatting if needed. Textual objects in computational chemistry output files are identifiable by keyword searches, e.g., the last occurrence of the keyword "ENERGY" precedes the desired energy value.

2. In the convert phase, statements assigning a textual object to another textual object or to a target database object are read and processed. Type clashes are resolved by invoking the appropriate type conversion functions.

3. The load phase actually loads a converted textual object into the database.

An example of output from the Gaussian application illustrates the extent of the parsing problem with which we are confronted, and how the three phases of the parser work to solve this problem. Gaussian represents molecular orbitals as matrices. When there are too many columns in a matrix to fit on one $8\frac{1}{2}$-inch-wide page, a matrix will be "folded". Thus, for example, a 256x256 matrix might be folded into twenty-one 12x256 (column by row) submatrices and one 4x12 (column by row) submatrix, each of which fits onto one $8\frac{1}{2}$-inch-wide page. Column and row headers are repeated for ease of reading each submatrix. In other words, labels and data are intertwined, and labels are repeated for human convenience. The human reader easily "unfolds" the twenty-two submatrices into one 256x256 matrix, but for our parser to unfold the submatrices it must distinguish the intertwined labels from data, and ignore redundant labels.

Figure 4.11 shows how a textual matrix is physically transformed by the parser during the convert phase so that it can be parsed and loaded. We use intermediate text types *FoldedMatrix*, *Matrix*, and *DeNormalizedMatrix* to render the textual molecular orbital into a form suitable for parsing. These intermediate collection types are effectively textual layouts, and are defined as follows:

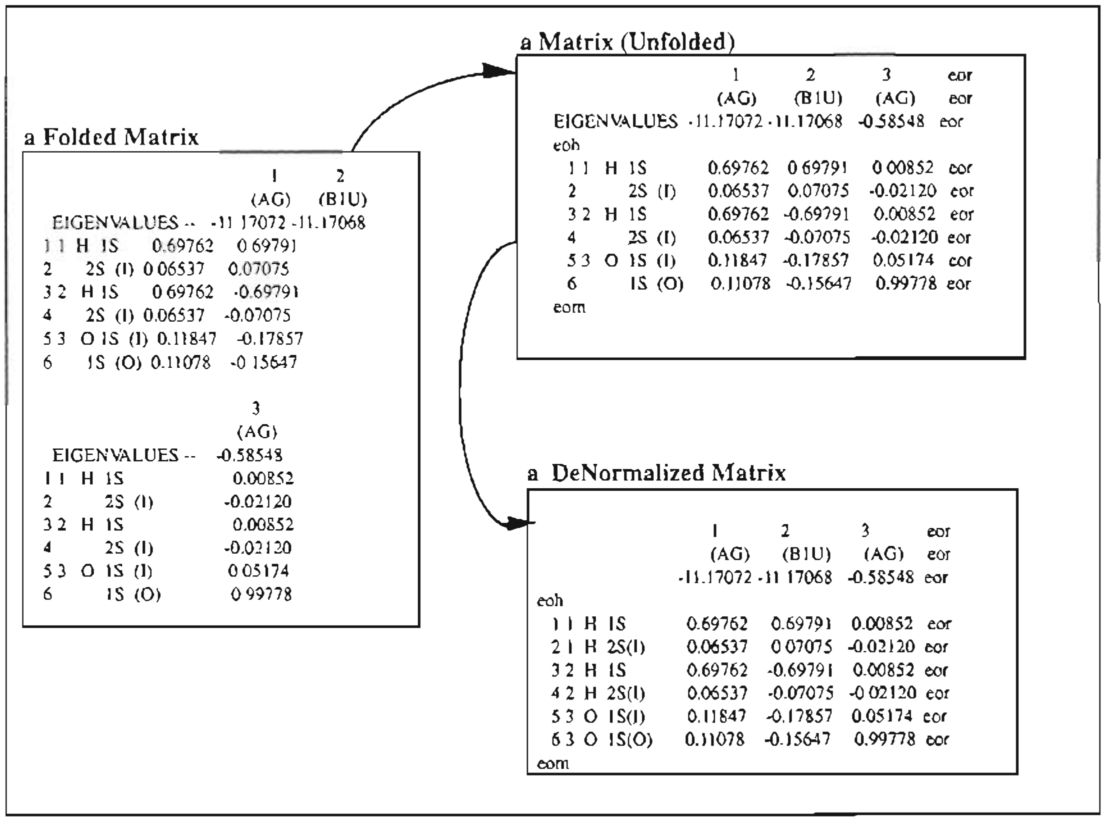
- A *FoldedMatrix* is a matrix whose columns are not contiguous across horizontal space. Thus, a 25x25 folded matrix might have three parts: columns 1 through 10 for all 25 rows, columns 11 through 20 for all 25 rows, and columns 21 through 25 for all 25 rows. Each part of the *FoldedMatrix* is preceded by any number of column headers (**colhead**), and each row of data is preceded by a row header (**rowhead**), consisting of an arbitrary number of items each of arbitrary length. The row headers for corresponding rows in each of the parts of any particular *FoldedMatrix* will be identical.

- A *Matrix* is a row-by-column array of data preceded by any number of column headers (**colhead**). Each row of data is preceded by a row header (**rowhead**), consisting of an arbitrary number of items each of arbitrary length. Transforming a folded matrix into *Matrix* form reconstructs the folded matrix from its parts, but the *Matrix* type is still difficult for a machine to parse because some row header information may be elided. Row header information is typically elided when the row header information for a particular row is the same as the row header information for the previous row.

  The format of the *Matrix* row headers is designed for reading by humans, and is not readily parsable: tokens that may themselves contain spaces are delimited by spaces. Thus, for example, the row headers in Figure 4.11 identify the first two rows to be the electron densities associated with the first hydrogen atom of the molecule. "1S" and "2S (I)" designate the names of the two associated electron shells, and are each one token. The space between "2S" and "(I)" in the second of these tokens would cause most lexical analyzers to read them as two tokens.

- An *Unfolded DeNormalized Matrix* is one whose row headers have been reformatted so that there is an equal number of tokens per row header, so that no token has whitespace, and so that tokens are delimited by whitespace. Column headers have been similarly "regularized", i.e., the number of tokens in each column header is the same as the number of columns in the body of the matrix. Thus, in Figure 4.11 we see that the string "EIGENVALUES", which appears in the third column header of the *Matrix*, has been deleted, and single tokens such as "2S(I)" have no blanks.

To summarize, the computational proxy parser reformats matrix data from forms that are printable on standard size paper and easy-to-read by humans (e.g., a *FoldedMatrix*) to forms that are easy-to-parse (e.g., an *Unfolded DeNormalized Matrix*). Unfolding a matrix is easier and more efficient if information such as the number of rows or columns can be deduced prior to parsing. In the cases we examined, querying the database for the cardinality of certain collections or writing methods to calculate the size of results gave us this information.



```
                              1         2
                            (AG)     (B1U)
  EIGENVALUES --   -11.17072 -11.17068
 1 1  H  1S      0.69762    0.69791
 2     2S (I)  0.06537    0.07075
 3 2  H  1S      0.69762   -0.69791
 4     2S (I)  0.06537   -0.07075
 5 3  O  1S (I)  0.11847   -0.17857
 6     1S (O)  0.11078   -0.15647

                             3
                           (AG)
  EIGENVALUES --   -0.58548
 1 1  H  1S              0.00852
 2      2S (I)          -0.02120
 3 2  H  1S              0.00852
 4      2S (I)          -0.02120
 5 3  O  1S (I)          0.05174
 6      1S (O)           0.99778
```

```
                          1          2          3        eor
                        (AG)      (B1U)       (AG)      eor
 EIGENVALUES  -11.17072 -11.17068  -0.58548   eor
 eoh
  1 1  H  1S        0.69762    0.69791    0.00852   eor
  2       2S (I)    0.06537    0.07075   -0.02120   eor
  3 2  H  1S        0.69762   -0.69791    0.00852   eor
  4       2S (I)    0.06537   -0.07075   -0.02120   eor
  5 3  O  1S (I)    0.11847   -0.17857    0.05174   eor
  6       1S (O)    0.11078   -0.15647    0.99778   eor
 eom
```

```
                        1          2          3       eor
                      (AG)      (B1U)       (AG)     eor
                  -11.17072  -11.17068  -0.58548   eor
 eoh
  1 1  H  1S        0.69762    0.69791    0.00852   eor
  2 1  H  2S(I)     0.06537    0.07075   -0.02120   eor
  3 2  H  1S        0.69762   -0.69791    0.00852   eor
  4 2  H  2S(I)     0.06537   -0.07075   -0.02120   eor
  5 3  O  1S(I)     0.11847   -0.17857    0.05174   eor
  6 3  O  1S(O)     0.11078   -0.15647    0.99778   eor
 eom
```

Figure 4.11: Reformatting GAMESS molecular orbital text for easier parsing.

Output Templates and Parsing Directives

Because the output parsing problem is inherently more complicated than the input generation problem, it is not surprising that the parsing specification (i.e., the output template structure) is more complicated than the input file specification.

Each application-specific output template holds a sequence of parsing directives sufficient to parse a textual object, perform syntactic conversions if necessary and load it into a target database object. There is a one-to-one correspondence between each textual object to be parsed and each database object to be loaded, and one output template per database object. Thus, an output template instance describes the layout of a textual object as it appears in the output file for a given experiment type and application, and maps the textual object to a class in the database. Parsing directives are written in a data description language that we have dubbed the "Computational Chemistry Output Language" (CCOL) and describe below.

For each application, output templates are organized into lists by experiment type. Each list is called a *Master (Output) Template*. The order of templates in each master template indicates an appropriate parsing order. If templates are interpreted (or compiled) in the order of their occurrence in this list, one pass over the output file should be sufficient to parse the entire output file and load all database objects in the mater template. To parse an output file for an experiment of a given experiment type and application, the proxy traverses the master template to find the template for each database object to be loaded. The inquisitive reader at this point might ask what happens if more than one object of the same type occurs in one output file. There are two typical cases: (1) We may wish to read all of the objects into some collection. For example, there is usually more than one atom object to be loaded. Using the structure inherent in the molecule object (that it contains a collection of atom objects), the parser iterates over the template that describes atoms, and accordingly parses, converts and loads atom objects into a database collection of atoms.[9] (2) We may wish to read only one of two or more occurrences of an

[9]How the parser knows about the structure of database objects is an implementation issue. A smart parser would be able to read the database schema for a database object to be loaded, realize that it consisted of a collection of (other) objects and iterate over that collection, using parsing directives for the

object from a text file. For example, there are usually many molecule objects in a given output file for a geometry optimization, one printed at each major iteration point of the algorithm. The parsing directive for the molecule object indicates which occurrence (first or last) is desired. Except in cases where we loaded objects into a database collection, none of the output files we examined contained more than one database object of any type to be loaded.

Output templates may be shared. If an application always represents a database target object as the same textual type regardless of experiment type, only one template (one sequence of parsing directives) is needed. Where two experiment types for the same application engender two different syntactic forms for the same database target object, two templates are needed.

Each template has a sequence of one or more parsing directives. Each parsing directive invokes one parsing action, and directives are interpreted in order of their appearance in the template. Output parsing directives direct the parser to position the cursor in the output file, read the data according to the textual data structure, determine whether the data's type is different from that of the database object (and if so, perform the necessary conversion), and finally load the data into the database.

Figure 4.12 shows the three major inputs to the proxy Parser-Converter-Loader:

1. The output template, written in CCOL, which contains the parsing directives themselves. (A praecis of the CCOL follows below.)

2. The CCDB schema, which is available to determine the database target type. The database target type is compared to the type of the textual object given in the parsing directive. The CCDB schema also contains type definitions for type conversion functions and foreign types (i.e., types appearing in the output file that cannot be adequately described with the types nateive to the CCDB and parsing directives).

3. The application output file itself, which contains the text to be parsed.

higher level object or, if there were none, a template for the lower level object as a guide. The parser-converter-loader that we implemented was imbedded into the database schema as input operators. These input operators iterated over a collection by calling the input operator for the lower-level object. Each input operator read the corresponding template.

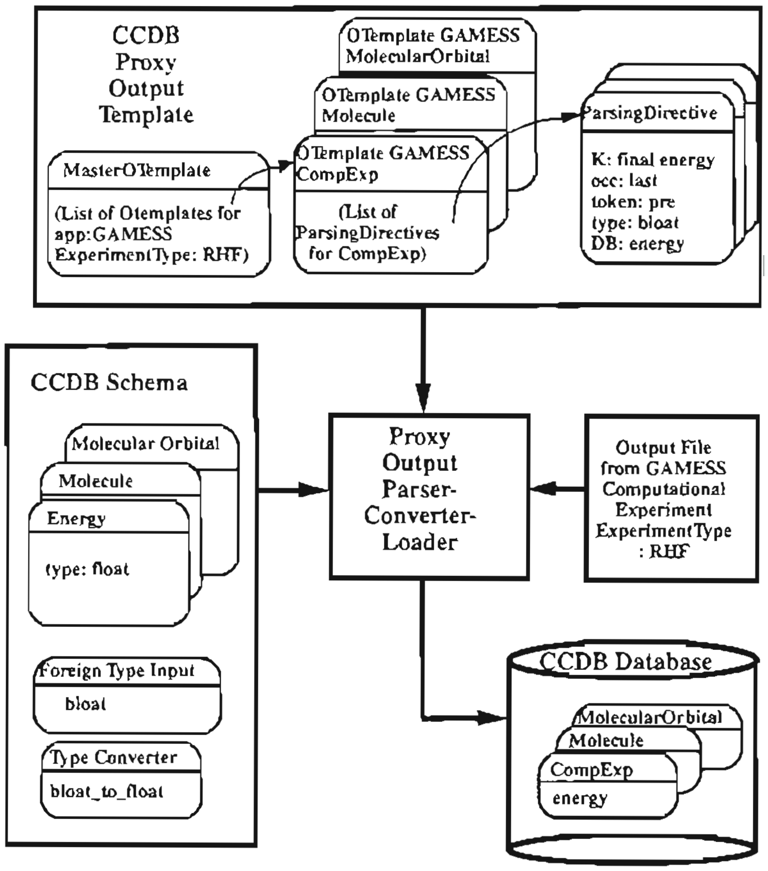

CCDB Proxy Output Template
MasterOTemplate
(List of Otemplates for app:GAMESS ExperimentType: RHF)
OTemplate GAMESS MolecularOrbital
OTemplate GAMESS Molecule
OTemplate GAMESS CompExp
(List of ParsingDirectives for CompExp)
ParsingDirective
K: final energy
occ: last
token: pre
type: bloat
DB: energy
CCDB Schema
Molecular Orbital
Molecule
Energy
type: float
Foreign Type Input
bloat
Type Converter
bloat_to_float
Proxy Output Parser-Converter-Loader
Output File from GAMESS Computational Experiment ExperimentType : RHF
CCDB Database
MolecularOrbital
Molecule
CompExp
energy

Figure 4.12: The computational proxy's output template.

The output of the PCL is the updated database. Once the PCL has updated the database, the proxy notifies the user that the experiment has terminated.

The PCL is always invoked in the context of a particular experiment and expands database references in the output template so that data from the output file for an experiment can be loaded into database objects corresponding to that experiment.

The Computational Chemistry Output Language

The Computational Chemistry Output Language (CCOL) is a layout language designed to define ASCII text (output) file formats for ab initio computational chemistry applications. A sequence of CCOL statements describes a textual object in the output file, and associates it with a database object so that the proxy can load the textual data into the database.

As with the CCIL, the sequences of CCOL statements constituting an output template object could be written directly by the registrar, but we think of the CCOL as an intermediate language to be generated by the registrar user interface. An output template maps a textual object to a database object, implicitly indicating if a type clash occurs and, if so, what conversion function(s) to invoke. For complex objects, such as molecular orbitals, additional layout information for the textual object must be specified in parsing directives.

The language specification for the CCOL follows below. The reader may find it helpful to refer to the example output file templates in Figures 4.13 and 4.14 while reading the CCOL specification.

- A parsing directive is a CCOL statement.
- An output template is a database object constructed from a sequence of parsing directives. Parsing directives may be separated by the new line control character, or by a semi-colon. Each template is applicable to a target database object for a particular application and experiment type.
- A CCOL parsing directive consists of a designator and value pair, or a command.
- A designator is a CCOL reserved word that indicates which application and experiment type a template is associated with, or is a parameter to the parsing process. The designator indicates which parameter to the parsing process (i.e., which parsing action) the corresponding value should be associated with. Parsing actions are associated with either the source (the textual object to be parsed) or the destination (the database object to be loaded). A designator value is separated from the designator by a colon (":"). CCOL designators with corresponding acceptable values are given below:
  - Designators indicating in which master template the template should be placed:
    * Application:<*application*>. This CCOL designator specifies the application to which the template applies. <*Application*> must be an application

already loaded into the database as an instance of application.

    * ExperimentType:<*experiment type*>. This CCOL designator specifies the experiment type to which the template applies. <*Experiment type*> must be known to the database as a valid experiment type for the application that this template describes.

- Designators describing the source:

    * whitespace:<*whitespace*>. The whitespace designator specifies the white space for a given application and experiment type. When searching for keywords and parsing textual objects, the PCL skips application-specific white space such as blanks, "IS", or line-feeds. Special character definitions, such as those for *tab* and *space*, are defined as for the C language [99]. Other white space is given enclosed in quotes. One white space value is given per designator-value pair. Whitespace may be defined for an application, or an application and experiment type, or an application, experiment type and database object.

    * Keyword (or K):"<*keyword*>". The PCL is keyword-driven, and searches the output file until the keyword given in the string <*keyword*> is found. Once its cursor is positioned properly, the PCL will begin parsing the output text. A keyword containing blanks or other white space must be enclosed in quotes, and a null keyword ("") indicates that the next token (ignoring white space) is to be parsed.

    * occ:*first*|*last*. The occurrence designater indicates whether the PCL is to search for the first or last occurrence of the keyword, in situations where a keyword might occur more than once in the text to be parsed. In the absence of an occurrence indicator, the last occurrence is taken.

    * token:*pre*|*post*. The token designator specifies whether the keyword precedes (*pre*) or follows (*post*) the textual object to be parsed.

    * type:<*type*>. The type designator specifies the type of the variable into which the textual object can be read. Predefined types are as per the C

programming language[99] and the application database schema (in our case the Computational Chemistry Database schema). Additional syntactic types may be defined in CCOL by using the DEF command.

- Designator describing the destination:
  - DB:<*database object*>. The DB designator specifies the database object into which the textual object in the output file is to be loaded. As in the CCIL, database objects are referenced by database path names, relative to the computational experiment class.

- A CCOL command may define a textual object, assign a textual object to a database object, or indicate how to reformat a textual object. These are defined below:
  - Defining a textual object (DEF command). A DEF defines a textual object in terms of a predefined type.

```
DEF [<target-type>] <name> [(rowVariable, columnVariable)] [{
    matrix body definition}]
```

  DEF gives a matrix a name, and defines its layout (format). The FoldedMatrix, DeNormalizedMatrix and Matrix CCOL types are appropriate *target-types* in a DEF command and are defined below.

  - Assigning a textual object to a database object. The mapping from the textual object to database object is given by assignment statements, with the database object (or attribute) appearing to the left of the assignment operator (=). Where a series of conversions is required, as in the molecular orbital example below, a series of assignment statements is given. The object to which the assignment occurs need not be predeclared, but if it has not been declared its type must be given in the assignment statement.

```
[Using <object>]
  [<target-type>] <target> = <source>
```

  To make assignments to components of a complex object, the "Using" designator may be employed to specify the complex object from which the components are taken. (This feature is used for parsing a molecular orbital in the example below.)

- Reformatting a textual object. The "StrikeBlanksFrom" command will examine the target textual object and remove all blanks from it.

  ```
  StrikeBlanksFrom <target>
  ```

- Three types are predefined in the CCOL for reformatting matrix data: FoldedMatrix, DeNormalizedMatrix and Matrix. Conceptual descriptions of these textual types are given above; here we show how to define an object as one of these types in the CCOL. All three types have the same components and structure: column headers, row headers and data.

  - A *FoldedMatrix* is a matrix whose columns are not contiguous across horizontal space, and whose row headers and column headers are repeated for each group of matrix columns. A FoldedMatrix is defined in CCOL to have three components: column header (Colhead), row header (Rowhead) and data (Data); each is defined by giving types and names for each element in the header. The variables *nrow* and *ncol* designate the number of rows and columns in the folded matrix; values for *nrow* and *ncol* can be inferred by the PCL from other database values, or from the textual object itself (e.g., two successive line-feeds indicates the last row of a matrix).

    An optional mask (enclosed in single quotes) can define the number of characters expected in a matrix component, and are used when white space might appear in a matrix component. Mask parameters are shown in square brackets below. A mask is a string (enclosed in single quotes) consisting of Xs or 9s, similar to the PL/1 or COBOL PICTURE clause. A mask indicates the number of characters in a matrix component, and whether those characters are alphabetic ('X') or numeric ('9'). For example, the mask '99' denotes a string of exactly two numeric characters, and 'XXX' exactly three alphanumeric characters.

```
FoldedMatrix <name> (nrow, ncol) {
   {Colhead  [<mask>] <type> <name> (1..ncol),
    .
    .
    .}
   {Rowhead
      {[<mask>] <type> <name>,
       .
       .
       .}}
    Data        <type> <name> (1..ncol)};
```

A Foldedmatrix is unfolded by assigning it to a Matrix or DeNormalizedMatrix.

– A *Matrix* is a row by column array of data preceded by any number of column headers (colhead); its columns are contiguous across horizontal space, but its column or row headers might not contain a regular number of tokens. It is defined to the CCOL in the same way that a FoldedMatrix is defined.

– A *DeNormalizedMatrix* is one whose components contain an equal number of tokens per component such that no token has whitespace and all tokens are delimited by whitespace. It is defined to the CCOL in the same way that a FoldedMatrix is defined. A DeNormalizedMatrix can be loaded by the PCL into the database.

When a FoldedMatrix is assigned to a Matrix, or a Matrix to a DeNormalizedMatrix, the target object inherits subcomponent names from the source object. See the example for loading a MolecularOrbital object below.

Type checking between the textual objects or CCOL objects and the database is accomplished by comparing the types of source and target objects. The PCL determines the type of database objects by referring to the database schema. Type conversion functions are supplied either in the database or in the CCOL, and are named as in CCIL.

For example, to parse the final energy textual object in the output file the parser reads the output template instance for energy for the GAMESS application. (See Figure 4.13.) The template indicates the location of the text in the output file (after the final occurrence of the keyword "FINAL ENERGY"), its format (floating point), and where to put

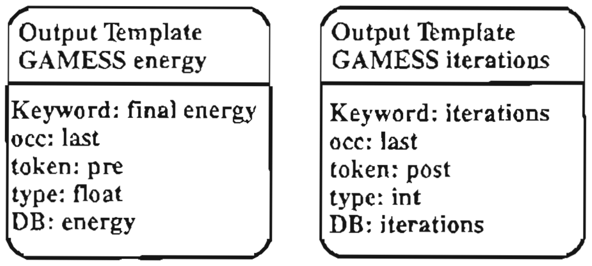


Figure 4.13: Instances of OutputTemplate for energy and iterations.

the value in the database (CompExperiment.energy). The parser searches the output file for the appropriate keyword occurrence and reads the value into transient memory as a floating point object. The parser then loads that floating point value into the CompExperiment.energy database object for the particular computational experiment in context. Because the parser is called in the context of a particular experiment, it knows which experiment in the database to update.

Given the last occurrence in the output file of the text:

```
FINAL ENERGY IS -78.0561311759 AFTER 12 ITERATIONS
```

and the *energy* and *iteration* template instances depicted in Figure 4.13, the interpreter will load the values -78.0561311759 and 12 into the database objects `CompExperiment.energy` and `CompExperiment.iterations` for the experiment in context. The white space "IS" is ignored.

A *data element* in the CCOL is usually in a one-to-one correspondence with a database object. However, for complex textual and database objects such as molecular structure, a data element may refer to components of the textual object, such as atom name. In some cases, a textual object may undergo a series of transformations before being loaded into the database. Thus, for example, data elements needed to describe the molecular orbital textual object include names for the object and its components at each stage in the transformation. The CCOL example below in Figure 4.14 gives the CCOL directives to define a molecular orbital textual object as a *FoldedMatrix*, and to convert it first to a *Matrix* and then to a *DeNormalizedMatrix*. The resulting denormalized matrix is

```
Keyword: "Molecular Orbital Coefficients"
occ:last
token:pre
type:complex
DB:molecularOrbital

DEF FoldedMatrix molOrbTxt(nrow,ncol) {
   {Colhead        int   cols  (1..ncol),
    Colhead        char* slabel(1..ncol),
    Colhead 21'X', float eigen (1..ncol)}
   {Rowhead {
      '99'       int          nrow,
      '999'      int          atom,
      'XX'       AtomicSymbol aname,
      'XXXXXXXX' char*        label}}
   Data float coef(1..ncol)
};

Matrix molOrbMat = molOrbTxt;
StrikeBlanksFrom molOrbMat.label(1..nrow);
DeNormalizedMatrix molOrbDMat = molOrbMat;

USING molOrbDMat{
  'nmo->nMOs'                               = cols(ncol);
  'nmo->symmetry_labels' (1..ncol)          = slabel(1..ncol);
  'nmo->orbital_energy'  (1..ncol)          = eigen(1..ncol);
  'nmo->so->atomicNumber'(1..nrow)          = aname(1..nrow);
  'nmo->so->label'       (1..nrow)          = label(1..nrow);
  'nmo->so->moCoeffs'(1..ncol,1..nrow) = coef (1..ncol,1..nrow)]
```

Figure 4.14: Output template for parsing a molecular orbital.

assigned to the database object MolecularOrbital. An additional type conversions function from AtomicSymbol to AtomicNumber will be invoked automatically, since the database schema not only contains type definitions for *AtomicSymbol* and *AtomicNumber*, but also type conversion functions from one to another. Note that molOrbDMat is a complex data element, containing the data elements "slabel", "eigen", etc. "Using molOrbDMat" allows the user to simplify references to the data elements within molOrbDMat. Thus, in the context of the "Using molOrbDMat" complex statement, the statement

```
''nmo->nMOs = cols(ncol);''
```

really means: "assign the value in the textual object cols(ncol) of molOrbDMat to the database object 'nmo->nMOs' ".

The proxy Parser-Converter-Loader methods process the directives given in Figure 4.14 as follows:

1. During the define phase, the parser identifies and defines the object to be parsed. It searches the text file for the keyword "MolecularOrbitalCoefficients", and extracts the FoldedMatrix molOrbTxt. Variables *nrow* and *ncol* contain the number of rows and columns (respectively) of the original matrix. In the example given here, the value of these variables can be assigned from a value in the database; in other cases, they could be be deduced from the textual object by the parser.

2. During the convert phase, assigning the molOrbTxt object to a Matrix object (molOrbMat) causes the PCL to reformat the textual object from a folded matrix to a matrix object (molOrbMat). The masks ("999..." and "XX...") allow the parser to determine the length of the label field in the row header and hence from where to strike blanks. Once blanks are stricken from the row header, the matrix molOrbMat can be assigned (and hence reformatted) to a denormalized matrix, molOrbDMat.

   Because all three matrix forms share the same matrix components (Colhead, Rowhead and Data), names of components of molOrbTxt (e.g., aname) can be carried forward.

   No type conversions are required at this point.

3. During the load phase, components of the molOrbDMat are assigned to database objects, as directed by the assignment statements. Type conversions supplied by the database are applied at this point, for example AtomicName_to_AtomicNumber.

To summarize, during the define phase, a text file is searched by keyword, and a textual object is extracted. During the convert phase the textual object is reformatted, if necessary, in terms of an easily parsable type such as *DeNormalizedMatrix*, and a CCOL-defined type conversion function may also be invoked. During the load phase, a textual object (or where there was a conversion, an intermediate CCOL object) is loaded into the database, perhaps invoking a database-resident conversion function.

Using The CCOL

When adding a new application to the database, one must describe in the CCOL language the application's output file(s) for each experiment type the application supports. Just as for the CCIL, this involves writing a sequence of CCOL statements for each database object to be loaded and loading these into the database as an output template.

As with the CCIL, there are three ways in which writing CCOL code alone might not be sufficient for generating output files for an application:

1. If the new application's output file require a format conversion that cannot be expressed in the CCOL, a new database method must be added to the database schema.
2. The application schema itself may not be able to represent output of the new application. Here, the schema must be modified.
3. The CCOL may not be adequate to define the output file, in which case the CCIL grammar must be extended.

4.3.5 Mission Control

As shown in Figure 4.15, mission control encapsulates the proxy's network services interface, and is responsible for launching and cancelling experiments. Mission control also notifies the proxy that an experiment has completed and triggers it to capture the application output; we call this last mission control function *mooring an experiment*. In addition, mission control responds to requests about the status of ongoing experiments, such as

- Has my current experiment completed?
- How much CPU time and disk has the current experiment consumed?
- How much clock-time has elapsed since that experiment was launched?
- How close is the computation to converging?
- Did the most recent experiment terminate normally?

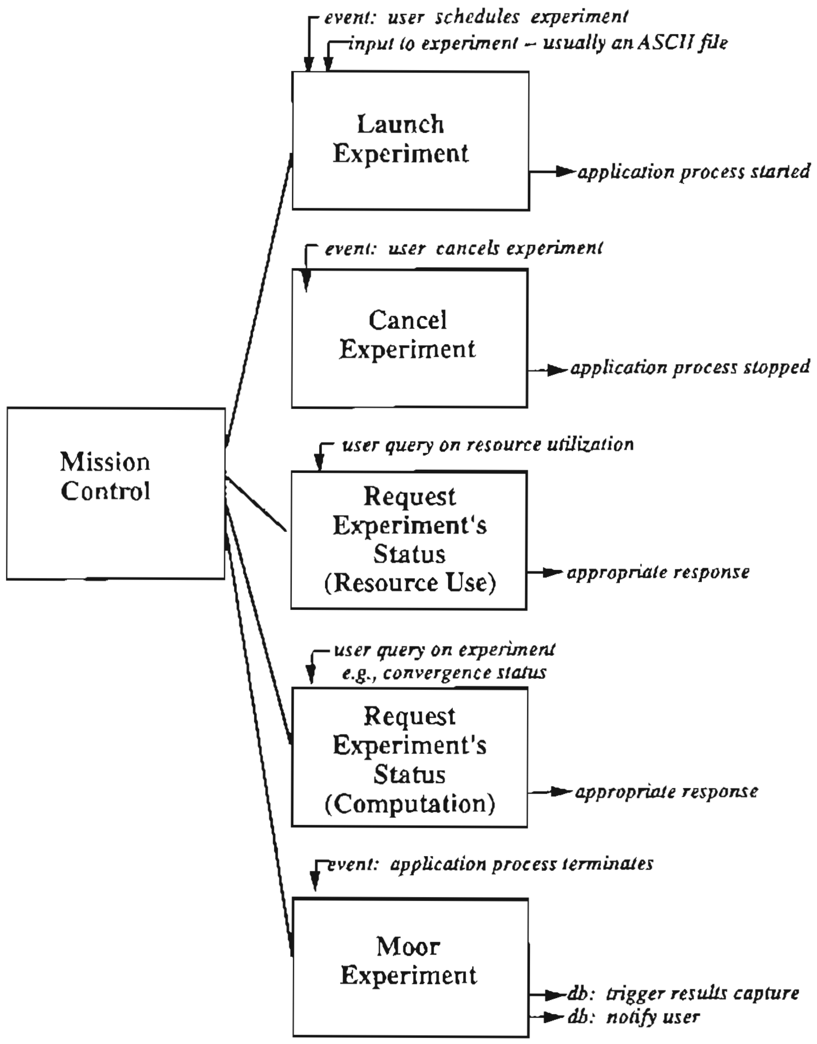

event: user schedules experiment
input to experiment -- usually an ASCII file
Launch Experiment
application process started
event: user cancels experiment
Cancel Experiment
application process stopped
Mission Control
user query on resource utilization
Request Experiment's Status (Resource Use)
appropriate response
user query on experiment e.g., convergence status
Request Experiment's Status (Computation)
appropriate response
event: application process terminates
Moor Experiment
db: trigger results capture
db: notify user

Figure 4.15: Mission control functions needed by the computational proxy.

Some of the queries above involve straightforward requests to the operating system regarding process status. Others, such as how close a computation is to converging, are more difficult to support and require monitoring the semantic state of the computation during execution. Such queries can typically be accommodated by parsing application-specific intermediate files, in a manner analogous to the parsing of output files. Mission control is also responsible for shipping the output file back to the database host.

Both operating system and application-specific information is required to monitor an ongoing experiment. Operating system information is gleaned by the proxy directly from operating system services and is not application-specific. However, information about an experiment that will indicate how close it is to convergence is application-specific and is gathered by the proxy using templates. A monitor template looks like an output template, is written in the CCOL, and maps a textual data object to a database object, defining any required syntactic translation. The textual data objects are found in intermediate files created and updated by the application in the course of execution.

4.4 Computational Proxy Architecture

This section delineates the responsibilities of the proxy mechanism at both the system-level and the user-level.

A conceptual architecture implied by the division of labor that we propose is shown in Figure 4.16. Roughly, the proxy is the locus of experiment control and the connection to operating system services on behalf of the user interface. Network (system) services provide the proxy with the means for shipping files across the network and for starting up processes anywhere in the network. The Database Monitor provides and controls access to the database for the database client process. In our case, the database client process (CCDB Client) is the CCDB User Interface, i.e., the consumer of data from the database. Analogously to how the Database Monitor provides data access by responding to queries on experimental data, the Compute Monitor provides clients with access to programs and files on computers that run computational applications, responding to requests for computational services. (Sample computational requests serviced by the proxy are enumerated

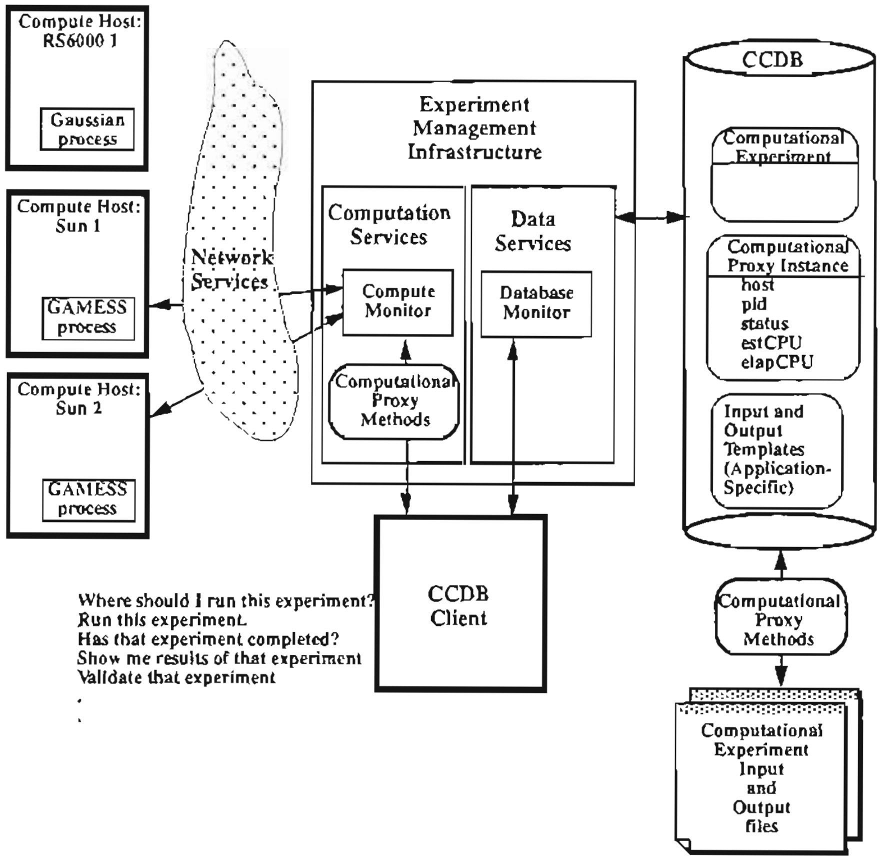

Compute Host: RS6000 1
Gaussian process
Compute Host: Sun 1
GAMESS process
Compute Host: Sun 2
GAMESS process
Network Services
Experiment Management Infrastructure
Computation Services
Compute Monitor
Computational Proxy Methods
Data Services
Database Monitor
CCDB
Computational Experiment
Computational Proxy Instance
host
pid
status
estCPU
elapCPU
Input and Output Templates (Application-Specific)
Computational Proxy Methods
Computational Experiment Input and Output files
CCDB Client
Where should I run this experiment?
Run this experiment.
Has that experiment completed?
Show me results of that experiment
Validate that experiment

Figure 4.16: Compute Server architecture.

in Figure 4.16 to the left of the **CCDB Client** box.)

A computer that runs a computational application is called a *compute host.* The proxy interface to the Compute Monitor consists of database methods linked into the client process; these methods can be invoked only when a database client process is active. The Compute Monitor is part of the proxy architecture, but is a long-lived process, active when the Database Monitor is active and available to service compute requests from computational clients. The Compute Monitor communicates with application processes through network services, communicating either directly with file services on the compute hosts, or with compute daemons running on those machines. One compute daemon process is active per compute host; daemons explicitly start up and control computational processes. Note that compute hosts need not all be the same type of computer; indeed, to be most useful to the scientist, the Compute Monitor and network services should support a number of different platforms.

4.4.1 System-level Requirements — Network Services

We want the proxy mechanism to spawn computational processes anywhere in the user's computational environment. We thus extended a client-server database architecture to include computation services that can operate in a heterogeneous computing environment.

Figure 4.17 illustrates a typical scenario for the use of the proxy mechanism in which a scientist (ClientA) decides to schedule a computational experiment (ExpA). The proxy, using experiment-, application-, and network-specific information, is the logical place for estimating resource requirements and determining (with the user) where to run the application.

Once the proxy knows where to run an experiment, it prepares one or more input files and asks the Compute Monitor to send the files to that compute host along with a message requesting that a new computational application process be spawned. Once the computational process has been started, the Compute Monitor returns the process identifier to the proxy, which updates the database accordingly. ClientA can log off the system any time after scheduling the experiment and later log on as ClientB, requesting the status of the experiment in question. In response to this query, the proxy generates a

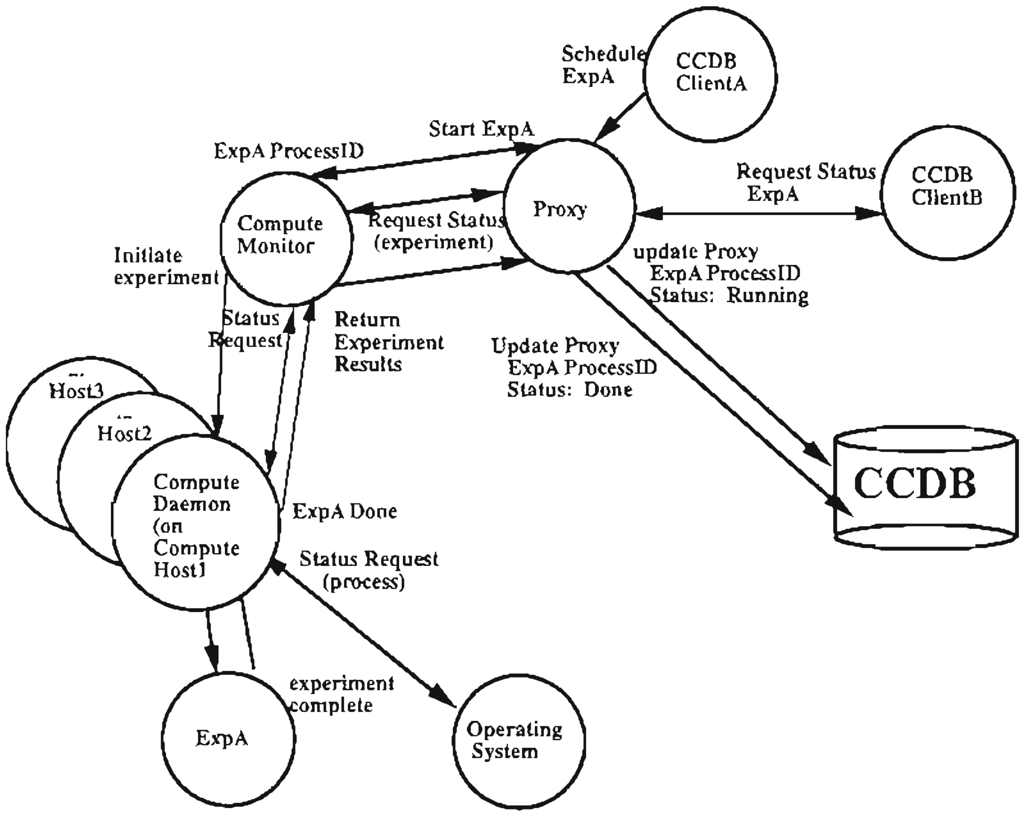


Figure 4.17: A typical scenario for use of the proxy mechanism.

status request message that the monitor sends to the appropriate compute host. The host then queries the operating system for current CPU and disk utilization of that process, which the Compute Monitor relays back to the user via the proxy. When the process terminates, the compute host informs the proxy, which then parses the output file and places the results of the experiment into the database. In addition, metadata about the experiment, such as which computational application was used and the resources consumed, are recorded in the database.

This scenario illustrates how a proxy provides application services within a network of heterogeneous processors, maintaining a consistent view in the database of ongoing application processes. To manage experiments in a network of compute hosts, the proxy must access files across remote machines, as well as start up and monitor programs on those machines. To simplify the user interface to these services, the proxy encapsulates them with a common interface to minimize the architectural complexity. To simplify the development of the proxy mechanism, and to assure that the mechanism be extensible to new platforms, we proposed a division of labor between proxy, i.e., database, and the operating system services.

Accessing files and programs within a heterogeneous network requires a global name space for files and programs, authorization and security for file and program access, a guarantee that the order of message delivery is the same as the order of message sends, and directory services for locating programs and files. An adequate directory service for programs must include the ability to register the its calling sequence (or signature) for a program on the system. The design alternatives open to us for meeting these requirements are to use existing UNIX tools, to use one of the emerging network services, or to implement an interface to network services that is domain-specific to computational science. We argue below that neither of the first two alternatives are sufficient, and go on to specify what features are required to supply network services to computational science applications.

Existing network services generally supply either a remote procedure call (RPC) or network message-passing facility. With RPC only, we would be required to create (fork) an additional process on the same processor as the Compute Monitor, which would then invoke the application process on the chosen processor. With message passing only, the

Compute Monitor would pass an appropriate message to a compute daemon on the chosen processor, which would then create the application process. We used a combination of RPC and message passing to implement the computational proxy for our feasibility study. While RPC and message passing worked reasonably well for our purposes in a distributed but homogeneous architecture, subtle differences in UNIX implementations might make this solution difficult to maintain and extend to multiple architectures. Thus we conclude that UNIX tools such as remote procedure calls are not adequate for the proxy's needs for manipulating remote processes.

Current trends in distributed operating systems and application integration environments suggest that the network services required by the computational proxy will be supported within the next five years [14, 67, 72, 86, 131, 154]. Such "application integration architectures" [86] aim to provide secure network-wide access to programs, files and other network facilities through a common interface. Specific systems propose, and in some cases already provide, a range of solutions to the heterogeneous distributed computing environment problem, from message passing mechanisms [67] or remote procedure calls across heterogeneous architectures to whole systems aimed at producing the illusion of seamless, one-system computing [14].

If application integration architectures such as Parallel Virtual Machine (PVM) [67] and Distributed Computing Environment (DCE) [154] are currently available, why not connect the user interface and the application directly using PVM or DCE? Why bother with the proxy as an intermediary? Tools such as PVM and DCE provide *only* the operating system support necessary to build heterogeneous distributed applications. With PVM or DCE one can write *new programs* that use distributed computing resources. Users of computational applications do not want to write more programs, they want to use existing monolithic programs; they cannot rely on authors of those programs to rewrite their applications using PVM or DCE. Our users need to start up processes that can run independently for long periods of time, accessing those processes periodically to monitor their status. We thus need a tool that will allow us to create *systems* that use the distributed computing resources provided by the CCDB and the monolithic computational applications. PVM or DCE could be used as a basis for implementing the proxy architecture;

but either alone is too low level a tool for our users and we would have to implement additional functionality. DCE, for example, does *not* provide the following features:

- Starting up a remote stand-alone process. DCE requires that the process issuing a remote procedure call remain active until the remote process completes. The proxy is a client process, and should not remain active for days or weeks until a remote process completes.
- Extracting program parameters in the syntax required by the application from the database.
- Parsing program outputs into a common format and loading these into the database.
- Monitoring an ongoing process.

Furthermore, the DCE interface is a complex program-level interface, and the proxy provides a good way to isolate this code from the rest of the system. In addition, at the time we were designing the proxy infrastructure and implementing our prototype, DCE was itself neither fully specified nore implemented. While PVM [67] meets many of our functional requirements for a network infrastructure, we felt that it was not (yet) available on an adequately broad range of architectures to satisfy our desire for portability. As a result of our investigation of general-purpose services such as DCE and PVM, we concluded that these are not widespread enough and operate at too low a level to provide the same functionality as the proxy. Such systems could, however, be used to implement higher level network services that the proxy could then use.

The following commands and features fulfill the proxy's requirements for network services:

1. Remote Start Command (*rstart*). The Compute Monitor must be able to start up a remote standalone application. *Rstart* takes four input parameters: an application designator, a list of resource requirements, input parameters (as per the application signature) and the name(s) of one or more input files. *Rstart* returns a remote process identifier (*rpid*) of the application process.

- The application designator identifies the application either fully or partially. A full designation is equivalent to the system-wide application name, i.e., a network cluster (local area network), processor name, and application. When given a partial designation, i.e., a cluster, or a processor, or a computer type as opposed to a processor name, the network service chooses the processor on which to run the application.
- Any of the following resource requirements may be specified: computational requirements, estimated file space needed, and estimated memory requirements.
- The program signature specifies the calling parameters for the application, since the application calling parameter passed to *rstart* may have to be modified to accommodate peculiarities of the particular computer on which the application is run. For example, all installations will probably require that intermediate files (which are usually large and often accessed) be local, but a particular installation might choose not to use the standard file-name scheme for that application.
- The input file name is usually passed to the application program as a calling parameter.
- The remote process identifier *rpid* is an identifier unique across the network service that will allow the Compute Monitor to query the status of the application process.

2. A global name and directory service for files. Input and output files must be accessible to both the proxy and the application. The proxy and Compute Monitor must access the executable file for each application running on the processor of choice. The signature for the executable files must be specifiable to the directory service.

3. Remote Process Status Command (*rps*). When the user process requests the current status of an experiment, the Compute Monitor forwards an *rps* to the Network Services. *Rps* has one parameter, *rpid*, and returns information about resource utilization in terms of the machine running the experiment. Resource utilization should

be "normalized" to generic terms that can be compared across multiple computer processor types.

4. Remote Done Command (*rdone*). The network service notifies the Compute Monitor when the application process has terminated. *Rdone* is an unsolicited message from the compute daemon to the Compute Monitor and has two parameters: *rpid*, and the same resource information as *rps*.

5. Remote Kill Command (*rkill*). The user process may wish to stop an experiment. *Rkill* has one parameter, *rpid*.

We illustrate the use of the network commands required through the computational proxy by an event trace of an interaction between the proxy and network services, given in Figure 4.18. The processes involved are User, Proxy, Compute Monitor, Network Services and Application; these are categorized as Client, Database or (Possibly) Remote Processes, and so labeled at the bottom. The vertical lines indicate when these processes are active. User and proxy together constitute a *Database Client Process*, and are active only when a user is logged on and until responses from all outstanding events are received. The Compute Monitor (a *Database Process*) and Network Services, on the other hand, are normally active, waiting for requests to service. Of course, either the Compute Monitor or Network Services could be interrupted. The Compute Monitor, like the Database Monitor, must have crash recovery facilities. The Compute Monitor should be able to provide relatively up-to-date information about on-going experiments even if network service is interrupted. The gap in the Network Services event trace indicates a temporary lost of service. Application Processes are active only when invoked. Each network service request is represented by a horizontal arrow (left-to-right); responses are represented by horizontal right-to-left arrows. The process initiating the request or response is at the source of the arrow.

Sections 3.2.4, 4.2.1, and 4.2.2 specified (respectively) information about applications, computer platforms and particular computers (*compute hosts*) for the chemists and the proxy. While this information could all be maintained by hand by the application registrar, the network addresses for and the resource availability on compute hosts would be more

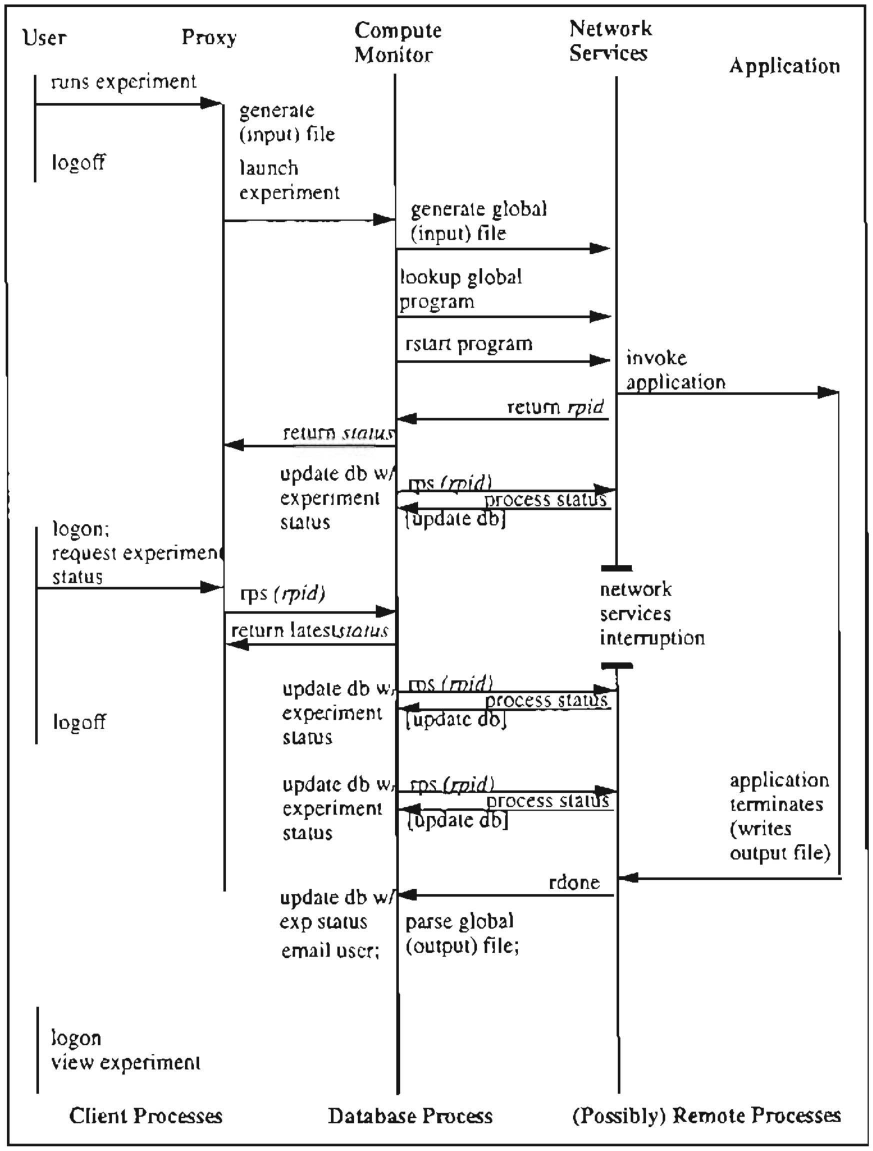


Figure 4.18: Event trace illustrating the proxy's use of network services.

appropriately maintained in near real time by the Compute Monitor using information supplied by the network services.

4.4.2 User-Level Requirements — The User Interface

The proxy provides an application program interface to the system-level services defined above, providing mechanisms and structures for launching, controlling and mooring computational experiments. The proxy does not provide a user-level interface to its services. This section describes a possible graphical user interface (front-end) to the database and proxy.

The primary responsibility of such a front-end should be to provide suggestions for input parameters to computational applications, based on both heuristics and previous experiments stored in the database. Below are additional responsibilities that should be assigned to the user interface from the point of view of the proxy. By this specification of user interface responsibilities we aim to decant the responsibilities of the infrastructure (proxy and database) further than we already have. The user interface should provide for:

1. Browsing the database, providing graphical and tabular views of objects such as molecular structure, molecular orbitals, and tabular views of experiments. Editing features for experiment inputs should also be supplied.

2. Initiating database searches for experiments on molecular structure, molecule name, property type, chemist(s), laboratory and date. Once experiments have been retrieved, the user interface should allow for grouping selected experiments into named (personal) collections and retrieving those collections by name.

3. Annotating both the user's own experiments and "personal copies of" others' experiments. Annotations should be markable private or public. Of course, the database stores annotations so that they can be accessed with their associated experiments, but the user interface must provide a means for the user to annotate experiments and to read annotations by other scientists.

4. Tabulating the resource utilization of previous experiments, and applying rules of thumb to that data to predict resource utilization for a planned experiment. Again,

the proxy's responsibility is providing data; the user interface should aid in interpreting that data.

5. Tabulating previous input parameter values, explicitly or implicitly, and making recommendations with respect to such parameters as basis set, target application, platform and particular computer.

6. Providing for an ongoing graphical "view" of the experiment in process using data transferred by the proxy back to the client process or into the database. This view could include resource utilization such as CPU, memory and disk, or semantic information plotted on every iteration, such as energy gradient.

7. Contacting the user when an experiment has completed, and giving him or her an opportunity to mark the experiment valid or successful. Once an investigation has completed, the user interface should provide a means of marking experiments or groups of experiments archivable, and marking a particular experiment confirmed by another experiment. Of course, if the user has no current active interface at the time the experiment terminates, then the proxy mechanism must contact the user asynchronously. One way to do this is to generate an e-mail message; another is to provide database support for storing and sending such user messages.

Dr. Feller and his associates at Battelle Pacific Northwest Laboratories have already begun work on a front-end to ab initio computational chemistry applications that provides similar functions to those delineated above [51].

The proxy depends on the user interface (1) to assure that an experiment object is complete before a proxy receives a request to run an experiment and (2) to issue all requests for proxy services.

4.5 Chapter Conclusion

This chapter considered design alternatives and defined our infrastructure for computational experiment management, the computational proxy. After showing a conceptual model of the proxy (its structures and behavior), we defined the functional tasks of registering applications, and launching and mooring experiments. The final section delineated the proxy system's responsibilities in greater detail by discussing the interfaces of the proxy with the system-level and user-level components of our architecture, namely the network services and the user interface. In the following two chapters, we describe our prototype implementation of the proxy itself and evaluate the proxy design.

Chapter 5

Implementing the Database and Proxy

The primary goal of our prototype effort was to demonstrate the feasibility of the computational chemistry database and the computational proxy mechanism. Secondary goals were to verify our hypotheses that (1) the model is flexible enough to capture syntactic structures of input and output files for representative applications, (2) an object-oriented database system can is sufficient for implementing the model, (3) the proxy can effectively serve as a locus for experiment management, and (4) the proxy can hide syntactic heterogeneity without requiring special-purpose programming on an application-by-application basis. In addition, building an implementation refined our requirements specification and identified candidate vehicles (platform, language and database management system) for a more extensive implementation.

After an initial feasibility study to identify acceptable database systems and languages [42], we implemented a prototype database and computational proxy system for computational chemistry in C++ and Version 2.0 of ObjectStore, on the Sun SPARC Station 2 platform under SunOS (UNIX). Our implementation effort consisted of two phases: the first to explore running computational applications from the database, and the second to write an output file parser and loader that was driven by our own descriptions of application outputs. We also defined output templates for the GAMESS and Gaussian applications. The computational proxy mechanism itself is implemented as a persistent C++ ObjectStore class. A Compute Monitor and compute daemons were implemented in C and provided the computation services of distributed Sun workstations via standard UNIX socket interfaces.

This chapter first relates, in Section 5.1, the implementation constraints we faced and the high-level implementation choices we made, and goes on to describe simplifying assumptions, implementation issues and alternatives. The prototype implementation for each of the major functional components of the proxy infrastructure is then discussed: the implementation of the database services in Section 5.2, the implementation of input file generation, experiment launching, and output file parsing in Section 5.3, and the method we used to register applications with the proxy in Section 5.4. In Section 5.5 we discuss our implementation of the network services component.

5.1 Implementation Constraints and Choices

The major constraint confronting us during implementation was limited personnel: one full time Ph.D. candidate over the course of about two and a half years and two Master's students part time for about ten months each. Since limited personnel precluded implementing the entire model, we tried to choose features that were most essential to the proxy concept. Thus, for example, we implemented neither a molecular search facility nor polar coordinate representations of molecular structure, because other research and development efforts are addressing these problems [23, 24, 45, 57, 58, 106]. Where we limited the extent of implementation, we made a sincere effort to balance generality with expected volume of use. For example, we limited our choice of applications to GAMESS and Gaussian: GAMESS because it is the application with which our collaborators are most familiar, and Gaussian because we believe it is the most well-known and extensively used. For completeness' sake, we also considered input and output file formats for the MELDF application. In limiting the number of experiment types to five, we chose the first four (RHF with and without Gradient, Direct RHF, and UHF) because they probably represent 90% of the experiments performed at our collaborator's laboratory; we chose the fifth (SDCI) because of the diversity of syntactic structure in its output file.

The major high-level choices influencing the implementation of our system were the decisions to use a commercial object-oriented database management system (DBMS) and an object-oriented programming language along with the Sun architecture and UNIX.

We opted for a commercial product because we felt it would be more reliable and well-documented than a research prototype, and less time-consuming than building a database from scratch. As for our decision to use object-oriented technology, we welcomed the efficiency of directly modeling our constructs, rather than recasting our object-oriented design into a non-object-oriented database system and language. This choice of object-oriented technology entailed the second major constraint we faced, namely working with an immature, albeit powerful and promising, technology. Thus, for example, we did not use such proven productivity tools for compiler generation as LEX and YACC because we wanted to take advantage of the data structures defined in our C++ classes when writing compiler specifications, and we wanted to generate C++ code that used those C++ classes. We could see no easy way to do this with LEX and YACC, or with other lexical analyzers available at the time.

Our preliminary feasibility study led to two viable alternatives for a DBMS: Object-Design's ObjectStore [132] and Servio Logic's GemStone [163]. We chose ObjectStore partially because our collaborators preferred a C++ prototype, and ObjectStore at that time (we felt) provided a better C++ interface. ObjectStore developers have two options for implementation: a C++ library interface to the database system routines and an ObjectStore-specific Data Manipulation Language (DML). The DML consists of several extensions to C++, such as overloading the *new* operator to instantiate a new persistent object and the *foreach* iterator over collections. We chose to use the DML because it is far more intuitive to write and easier to read; however, choosing to use the DML precluded the use of C++ programming environments, primarily because productivity tools such as "smart" compilers or debuggers did not (at least then) recognize ObjectStore's extensions to C++ or handle ObjectStore identifiers.

We chose to implement our prototype for a distributed but homogeneous environment, rather than a heterogeneous environment, because we wanted to simplify the first implementation, and because we believe that the network services described in Chapter 4 will alleviate the need for the proxy to deal explicitly with architectural heterogeneity. We selected the Sun SparcStation 2 as the hardware vehicle for our implementation because it is commonly used by computational chemists, has a good ObjectStore implementation, and is available both in our laboratory and our collaborator's. The choice of UNIX as an operating system was imposed by our selection of the SparcStation, but because we want our system to be as portable as possible, we chose the most general UNIX system features we could for implementing system-level services. Thus, for example, we chose to implement the proxy's message passing facility via UNIX sockets.

Before describing the implementation in more detail, we remind the reader of our major system goals:

- Minimize the need for future schema changes to major domain entities. A common domain-level information model is essential to integrating application programs and data; if this information model is stable then substantive schema changes are rare. We distinguished between changes to the model and changes to the schema: (1) A domain *model* change occurs where an application contains domain-specific semantic information not in the domain model, and (2) A mere *schema* extension may be required when an application contains a syntactic type not handled by the proxy layout or parsing mechanisms.

  The former case (semantic changes) will occur infrequently, and the user community should be consulted extensively before changing the model. The latter case (syntactic changes) merely involves additive changes to the database schema; when the proxy's syntactic transforms are inadequate for describing a new type, that type must be added to the schema along with input or display methods and conversion functions to or from the domain model. Such additive changes to the schema do not require major updates the data in the database (i.e., database migration).

- Minimize the need for special-purpose programmed interfaces to application packages. To that end, our implementation aimed to represent syntactic details of programs declaratively, rather than programmatically. This goal was based on the assumption that, at least for the near term, application developers would have little incentive to modify their applications to read and write common data structures.

  As we saw in Section 4.1, this goal led to our designing an input file generation language and an output file parser.

These two overarching goals guided virtually all remaining implementation choices.

5.2 Implementation of Database Services

In this section, we describe our implementation in ObjectStore's Data Definition Language (an extension of C++) of the domain-specific computational chemistry model. Figure 5.1 shows the entities of the model that we chose to implement. Object types not critical to the computational proxy concept were either not implemented or implemented only partially; these are shaded. For example, we only partially implemented the laboratory experiment object. In addition, the idea of grouping molecules into families by molecular templates, though interesting, is only of marginal interest to the computational proxy project. We used only the Cartesian coordinates representation for molecular structure. The *basis set* class was implemented, but basis set data itself still resides in a file-based library rather than the database; the prototype constructs basis set instances on demand from the existing file-based library. In some cases, we wanted to verify the ability of ObjectStore to perform a particular function that would be useful to the domain, but not critical to the proxy. Thus, for example, we implemented a feature to group experiments into personal sets, but not into suites or investigations, since they are similar to personal sets.

For ease of debugging, we implemented a collection for each major class; in ObjectStore parlance, these are known as *extents*. For each extent we created a persistent variable (database root) that allowed us to directly access the extent classes. Each persistent variable in Figure 5.1 is underlined and an arrow is drawn to the class whose object-identifiers its corresponding class contains.

We used ObjectStore's Schema Designer to generate our first physical schema with classes and relationships between classes, making extensive use of ObjectStore extensions to C++ for defining relationships and collection types. Binary relationships mapped directly into attributes of the ObjectStore DML, and we found the modeling power of the DML sufficient to represent even complex structures such as basis sets and molecular orbitals. For us, one knowledge representation shortcoming of the DML was that only

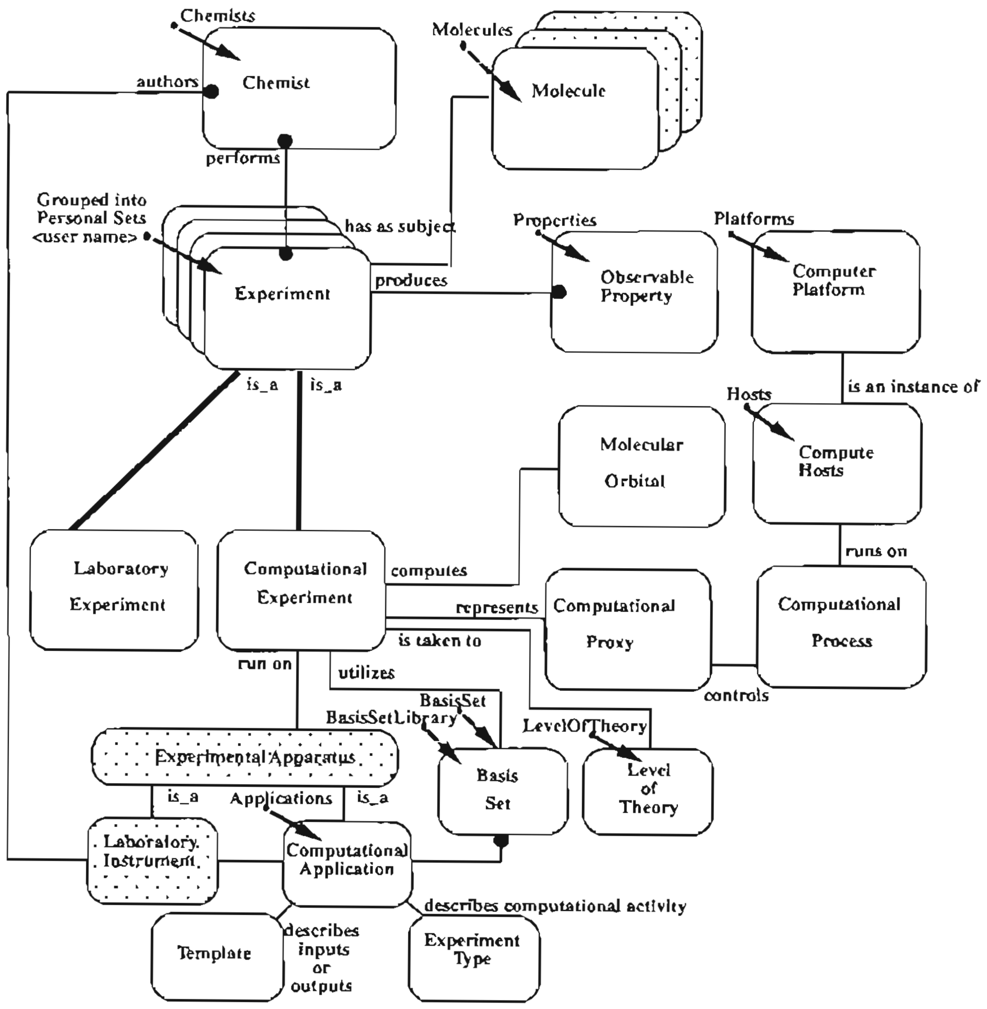


Figure 5.1: Implemented Computational Chemistry Database

class-level, and not instance-level, methods were supported. Thus, for example, we could not include in the computational application class a different method for each instance of a computational application. To get around this problem, we included the name of the function as an attribute of the application class. When we wanted to invoke the method for a particular instance of a class, we passed the name of the instance level function to a class method, and this method called the function.

5.2.1 Populating the Database

To test our schema, we loaded the database with 20 sample experiments. We implemented the database loader as a C++ program that used the C++ iostream libraries to read highly structured text files, copied from a set of representative experiments. The automatic loader made it easy to modify the schema, refine the input data sets, and reload the database. Entering data on an object-by-object basis was sometimes useful, so we also wrote an interactive data entry program. An object editor and bulk loader would be a welcome addition to ObjectStore.

Loading experiment data *post facto* (i.e., after the experiments have been performed) involved loading data from files excerpted from the output files and formatted by hand. Even with this meticulous preparsing, we were faced with the somewhat troublesome issue of value-identity vs. object-identity. ObjectStore, like other object-oriented DBMS, bases its identification of objects on object-identifiers, rather than on attribute values. ObjectStore makes it easy to determine if two references to objects already in the database are "identical" (one simply checks whether the two object-identifiers are equal), but there is no automatic facility for recursively checking value-identity for two objects. These object-identifiers are not known outside the physical DBMS, however, and one must base determinations about identify of a new object with a database object on user-defined value-identity. Therefore, in writing a data loader, one must either invent some value-based identifier to link one object to another (as for a relationship) or physically embed objects within the objects to which they relate (an interesting task for cyclic relationships!). For our initial loader we chose the latter option, but decided in general not to create a new object if a corresponding object with "equal values" already existed in the database.

An exception to this rule is molecular structure; we did not check every atom in every molecule to find out if there was an equal atom at exactly the same x-y-z location.

To determine unequivocally whether the database contains objects value-identical to new objects one would have to write a program for each object that checks each attribute in the candidate instance against each attribute in the existing instance; this is known as "shallow equality". We found, as have others [138], that something akin to the relational discipline of defining unique primary keys as tuple identifiers alleviates the necessity of writing code to ascertain equality. To determine if a referred-to object was "equal-in-value" to an object already in the database, we used either a combination of value-based key values or an an artificially generated primary-key-like attribute (in effect, a serial number). These serial numbers were assigned and inserted manually in the experiment load files. This solution worked for our small prototype database, but would be impractical for a large load file or for incremental loading into the database of experiments. In the course of our research, this object-identity crisis pushed us towards the decision to initiate experiments from the database. Thus, with our infrastructure, the experiment is invoked in the context of the objects of interest, and duplicate object recognition is avoided.

A further (somewhat minor) shortcoming of C++ and ObjectStore was the lack of a dictionary class in either. We responding by implementing our own dictionary-like class to manage the three atomic identifier types: atomic name, atomic number, and atomic abbreviation. The alternative to writing our own dictionary facility was to include enumerated types explicitly as part of the schema. We shunned this approach because of our belief that such dictionary information is data and data belongs in the database, not in the schema.

5.2.2 Implementation of Queries and Database Operations

We chose to implement six database queries and operations, thus verifying the suitability of ObjectStore as a vehicle for further implementation. These operations covered displaying and deleting experiments, and creating new collections. Further details of this implementation are given elsewhere [42, 43].

5.2.3 Summary of Database Implementation Experience

Our experience implementing the computational chemistry database confirmed our hypothesis that object-oriented technology is an appropriate medium for this application area. The lessons learned fall into three categories: the modeling power of the language, the ease with which we accomplished database management tasks, and the usefulness of the programming environment. Of course, our comments about the database implementation are applicable specifically to the development environment we used, ObjectStore and C++. Only in the case of modeling power can we generalize to other object-oriented languages and databases.

The conceptual model mapped easily and intuitively onto object-oriented structures. Object Design's Data Description Language (an extension of C++) and schema designer tool made this mapping somewhat easier than it would have been with the C++ standard class definition structures, because we did not have to explicitly define intermediate classes for aggregates such as sets or lists. Mapping our conceptual model onto another object-oriented language, in particular Smalltalk, might have been somewhat easier than using standard C++, because those languages provide facilities not available in C++ such as predefined dictionary classes and easily accessible instance-level methods. Even so, we certainly found even standard C++ classes more intuitive than the relational tables into which we would have cast our model had we chosen to implement our system as a relational database.

ObjectStore is a "complete" [6] database management system and provided all the major database functions we required: concurrency control, backup and recovery, distributed client-server architecture, and excellent extensions to C++ for data definition and data management. Indeed, we found ObjectStore's methods of mapping persistent memory onto transient memory both transparent and efficient. Other database management tasks where we would have appreciated more functionality include loading and modifying single objects, bulk loading data from ASCII files, browsing the database, and displaying (or printing) database objects. While the ObjectStore Database Browser was helpful in browsing the database and performing simple *ad hoc* queries, the additional functionality of displaying floating point numbers and invoking user-programmed methods from the browser is needed for scientific applications. ObjectStore provided neither a bulk loader nor an object-entry and object-modification facility, and both of these would be helpful in developing a database such as ours. Finally, because scientists need to display complex types such as matrices, one would ultimately like to have a "scientific report writer". While it is perhaps unreasonable to expect such a specialized tool from a general-purpose database system, a complex object renderer or report writer would have been a much appreciated database tool — one from which more specialized tools could be built.

Our implementation experience indicated three shortcomings of the object-oriented database and programming development environment for this application class. We expect these shortcomings to be addressed within two years as more mature products and third-party tools come on the market.

1. Lack of a programming environment. We advocate better integration of database development environments with programming languages and programming environments. The major drawback of using ObjectStore's excellent DML was its incompatibility with third party C++ development environments. Furthermore, even the rudimentary debugging facilities provided with ObjectStore were incompatible with ObjectStore's run-time memory management, e.g., the debugging tool did not expand object-identifiers and thus relationships could not be traversed using the debugger. The lack of a debugging facility was a minor inconvenience for us given the good database browser, but we would have liked to use a programming environment.

2. Lack of abstract data types (or a Class Library) for scientific applications. In time we expect that class libraries for scientific application areas will be developed. For our implementation, we would have liked a matrix type and a molecule type.

3. Lack of a C++ dictionary facility. A dictionary class would have provided us with the facility to easily look up synonyms for database instances, e.g., we could have referred to chemical element by using atomic name, or atomic abbreviation or atomic number. However, C++ does not have a built-in dictionary class, and we had to code a dictionary for periodic table conversions. For future extensions to the prototype,

a class library for unit conversions (probably using a dictionary facility) would be helpful; unit conversions would be useful not only for scientific applications but also for commerce and internationalization efforts.

Overall, we found object-oriented technology an appropriate tool in implementing a complex scientific database. The minor shortcomings we identified (primarily involving needed development tools) will likely be addressed as the technology matures within the next few years. The reader should note, however, that we did not test a number of features that are often cited as requirements for scientific databases, such as the ability to handle large data volumes, or to collect data at high rates [60, 61].

5.3 Implementation of the Computational Proxy

This section describes our implementation of the proxy's encapsulation of the applications' syntactic heterogeneity. The implementation of the proxy prototype actually consisted of three phases:

1. An initial feasibility study to determine if the ObjectStore database management system was adequate for representing the computational chemistry database.

2. A first prototype that generated an input file, shipped it to a remote processor, started up the computational application, performed rudimentary process monitoring upon request, shipped the output file back to the database host, parsed some single value outputs and placed these values in the database. This prototype was application-specific and had no facility for describing input or output files declaratively.

3. An extension of the first prototype that addressed the most difficult problem identified during the prototype effort, namely describing the application output files and parsing them. This constituted a first effort to describe output files descriptively, and hence formed the basis of our specification for application registration.

The latter two phases are discussed here in two sections: the prototype proxy implementation for input file generation (5.3.1) and output file parsing (5.3.2).

Section 5.3.1 outlines the mechanism currently used to generate input files and launch experiments. Because the major objective of the first prototype effort was to determine the feasibility of managing experiments from the database, its major product was the Compute Monitor. The experience gained about input file generation from this effort formed the basis for our definition of the Computational Chemistry Input Language.

Section 5.3.2 describes the effort to moor experiments by parsing application output files. Our experience writing output templates for the GAMESS application indicates that the output template structure is sufficient to cover the three representative applications. As noted in Section 4.3.2, however, the application registrar will be obliged to enter particular template instances for each application. Some template instances will serve several experiment types, but the registrar must specify which templates apply to which experiment types.

5.3.1 Generating Input Files and Launching Experiments

The prototype system contains a rudimentary facility for users to enter information about computational experiments. Information directly relevant to the scientific subject of the experiment is placed into the experiment object; that relevant to the running of the experiment into the proxy object. We hand-coded an input file generation program in C++ that transforms these database objects to the particular textual format required by the GAMESS computational application. This program invokes methods to generate textual representations of molecular structure and basis set for GAMESS. After generating the input file, the proxy sends a message to the Compute Monitor (see Section 5.5) to launch the experiment.

Our implementation efforts led us to conclude that handcoding a data input program for each computational application would be unacceptable in the long run because of the program maintenance required on new releases of the software. Generating input files and invoking applications on remote processors led us to the specification of our input and output file specification languages (the CCIL and CCOL) and of our requirements specification for network services. We also determined that that the proxy methods for generating basis sets for input files would probably have to be particularized by application.

We did not, however, actually implement an interpreter for the CCIL, choosing instead to spend time working on output file parsing which we believed to be a more difficult (though related) problem. We wanted to gain experience with the more difficult of these tasks, and then apply what we learned to both the design of both the CCIL and CCOL grammars, and to their implementations. Our ultimate objective is to make the CCIL and the CCOL as similar as possible.

5.3.2 Output File Parsing

Initially, our prototype shipped output files back to the database host, where a simple PERL script [185] picked out a few single-valued results (such as final energy), and updated the database with those values. Our experience working with PERL gave us the basic understanding of the different applications' output file layouts that led to the specification of the CCOL. This section describes our experience specifying and implementing a more extensive and declarative output file capture facility for the computational proxy. A particular objective of the implementation was to generalize the parsing task to enable a possible next step: building higher-level tools for generating parsers from a descriptive layout language, namely the CCOL.

Implementation Alternatives

We considered two alternatives for implementing the output file parsing facility:

- Break the required task up into subtasks that could be handled by existing tools, and use those to perform the subtasks. Under this scenario, one might use an existing UNIX tool, such as PERL, or a specialized text processing language, such as SNOBOL, to parse and reformat the output so that it could be read by the proxy and captured in the database.
- Write a single system that integrated all required parsing and loading tasks as part of the computational proxy.

The existing tool we explored was PERL. Though satisfactory for single-valued outputs, we found writing the PERL scripts for complex objects such as molecular orbitals somewhat daunting. We also saw no direct way to specify the correspondence between data objects (output by PERL) and database objects, and therefore, we would have had to define a separate language to link PERL output with database objects.[1] Furthermore, we wanted a solution that would eventually incorporate a single parsing mechanism, rather than having to deal with several languages. Finally, we felt that existing tools such as PERL do not interface cleanly with database systems; database methods in C++ would have to invoke the PERL scripts and load the objects parsed. We felt this constituted a cognitive clash ("impedance mismatch") for the programmer [120]. As a result, we decided against using existing tools such as PERL to implement the parser, and instead chose to implement output file parsing directly as part of the proxy itself.

We thus chose the second alternative because we believed that using a single programming language would help us better generalize the parsing problem so that we could eventually design and implement a more general tool for parsing scientific text. One way to implement the parser would have been to implement a compiler to read output templates and generate parsers (as database methods) for each application and experiment type. Then, when the proxy was invoked to parse an output file, the database methods would be used to parse that output file and load the data into the database. We opted to write an interpreter rather than a compiler, however, because we wanted the database methods that parsed output to be generated in C++ and there were as yet few proven tools for C++ compiler generators. Furthermore, before investing the effort required for writing a compiler we wanted to experiment with both the output-file parsing problem and our output-file description language. We felt that writing an interpreter rather than a compiler was a more efficient way to experiment. Thus, we decided to integrate the parser directly with the proxy rather than use existing tools, and to implement an interpreter rather than a compiler.

[1]We did not investigate writing a single language that would generate PERL scripts, and make the necessary correspondences with the database objects.

Design of the Output Template Interpreter

When an experiment has completed, the Compute Monitor informs the proxy mechanism, which invokes the output file interpreter. The interpreter reads a list of output template objects specifically for the computational application and experiment type of that experiment, one template per database object to be loaded. The specification for the interpreter is driven by the structure of output templates and is described in Section 4.3.4.

We decided that having the interpreter read output template objects rather than directly process template statements written in the CCOL would simplify the interpreter design. We thus eliminated the need for the interpreter itself to include a CCOL lexical analyzer and syntax checker by translating CCOL definitions into template objects. The interpreter that we implemented assumes an intermediate step that translates the output file description in the CCOL into output templates, one per object to be read by the proxy mechanism. This step is now only semi-automated; our registration procedure currently consists of writing a text file containing parsing directives for each destination database object for each application and experiment type. A program then loads these directives into the database as output templates.

For each database object to be loaded, the interpreter searches the output file for the keyword given in the output template, then carries out the template's parsing directives. The interpreter first delimits the textual object from its surrounding text, and then reads the textual object into memory as per the type given in the template, comparing its type to that of the target database object. If the types clash, the interpreter applies a simple heuristic to determine the appropriate type conversion function. Where appropriate, standard C++ type conversion functions are used; where not appropriate, the class names of the textual object and database object are concatenated to yield a data conversion function.

In the initial parsing phase, the interpreter works much like a lexical analyzer, moving a cursor through the text and picking up or discarding tokens as directed.

Implementation of the Interpreter

The "Parser-Converter-Loader" (PCL) fulfills the proxy's application output-file parsing and loading functions, and is in effect an interpreter for output templates. This section recounts our experience implementing the PCL. We outline the challenge and strategy of this implementation, and describe its high-level program design. We also address to what extent the PCL implements the system specification for the output-file parsing component of the proxy outlined in Chapter 4 and what steps are now required for a registrar to arrange for output files for a particular application to be parsed by the PCL.

The major design challenge we faced was how to load text from the file that was different in type from its target database object. Even though we knew the type of the textual object from the output template, C++'s dynamic run-time support was not flexible enough to provide an easy solution to this problem. A secondary challenge was implementing the complicated syntactic transformations as for folded matrices.

The strategy we used to meet our primary challenge was to overload the C++ input operator (>>) for the experiment object, each object in the experiment class hierarchy that might be an output of a computational application, and each type encountered in the output file. By *experiment class hierarchy*, we mean every class related to the experiment class that is a potential application output. About sixty operator functions were written. Once a computational application has terminated and the output file is available for parsing, proxy tells the experiment object to "read itself in". In the course of reading itself in, the experiment object traverses its own hierarchy depth-first, sub-object by sub-object. The experiment object hierarchy is traversed thrice: once to parse the output file and load the textual objects into memory, then again to convert those objects, and finally to actually load the objects into the database. The conversion phase can consist of syntactic conversion (as for a folded matrix to an unfolded denormalized matrix) or a semantic conversion (as from one database type to another). Note that the PCL implementation thus belies the parsing order defined by the master template in the PCL specification; this is a result of the implementation decision to overload the input operators for parsed objects and start the parsing operation by telling the experiment object to "read itself in".

Each sub-object in the experiment hierarchy uses PCL methods to "read itself in". PCL methods are available to the sub-objects because each object liable to be parsed, converted and loaded is also a sub-class of "parsable object". The *parsable object* is an abstract super-class that contains methods for interpreting output templates. To determine if it is to "load itself in", a parsable object determines whether there is an output template object corresponding to itself, for the application and experiment type for its corresponding experiment. (A PCL method is available to do this task.) If an output template object corresponds to the database object, the parsing directives in the output template are followed and that object is loaded. Eighteen directives were implemented.

For each experiment object to be read in, the PCL completes the following actions:

1. From the output template, determine the type of the textual object to be read, and allocate the required storage by creating a new C++ object to hold the object as it is parsed and converted, but before it is loaded into the database. For the purposes of generality, a temporary object is created even if there is no conversion required and the object could be directly read into the experiment object.

2. Read the textual textual object into memory by processing the parsing directives for that object. Parsing the textual object involves positioning the cursor in the output file to that object and performing syntactic transformations (such as reformatting a folded matrix to an unfolded, denormalized matrix). Syntactic conversions done at this step are only those that can be described using the CCOL.

3. Perform (semantic) type conversions (if needed) and move the data into persistent storage, i.e., into the database. The PCL determines the name of the conversion method to use by concatenating the string that names the type of the object read in, to the string "_to_", to the string that names the type of the database object. Thus, for example, the method "bloat_to_float" would be invoked to convert a textual object of type "bloat" to a database object of type "float". The type of the textual object is known from the output template.

   In addition to the path for the database object, the class name of the object to be loaded is found in the output template. Our design calls for the PCL to use ObjectStore's MetaObjectProtocol (MOP) to look up the type of database object. Note that looking up the type of an object in the MOP does not require that there be an instance of the class whose type we request. The MOP is essentially a database that holds ObjectStore schema; types can be looked up using class names. Thus, one can use the MOP to determine the object's type even if one might want to later create the object as a result of parsing it. (For the prototype, we did not implement an interface to the MOP. Instead, we implemented virtual functions that returned the type of the database object for each database object.)

The strategy of rewriting the input operator for objects in this hierarchy and deriving all experiment objects from a single base class ("parsable object") has several advantages:

1. Rewriting the C++ input operators makes for a clean connection between the database schema and output templates. Because loading is accomplished in the context of the experiment object, that object's methods are available to the PCL.

2. We could override the input operator for objects in the experiment class hierarchy via the virtual member function, and get around the difficulties in C++ of dynamic typing.

Its primary disadvantages are:

1. We had to write classes and input operators for each type of textual object (even standard C++ types such as "string" or "int"). Writing these classes was easy, but tedious and error-prone (and perhaps could be automated).

2. The order of parsing is controlled by the order of traversal of sub-objects in the experiment object. Thus, our PCL may make several parsing passes through the output file, even though the registrar initially ordered the CCOL statements according to the order of textual objects in the output file.

The PCL Implementation — Discussion

To determine how well our implementation of PCL matches the specification given in Chapter 4, we examined the connection between the CCOL and the parsing directives. The CCOL described in Section 4.3.4 is a relatively high-level language appropriate for use by the registrar. Our specification calls for a CCOL compiler that takes CCOL statements (as described) and writes output templates that can be processed by our interpreter. CCOL statements are currently manually processed as follows:

1. CCOL statements referring to a particular experiment object, application and run type are placed into a single file. Semi-colons are removed.
2. CCOL statements that unfold and denormalize matrices are translated into a simplified language (see below).
3. A program loads these files into a list ordered by application, experiment type, and experiment object, thus creating output template objects.

The current PCL directly interprets CCOL statements to load objects that require no extensive syntactic transformations (such as final energy or molecular structure).

For complicated syntactic transformations needed for such textual objects as folded matrices, however, CCOL statements must be converted into a simpler set of directives. Recall that, in the case of parsing and loading a molecular orbital, CCOL statements describe the format of a FoldedMatrix, and give directives to transform the folded matrix to a matrix, strike blanks from the label field in that matrix, transform the matrix to a denormalized matrix, and finally load components of the denormalized matrix into the database.

Before PCL can process CCOL statements, they must be converted (or compiled) into directives in the output template. We currently accomplish this conversion by hand. The following scenario indicates how to convert CCOL statements:

1. The K:"Molecular Orbital Coefficients" directive positions the PCL cursor in the correct location in the output file. There is no conversion needed.

2. The CCOL declaration of the FoldedMatrix object triggers the reading of the folded matrix.

3. The CCOL statement assigning the folded matrix to a matrix is replaced by a directive "Unfold Matrix". Because the PCL cannot understand the masks in this declaration, the number of columns and rows in the folded matrix must be communicated to the PCL. The unfold matrix directive causes the PCL to create a new object of type matrix and to read an unfolded matrix into that object. The directive UnfoldMatrix has four integer parameters: number of rows, number of columns, number of column headers and the length (in ASCII characters) of the row header. The length of the row header must be constant across rows.

4. The CCOL statement assigning the matrix to a denormalized matrix is replaced by a directive "DenormalizeMatrix". This directive causes the PCL to follow sub-directives (explained below) that reformat the matrix so that a constant number of tokens appear in each row. The directive DenormalizeMatrix has two integer parameters: number of rows and number of columns. The sub-directives needed to accomplish this task are "move from above if blank" and "Move from right if blank". These directives each take two integer parameters, start and length.

   - "move from above if blank" is used to carry forward text from one row header to the next. Thus, the directive "move from above if blank 3,3" causes the PCL to rewrite the following two row headers, moving three characters ("1 H") from the third position in line 1 to the third position in line 2.

     ```
       1 1 H 1S
       2     2S (I)
     as:
       1 1 H 1S
       2 1 H 2S (I)
     ```

   - "move from right if blank" is used to remove embedded blanks from a row header. The directive "move from right if blank 7,5" causes the PCL to rewrite the following row header, removing blanks from the string of length five that begins at the seventh position:

```
2 1 H 2S (I)
```

as:

```
2 1 H 2S(I)
```

For further details on how to transform the CCOL to directives of this form, and for further details on the implementation of the PCL, see Donald Abel's Master's Thesis [1].

Adding Output Templates to the Database

For the registrar to add a new application or experiment type to the proxy, he or she must first create the list of output templates for that application and experiment type. Then, for each output template in this list, the registrar must complete the following tasks:

1. Determine which experiment object in the database schema corresponds to the textual object. This is the database object that is the target of the load. Its name is used in the output template.

2. If the textual object is of a semantic type new to the database, the registrar must add a class of the new type to the schema. For example, if the target database object is molecular structure in Cartesian coordinates (AtomCart) and the textual object is molecular structure in polar coordinates, a new class for atomic structure in polar coordinates (AtomPol) must be added. In addition, a conversion routine must be written and added to the schema for the class to which the data is being converted (here, "AtomPol_to_AtomCart" to convert from polar to Cartesian coordinates).

3. The registrar must write the output template for the new object in the CCOL (if there is not already one in the database). Even if there is already such an output template, the output template list for that application and experiment type must be updated to point it.

Our most difficult C++ implementation tasks were: (1) allowing for textual objects of arbitrarily large size, (2) reformatting ill-formed textual objects, such as matrices containing molecular orbitals, into well-formed matrices, and (3) adjusting for limitations of the C++ language and ObjectStore.

To allow for textual objects of arbitrarily large size was a tedious but straightforward programming task. We simply allocated memory for buffers as we read the input file.

Writing programs to parse well-formed matrices was a relatively simple task, given each like row of text has the same number of tokens. However, writing programs as specified in Chapter 4 to read and reformat folded and unfolded matrices would have involved implementing a lexical analyzer that could act on positional cues, recognizing masks for textual objects such as 'XXXXX' as "a numeric character field of length five, some of which may be spaces", and directives to "strike blanks from" those textual objects. Rather than do this, we chose to design new directives that could be more easily interpreted, and then to rewrite the CCOL statements into those directives. For now, this rewrite task is carried out by hand.

C++ and ObjectStore Limitations Encountered

The three limitations we encountered in the implementation of the PCL were: lack of dynamic typing in C++, lack of instance-level methods in C++, and lack of support for ternary relationships in ObjectStore.

We found it very awkward to program in C++ the dynamic typing we needed to read in textual objects whose types were unknown at compile time. Even though the templates tell the PCL input operator to expect an object with a type different from itself, C++'s static typing makes it awkward to dynamically recast that object into the appropriate type. For example, assume that the textual object of *TotalEnergy* for the Gaussian output file must be read into memory as an object of type *bloat*, but that the type of the corresponding *TotalEnergy* database object is *float*. Assume further that no suitable automatic C++ type cast from *bloat* to *float* exists. C++ left us with two obvious alternatives for where the conversion should be done:

1. Read the entire experiment object, and then perform all conversions as the experiment is loaded into the database. Under this scenario, all textual objects for one experiment are read into a schema specialized for that application and experiment type, e.g., one that includes the *total energy* as a *bloat* type. The type conversion is then performed when the parser loads the object into the database schema. The advantage of this approach is its simplicity; the disadvantage is schema proliferation, which would eventually have to be automated. For technical reasons, a schema specialized for the entire experiment object is required, even if there is only one type clash at a leaf node.

2. Perform the conversion as the textual object is being read. Under this scenario, the interpreter reads an object as a text string and flags that object with its type. After comparing types for the textual and database objects, the parser directly loads the object if the types match. If the types do not match, the parser applies the appropriate type conversion function. For this approach, the only schema modification is that new types must be defined as C++ objects, and conversion functions to any target types must be supplied. The implementation challenge arises from C++'s inflexible dynamic typing capability: The interpreter must intercept an input operator's reading of an object to perform the type check and pass the textual object through a conversion routine if there is a type clash.

Our solution to the dynamic typing problem used aspects of each of these alternatives. As detailed above, we derived all objects in the experiment hierarchy from a single base class "parsable object", and intercepted the input operator's reading of an object. Conversion of foreign types to database types occurred after all objects from the output file had been read in.

We also would have appreciated support for instance-level methods. By an instance-level method we mean the ability to create a method associated with a particular instance of a class. In our system, application-specific object display, input and convert functions are essentially instance-level methods of templates. We would like to store different display, input and convert methods for each template object. Unfortunately, C++ and ObjectStore provide only for methods at the class level. We experimented with writing instance-level methods ourselves by defining function-valued attributes for some display, input and convert functions.

When we needed to invoke these "instance-level" methods, we dynamically "linked" the appropriate function to the class by passing the name of the function to be called to a generic class method at the class-level for display, input, or convert. The generic method then invoked the method whose name is passed to it; for a discussion of a similar technique, see Coplien's *Idioms and Paradigms* [38]. Thus, for example, to generate application- and experiment-specific basis set instances, we provided appropriate conversion functions and a table with names of those conversion functions and calls to each. The name of the appropriate conversion function could be deduced from the name of the application, and we simply searched the table of conversion function names and made the appropriate call.

The attentive reader will observe that we could have avoided this problem by defining more classes; however, the number of classes needed would have been very large – one per template – and significantly complicated our schema. A better solution is for the ObjectStore C++ preprocessor to provide handles for instance-level methods that could be dynamically linked with class instances at runtime.

During our implementation of the proxy we also confronted a few limitations of the ObjectStore database management system, the most bothersome of which was the lack of support for ternary relationships. Given an experiment object, we must determine which template to use to parse that experiment's output file. Unfortunately, the ternary relationship among a template, experiment type and application (see Figure 5.2) cannot be directly represented in ObjectStore. The cardinality of this relationship is that there are many templates per experiment type and application — one template for each experiment attribute or experiment subclass attribute to be parsed and loaded. To deal with this issue, we defined a new entity, ExpTypeForApp to represent the binary many-to-many relationship between experiment type and application. To provide the interpreter with an effective way of accessing output templates, we defined entities to group output templates for a particular application and for a particular experiment type and application into lists. The latter list is ordered as per the order in which an interpreter would most efficiently

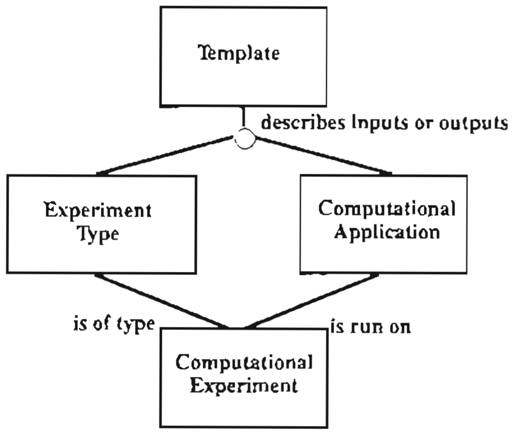


Figure 5.2: Ternary relationship among Template, Experiment Type and Application.

load experiment attributes and subclasses from the output file. To maintain integrity constraints, one must verify that there is exactly one template per experiment object for each experiment type and application when loading templates into the database.

Figure 5.3 shows our implementation of the above ternary relationships. The three shaded boxes, TemplatesForExperimentTypeAndApplication, TemplatesForApplication, and ExpTypeForApp, represent the classes needed to implement the relationships among the template, application, experiment type, and experiment classes. Thus, for example, to access the list of output templates needed to parse an output file for an experiment of type "energy" run on GAMESS, we would navigate from the experiment object, to a corresponding ExpTypeForApp object, and from there to the ListOfTemplatesForExperimentTypeAndApplication object. This list would then direct us to each template needed to parse the output file.

While we attribute this limitation to ObjectStore, one should note that the limitation is effectively that of the C++ language itself. The ObjectStore DML has provided effective enhancements to the C++ class structure to model binary relationships, and to enforce referential integrity and cardinality constraints.

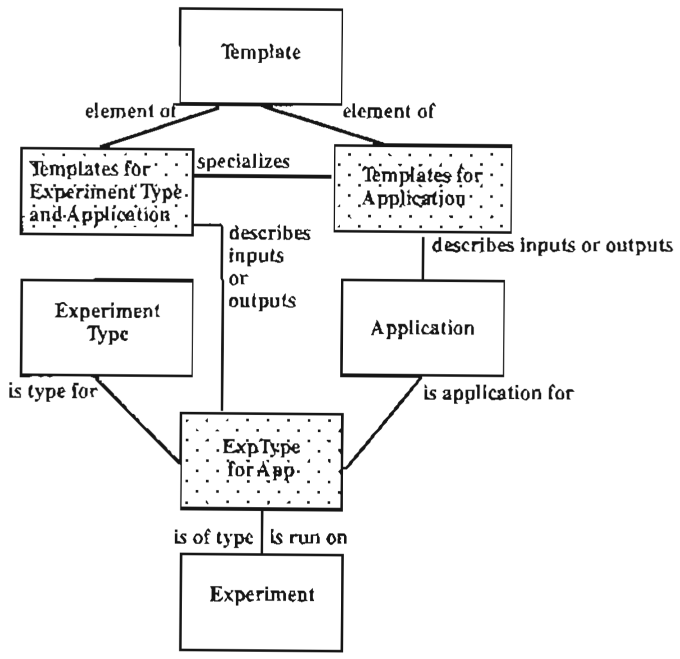


Figure 5.3: Implemented ternary relationship.

5.3.3 Summary and Conclusions of Proxy Implementation Experience

Our implementation of the proxy prototype demonstrated that the proxy mechanism is feasible using existing technology. However, the C++ language itself has certain characteristics that make that implementation difficult. In particular, object-oriented database systems using C++ as their data modeling and application programming language would benefit from additional modeling features such as ternary relationships and instance-level methods. We also found it awkward to write C++ class methods that would allow inputting types different from those of the class itself.

We also consider that the amount of time required to write the general-purpose code to parse the output files (approximately four person-months) is cost effective.[2] This assumes, on the basis of our experience, that it would take approximately two months to write a special-purpose program to parse the output of the first application, and one month for each application thereafter. It also assumes that writing and testing an output template takes less than one week. These estimates, contrasting the difference in number of weeks required to set up a proxy with output templates with the time required to set up a proxy that uses special-purpose parsing, are tabulated below:

	w/output templates	w/special-purpose parsing	total wks compared
1 app	4 months + 1 week	2 months	17 v 8
2 apps	+ 1 week	1 month	18 v 12
3 apps	+ 1 week	1 month	19 v 16
4 apps	+ 1 week	1 month	20 v 20
5 apps	+ 1 week	1 month	21 v 24
6 apps	+ 1 week	1 month	22 v 28

Given the estimates above, the break-even point for using output templates over special-purpose parsing is four applications. This analysis does not take into account the software maintenance that would be required with the special-purpose parsing option — each new version of each application may include changes in the output files, thus necessitating changes in the code written to parse the output files. Making changes to

[2]By "general-purpose code" we mean the time required to write the parser that interprets the output templates.

a parsing program is more expensive and error-prone than making changes to an output template.

We do not have adequate data to predict the cost-effectiveness of input templates, though we suspect it would be similar.

5.4 Application Registration

This section shows how we entered application information into the database. We did not implement an automated application registration facility for the proxy. Instead, the procedure we used for registering an application to the prototype system involved using a text editor to write information about the application into text files, and then loading the application and template objects from those text files. We designed and implemented programs to read and load into the database application objects and output templates. Thus, in effect, we wrote a bulk loader for the application and template classes. Adding a new application includes the following tasks:

1. Editing a file of *Application* instances to include the new application, and then running the program to load the database with application instances. For example, the following lines were added to the file to load the application object instance for GAMESS:

```
name            GAMESS
version         1
computer        Sun4
fmtBSIFcn       fmtBSIGAM
dateAvailable   2/19/92
timeAvailable   12:00:0
dateArchived    0/0/0
timeArchived    0:0:0
maxL            2
spherical       false
maxS            30
maxP            30
maxD            30
maxF            0
maxG            0
```

2. Making the program modifications necessary to build the input file for that application. This process has four steps:

   (a) Writing a program to generate an input file for the application, alter the calling sequence of the program that invokes computational applications to call this new program when the application is invoked. Compile and link the new program with the computational proxy system.

   (b) Providing a formatting function for basis set instances as required by the application program. The new function should be given a name fmtBSIxxx, where "xxx" is the first three letters of the application name. Thus, for example, the function fmtBSIGAM formats the basis set instances for the GAMESS application.

   (c) Modifying the user query in the user interface to verify specifically if that application should be used for the experiment about to be run.

   (d) Modifying the program (in the Compute Daemon) that actually calls the computational application with the calling sequence of the new application, and the names of work files that the application expects.

3. Editing the file of *OutputTemplate* instances to include template objects for each experiment type, and then running the program to load the database with those instances. For the ConvRHF experiment type of the GAMESS application, the following lines were added to the file:

```
GAMESS
ConvRHF
*CompProxy
   K "number of basis functions", F0, PreT, unsigned short, "nfcns"
*CompExp
   K "final energy", L0, PreT, float, "energy"
   K "cpu time", L0, PreT, float, "cpuTime"
   K "words of dynamic memory", L0, PstT, long integer, "memory"
*Molecule
  skip past "coordinates of all atoms"
  skip past "--+"
```

```
*Atom
  K "", FO, PreT,string,"AtomicNumber"
  K "", FO, PreT,unsigned short,"charge"
  K "", FO, PreT,float,"x"
  K "", FO, PreT,float,"y"
  K "", FO, PreT,float,"z"
```

While this registration procedure worked well enough for the purposes of our prototype, it would probably not be adequate for end users. Setting up the invocation of the new application, in particular, needs refinement. Section 4.3.2 has described the application registration procedure our final design requires.

5.5 Implementation of Network Services: The Compute Monitor

Because computational packages run on a variety of platforms, proxies encapsulate operating system and architectural complexity as well as the syntactic complexity at the application level. This section describes our prototype implementation of the network services required to allow the proxy to hide the distributed nature of the computing facilities in the computational laboratory. We use the term *client process* to refer to the process running the computational scientist's user interface; the client process requests database services from the computational chemistry database and computational services of the computational proxy.

In Section 4.4 we showed how the proxy enables the user interface to request computation services from a compute monitor much as it now requests objects from the database. We also described the three functional requirements of the network services required by the proxy architecture, i.e., to access files across remote machines, to start up programs on remote machines, and to monitor the progress of those programs. We then went on in Section 4.4.1 to specify five functional requirements for network service commands and services:

- A global name and directory service for files.
- Remote Start (*rstart*).

- Remote Process Status (*rps*).
- Remote Done (*rdone*).
- Remote Kill (*rkill*).

Below we describe our implementation of these network services. We chose to implement the network only for distributed *non-heterogeneous* computing resources because of the current complexity of implementing these services for different platforms.[3] We believe that the services we require will likely be available in the next five years, on an adequate range of processors, through distributed operating system facilities. We wanted to implement our own version of these services now, however, to verify our requirements analysis and provide a realistic computing environment for the computational scientists testing the prototype.

We implemented the network services component of the computational proxy on Sun4 workstations. Because our code implements the service at a very low level (UNIX system calls and socket-level communication), and separates whenever possible platform-specific and platform-general code, it is extensible to other platforms than Sun and could be used to provide computation services for a working scientist.

5.5.1 Design Objectives

Our major objective for the conceptual design of the network services was a clean separation of responsibilities among the client process, the database and network services, and the application(s). A clear demarcation is important because the network services are likely to be more platform-dependent than database or proxy functions and, if implemented separately, can be extended to other platforms more easily. Furthermore, as distributed operating systems facilities provide these services, this part of the prototype implementation can be easily replaced. Finally, a clean separation of responsibility encourages a

[3]Note that implementing the proxy's network service commands on top of the Parallel Virtual Machine (PVM) [67] would have rendered the proxy ipso facto operational for *non-heterogeneous* computing resources. However, we chose not to use PVM because (1) our users would have had to install PVM on each machine running ab initio applications as well as on the database machine, and (2) PVM was not, at least then, adequately widespread.

cleaner separation of policy decisions from mechanism by isolating the user-level, domain-specific decisions in the proxy. Separating policy from mechanism is important so that policy can be more easily changed to support different domains and user requirements. In principle one should be able to improve mechanism to support better performance without affecting policy.

The proxy is the locus of experiment control for the user as well as the interface to the network services, to which it issues directives and supplies application-level information. Because the network services facility has access to resource availability information, it should supply that information to the proxy and the user, as the basis for deciding where to run the experiment.

One critical design requirement is that the network services must allow the client process and the proxy to detach from the computation. If the computation cannot be completed while the client process waits for the response, then the client process must be able to disconnect from the system, returning later to check on its progress; the proxy must be able to start up an application on a remote processor and not have to wait for it to complete. This requirement is analogous to the need for data to reside in a database without having to have a process remain active for the life of the data. As we shall see below, this requirement eliminates otherwise acceptable implementation alternatives.

5.5.2 Implementation Alternatives

Our alternatives for supplying network services were to write our own from scratch or to use an existing network-based computing environment. As pointed out by Geist and Sunderam [67], network-based computing environments fall into two categories, those based on a specific distributed operating system [144] or programming paradigm [5], and those that provide general-purpose computing environments. The former category is inconsistent with the proxy objectives of providing computation services across a range of computing hosts, operating systems and architectures. Distributed computing systems in the latter category typically provide networked computing environments with the following characteristics: procedure-call access to system facilities, support for local inter-process communication, and unreliable data delivery [67]. Prime examples include the Open Software Foundation's Distributed Computing Environment (DCE)[4] implementation of the Object Management Group (OMG)'s Object Request Broker (ORB) [133, 154], the Oak Ridge National Laboratory's Parallel Virtual Machine (PVM) [67], and Plan 9 from Bell Laboratories [143].

At the time we began our work, the ORB specification was still under development. Therefore DCE was not a viable alternative for implementation. Plan 9 did not offer a good match for our functional requirements and was not widespread enough to satisfy our needs for portability. PVM was, at the time we investigated it, somewhat too primitive and not quite widespread enough to satisfy our colllaborator's portability desires. As a result, we chose to implement our own network service facility.

Alternative 1: Distributed Computing Environment (DCE)

DCE is a *software system* (see Section 2.2.2) that provides location transparency. That is, users see a shared system for running applications, but need not be concerned about where the computers that run those applications are physically located. Among other services, DCE provides a global naming service for files, remote procedure calls across architectures, security and directory services for locating programs and files, and a distributed file service. Figure 5.4 extends Figure 4.16 to show how DCE might be integrated into the computational proxy architecture.

The DCE global name space and file server would be acceptable for making input and output files available to the proxy and applications. The RPC runtime library, in conjunction with the preprocessor-generated stub code, provides system-wide, type-safe dynamic access to applications' signatures. Because DCE is based primarily on an RPC rather than a message-passing model, the Compute Monitor, running on a DCE host, would call a remote process (a compute daemon running on a DCE server) to start up an application. To implement the proxy experiment infrastructure using DCE, given our understanding of DCE, we would implement one active compute daemon per application. The compute

[4]DCE was formerly known as the Distributed Objects Environment (DOE).

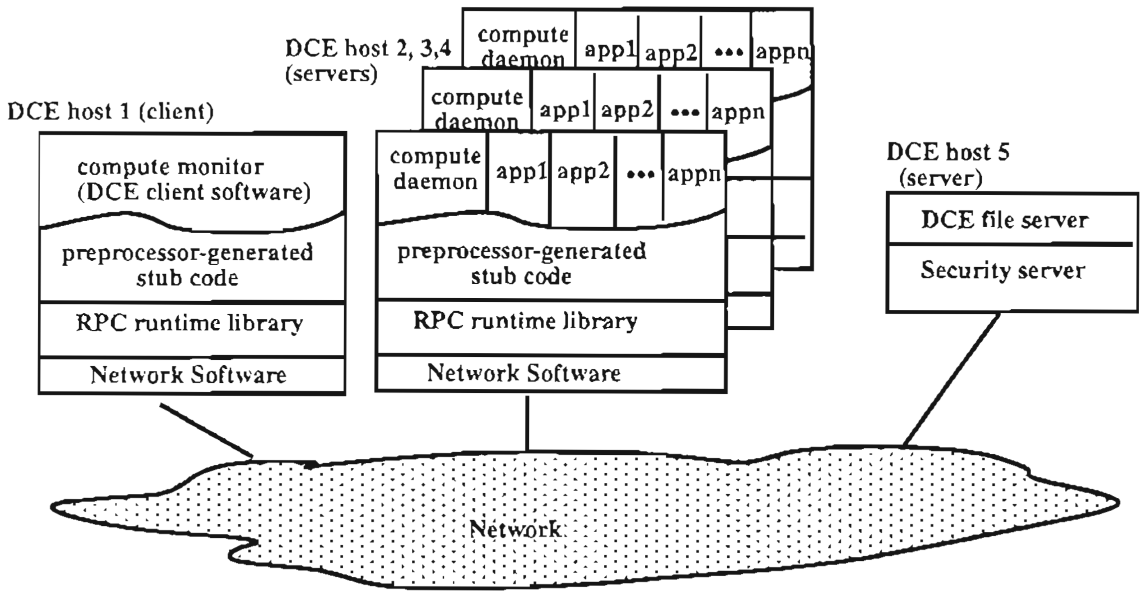


Figure 5.4: Using DCE to provide Network Services.

daemon would start up and manage the ongoing application processes. As stated earlier, however, we did not choose DCE for our prototype because its implementations were unavailable.

Alternatives 2 and 3: PVM and Plan 9

PVM and Plan 9 aim primarily towards partitioning computing tasks along lines of service functions [67]. PVM is designed specifically for use at the application program level, providing a basis for rewriting computationally intense applications from supercomputers to take advantage of processing power of distributed networks of workstations. We needed a distributed computing system that operated at a higher level of granularity than PVM, namely at the program rather than the procedure or subtask level. PVM would have been particularly useful for distributing one particular computational chemistry experiment over a network of workstations. Our major objective was, however, to schedule experiments one by one on computers in a network. While we could have used PVM to that end, we believe that implementing the proxy's network services in PVM would have involved writing almost as much code as was needed to implement the services using UNIX sockets. Furthermore (and more importantly), our collaborators believed that PVM was not available on the range of processor types they used, and thus that a prototype written using UNIX sockets would be more useful to them as a model for a future system.

Plan 9 is a multi-platform operating system and places special requirements on the computer platforms that run it in terms of network characteristics, processing and storage elements. We felt that Plan 9's requirements too greatly restricted the computational chemists' compute hosts and network services.

The Chosen Alternative: Implement our own Network Services

As DCE was unavailable, and PVM and Plan 9 not well suited to our needs, we decided to implement our own network service facility using UNIX operating system sockets and remote procedure calls. We required a combination of RPC and message passing at a higher level of abstraction than was offered by the alternatives. While we could have used the distributed facilities available in PVM to manage communication between our Compute Monitor and daemons, we felt that the considerable effort required to learn PVM and to extend its capabilities would not be worthwhile. We intended the implementation as a proving ground for the five functional requirements, but in retrospect observe that the proxy's network service component (implemented with UNIX sockets) is really quite portable because: (1) UNIX sockets themselves vary little from computer type to computer type, and (2) the proxy's network service component totals only about 100 lines of code. Thus, when porting the proxy's network services to a new processor type, one need review and revise a relatively small program that is unlikely to change very much. While implementing the proxy's network services using UNIX sockets was a good choice at the time we implemented the prototype, as lower-level network services offer greater functionality and become more widely available, our decision should be revisited.

5.5.3 Physical Design and Implementation

This section describes the physical design of the network services component of the proxy architecture and its implementation. In particular, we describe the program-level access to distributed computation services in terms of the division of the proxy responsibilities into two functional units, the Compute Monitor and compute daemons. We also show how information and control flow between the Monitor and daemons.

The computational proxy architecture is based on a client-server model. In a distributed database system a *database server*, typically one for each processor that houses part of the distributed database, handles data requests to that part of the database. There is sometimes a separate process (or a number of distributed processes), a *database monitor*, that keeps track of which servers and hosts are responsible for which data, and which database servers are currently running. In some systems, all database requests pass specifically through a database monitor.[5] Our network services architecture is analogous to those database architectures that have a central database monitor: a client process requests computation services from the proxy, which then passes the request to a *Compute Monitor*. The Compute Monitor is a process that keeps track of which compute hosts are currently available, and is responsible for sending a request for starting the computation to the appropriate host and then retrieving the result from the host. Compute daemons running on compute hosts serve a purpose analogous to that of database servers: on each compute host a daemon carries out computation requests from the monitor. The Compute Monitor and compute daemons are long-lived processes, started up as the machine is booted and remaining active as long as the processor is running.

The proxy is the mechanism whereby the client process communicates with the Compute Monitor and stores information about an ongoing computation (such as the process ID of the process carrying out the computation) across user sessions. Placing the process communication in the database between the user and the computation makes sense because information about the experiment (stored in the database) is required to launch experiments or to make sense of ongoing process information about the experiment. Figure 5.5 depicts the architecture for the computational proxy and computation services. The proxy's computation services consist of one Compute Monitor per database system, and many compute daemons, one for each processor where a computational application

[5]Because of the potential performance bottleneck with this design, some systems instead set up conventions so that a central database monitor is bypassed on specific requests. A database monitor mechanism, however, remains responsible for determining whether particular distributed databases are available.

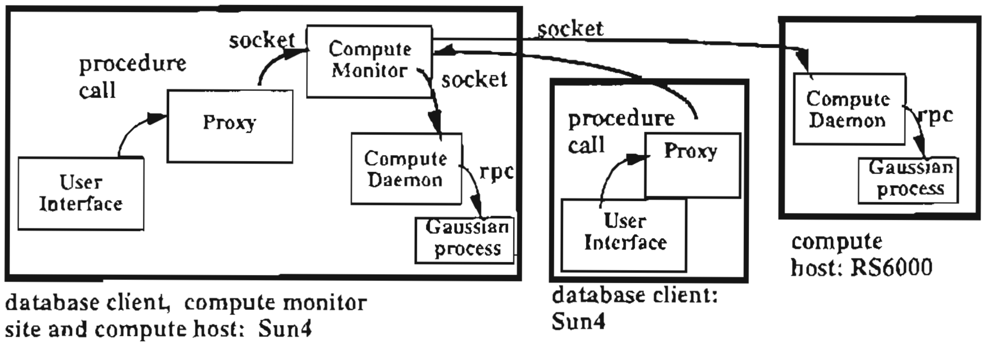


Figure 5.5: Implementation of the Computation Services Architecture.

might run. Just as in a distributed database system where a particular database server services requests for the data residing on the computer on which it runs, a compute daemon services computation requests that will run on the compute host on which it resides. In our system, the Compute Monitor runs on the same machine as the database monitor, but it could run on a different machine.

A proxy request to the Compute Monitor for computation services indicates which application to run, the input parameters and input file(s) to that application, and approximations of resources required. The Compute Monitor takes a request from the proxy mechanism and matches it with a particular compute host on the network. The Compute Monitor maintains in the database information about which compute hosts run which computational applications and about the status of each, e.g., available or unavailable; not busy or busy.[6]

Once a compute host is selected, the Compute Monitor moves its input file and sends a message to the appropriate compute daemon to start up the desired application. After thus invoking the application, the compute daemon passes the process identifier of the process running the experiment back to the Compute Monitor, and the computational proxy subsequently updates the database accordingly. This information (compute server and process ID) enables the proxy mechanism to monitor the ongoing experiment. The

[6]A host is "available" if it is currently running and connected to the network. A host is "not busy" if it has excess resources and can take on additional tasks.

Compute Monitor also manages the proxy's monitoring requests by passing them to the appropriate compute daemon.

When an experiment terminates, the compute daemon so informs the Compute Monitor, which updates the database with the status "experiment complete". The compute daemon sends the output file associated with the experiment back to the Monitor, thus triggering the parsing of the file and the placing of the results in the database. If the client process that requested the experiment is still active, the Compute Monitor signals the client process via the proxy. Note that neither the Compute Monitor nor the compute daemons write to the database. Only the computational proxy writes information about the computational process to the database; thus, there is no need for an operation analogous to a database commit to coordinate changes to the database record of the computation. The synchronous message passing functionality among the proxy, monitor and the daemons maintains the proper order of requests and responses.

The Compute Monitor and compute daemons are implemented in C. Messages are sent between the monitor and daemons via UNIX sockets. A compute daemon responds to two messages: (1) start an application process, and (2) determine the status of an ongoing application process. The computational proxy generates input files, and the Compute Monitor sends them to the appropriate host. We used the UNIX remote file copy (RCP) command to transfer input files to the compute host and output files from the compute host. File names are generated according to a simple algorithm using chemist's name, molecule and application. Output files are parsed by the computational proxy on the database host. For now, the Compute Monitor maintains a hard-coded list of machines on which specific applications run. Further details about the implementation of the Compute Monitor and daemons are available in Meenakshi Rao's master's thesis [151].

5.5.4 Summary of the Network Services Implementation Experience

Our experience in implementing the message passing, file transfer, remote procedure call, and procedure status request facilities needed for the computational proxy were useful in fine tuning the proxy's network services requirements. However, we believe that the network services facilities that we required and developed have a much broader user base than computational science. Such services should be provided by an operating system service, independently of the proxy architecture. Even with such services, however, the domain-level database and proxy infrastructure are both needed to supply the distributed system with application-level information about the required services. Furthermore, a database and proxy together are needed to supply a domain-specific application program interface (API) between client processes and distributed system services.

Our review of currently available network service facilities also indicates that our critical requirement of being able to detach from remote and ongoing processes is not directly provided by existing services such as PVM or Plan 9. To use these would require the implementation and systems management of Compute Monitors and compute daemons not unlike those we have already implemented. Our implementation experience has demonstrated that this capability to detach is both necessary and attainable. Thus, for an industrial-strength proxy implementation, we recommend building the proxy's computation services with a system such as DCE that implements the ORB specification.

5.6 Summary of the Implementation Experience

We end this chapter with a short summary of the prototype implementation experience. The computational chemistry database can store experimental data across three major, and representative, computational applications (GAMESS, Gaussian and MELDF) for five commonly used experiment types, namely Conventional Restricted Hartree Fock (RHF), Direct RHF, Conventional RHF with Gradient, Unrestricted Hartree Fock (UHF), and Single and Doubles Configuration Interaction (SDCI). These experiment types were chosen because they are the most commonly used.[7]

The proxy prototype mechanism successfully invokes computational experiments for the GAMESS application in a distributed environment, generates input files for GAMESS and parses output files for GAMESS. The prototype implementation was reviewed by computational chemists and application developers at Battelle Pacific Northwest Laboratory, who find that the prototype embodies adequate functionality for launching, controlling

[7]Our collaborators estimated that 90% of all experiments fell into these categories.

and mooring computational experiments we expect novice users to run.

While we did not formally analyze the performance of the prototype, we observe that the time the proxy takes to generate the input files and parse output files is virtually noise compared to that needed for running the experiment, and is more than justified by the convenience of having the proxy generate these files and manage experiments. With respect to transferring input and output files in their entirety across the network, we further note that we could observe no appreciable response time difference (on a local area network) between running experiments of about three minutes in duration on local and remote compute hosts. That this response time difference is imperceptible to the human user is not surprising, since the file size of output files rarely exceeds one megabyte, even for very large experiments. Transmitting these files over a wide area network (WAN) might take longer than over a local network, but users would typically run only larger experiments (processes running an hour or longer) over the WAN. For calculations running an hour or longer, the time required for sending files back and forth from the database host to the compute host is dwarfed by the time required to complete the calculation.

In addition to our observations about adequate functionality and performance, our experience suggests that implementing the proxy is cost effective in situations where a site provides end-user support for more than four applications.

Implementing the database and proxy identified additional database management system functionality needed to more fully support the computational chemistry database and its development. We saw a need for object-oriented database support for ternary relationships, and for additional object database tools, namely text parsing and formatting tools for database query results (i.e., report generation). Our development experience indicates a need for better integration between programming language and database development tools, class libraries for scientific applications, and a C++ dictionary class.

Finally, we observe that our implementation of the proxy involved only Sun workstations. To extend it for a heterogeneous environment would involve extending the Compute Monitor's socket-level communication and writing compute daemons for other platforms, a task that would involve reviewing (and perhaps modifying for different variants of the

UNIX operating system) about 100 lines of code in the proxy's network services component. While such special-purpose programming is probably cost effective, we believe that in the longer term it will become practical to rewrite the Compute Monitor and compute daemons using a distributed object service such as the Open Software Foundation's Distributed Computing Environment (DOE). Such high level services are preferable to lower level tools for distributed computing such as Parallel Virtual Machine (PVM), although any of these alternatives would probably be more cost effective in the long run to the low-level systems programming approach we used.

This prototype implementation demonstrated the feasibility of implementing the computational chemistry database and the proxy, even with existing database management, programming language, and network services tools.

Chapter 6

Validating the Model and the Proxy

Chapter 5 recounted the implementation of our infrastructure, and showed that the proposed design could be realized with a modest amount of programming effort relative to the benefit derived. Simply having a working system, however, only demonstrates the realizability of the infrastructure — it does not show that the database plus proxy are useful. This chapter, then, assesses the utility of proxies; here we evaluate the database and proxy design.

We have chosen to evaluate utility along three dimensions. First, in Section 6.1 we measure our infrastructure against a typical user scenario; we ask if user requirements are met by the infrastructure as it is now conceived. Second, in Section 6.2, we turn our attention specifically to the domain model of the infrastructure and ask three questions: Is the domain model sufficiently general to cover a useful range of ab initio computational chemistry applications? Are its structures intuitive enough for casual users and yet sophisticated enough to represent the needs of experienced theoretical chemists? Is the model extensible to a wider range of applications and chemistry domains? Third, Section 6.3 considers the effectiveness of the computational services component of the infrastructure, i.e., it evaluates the proxy mechanism itself.

We contend that these three measures constitute an adequate validation of the proxy infrastructure. If the infrastructure as conceived (data and computation services) meets the needs of computational chemists, if both these components as designed are both adequate for computational chemistry and extensible to allied domains, and if the design is feasible, then the proxy infrastructure is a valid solution to the problems facing ab initio computational chemists.

6.1 The Infrastructure as a Whole: A User Scenario

This section considers a user scenario for a single computational chemistry experiment to evaluate the database and proxy as currently designed. The scenario is the benchmark against which we verify whether our infrastructure does what it is supposed to do. We assume a user front-end that employs the database to help the user make intelligent selections of applications and application parameters. The user front-end "fills in a proxy" with parameters needed to run the experiment, and calls the proxy with requests for scheduling and monitoring the experiment.

In Section 3.1, we introduced a typical scenario for the use of computational chemistry applications involving a single state of a single molecule. We then identified which difficulties within that scenario were alleviated by providing a database of past runs. Chapter 3 concluded that such a database alone does not alleviate all of the difficulties in managing experiments in a distributed computing environment, and Chapter 4 introduced the proxy mechanism to address those difficulties.

Below, we specialize that same scenario to consider an experiment on ethylene, this time assuming a database of past runs, the proxy infrastructure and a user interface. The front-end is modeled on Dr. David Feller's prototype and design for the Computational Chemistry Input Advisor (CCIA) [51], currently under development at Battelle's Pacific Northwest Laboratory. We use the scenario first to show that the proxy fills an important gap in experiment management — a gap not readily filled by the database and user interface alone. Second, reading the scenario with an understanding of the proxy design and implementation demonstrates that the proxy does what it is supposed to do. Third, the scenario highlights features needed for the proxy to mature to a practical infrastructure for experiment management.

The scenario assumes the chemist's workstation is a Sun4 named *coho* that is running the user interface as a database client process. The GAMESS application is installed both on another Sun4 named *chinook* and on a Cray supercomputer; our chemist has access to each of these machines. Where the database itself resides is not relevant to this scenario, but the most likely place for it is on a third computer.

1. **Specify subject molecule and desired result.** Rather than using a molecular editor to define ethylene's structure from scratch, the chemist searches the database for previous experiments on ethylene. She finds a match by searching the database for all molecules with the chemical formula CH_4; two years ago, another chemist in the lab had used the Gaussian application to determine the lowest energy values for ethylene. In her current investigation, our chemist aims to determine an optimized structure for ethylene using a new implementation of the RHF algorithm introduced by the GAMESS application. Even though GAMESS and Gaussian use different input formats for molecular structure, our chemist can use the structure from the previous experiment as a starting point because the database can present molecular structures produced by Gaussian using formats required by GAMESS.

   Our chemist asks the front-end to set up a new experiment using the molecular structure from the previous experiment as a first estimate. In response to her request, the front-end sends a message to the proxy class which in turn creates a new instance of itself for the subject molecule, desired result, and application that the chemist has thus far specified.

2. **Consult previous runs on similar molecules.** The chemist uses the ethylene structure to search the database for molecules in the same family as ethylene, retrieving experiments on ethylene, ethane and methylene.

3. **Annotate records of previous runs.** While consulting previous experiments, the chemist notices that results of two of the experiments have since been corroborated by independent work at another laboratory. She attaches a personal annotation to those experiments.

4. **Choose input parameters for the current run.** The front-end uses its own rules of thumb and data from the previous experiments to offer "advice" on key input parameters. Drawing on this advice, in conjunction with her own experience as a GAMESS user and her own interpretation of the previous experiments, our chemist begins to select the required input parameters. She sets the ConvRHF Experiment Type parameter, and asks the front-end for advice on a basis set. Since

she is exploring its new structure-optimization algorithm, not all the information of the previous experiments will be applicable. For instance, GAMESS does not support the DunningDivGrad Basis Set, and one of the experiments on ethane uses that basis set. Two viable alternatives are presented by the front-end: STO-3G and 3-21G. Our chemist chooses 3-21G, and uses the same level of theory parameter that was employed by one of the previous experiments using it.

5. **Run the experiment.** Thus far, the experiment has been built inside the database, i.e., parameters have not been translated to the format required by GAMESS. Our chemist is now ready to run the experiment, which consists of the following subtasks:

   (a) **Select a target machine.** Because this is a preliminary run, our chemist decides to run the experiment on the Sun4 workstation *chinook*, and so informs the proxy.

   (b) **Estimate resource requirements.** To determine if the experiment as currently conceived is feasible for running on *chinook*, she asks the front-end for an estimate of resource requirements, e.g., CPU time and scratch disk space. Based on heuristics and previous experiments, the front-end predicts the experiment would take five days. That estimate agrees with our chemist's own intuition, so she decides to use the less powerful basis set (STO-3G). The front-end reminds her that the level of theory she had suggested is too high for the STO-3G basis set, and suggests an appropriate change. The new estimate is 6 hours of CPU time.

   (c) **Start the experiment.** Our chemist starts the experiment by issuing a command to the front-end, which in turn passes it to the proxy. The proxy then generates the GAMESS input file, offers the front-end (i.e., the chemist) the options of reviewing it and returning to Step 4 to modify input parameters. If the experiment is satisfactory as is, the proxy transfers the file to the Sun4 *chinook*. The proxy then sends a message to the Compute Monitor to invoke the application on *chinook*. Note that the chemist *starts the experiment*, but the proxy *generates the input file*, *transfers it to the computational host*, and *invokes the application.*

(d) **Monitor the experiment.** Two hours later, before going home for the day, our chemist asks the proxy for a status report on the experiment. She learns it has used twenty minutes of CPU time, and is using 400 megabytes of memory and two gigabytes of intermediate disk storage. Confident that the calculation is proceeding, she goes home.

Note that our chemist does not have to log on to the remote machine in order to monitor the experiment, nor to find out whether her experiment has terminated.

(e) **Transfer results.** When the process running the experiment terminates at 4:17 am, the proxy transfers the output file back to the machine on which the proxy is running.

6. **Analyze results, adjust parameters, and rerun the experiment until it runs successfully.** Our chemist notes that the experiment process terminated normally, and checks the results in the database. The results are plausible but because our chemist wants more refinement on the structure, she decides to run the experiment again. She sets up a new experiment using the molecular structure just calculated with the more powerful basis set and higher level of theory, and schedules it on the Cray supercomputer. On this second run, she also requests that the property hydrophobicity be calculated, because she will verify her theoretical results by corroborating them with a laboratory value for the hydrophobicity of ethylene. This laboratory experiment resides in the database as well, so a comparison of the figures can be made easily.

   As a result of her request to rerun the experiment, the proxy creates a new experiment object, replacing the old basis set and level of theory parameters. It then replicates itself, setting the computer platform on which to run the experiment to Cray.

   The chemist repeats scenario items 5 and 6 with the "new" experiment and proxy. Our chemist need not make extensive changes to her experiment as a result of changing the target machine on which the experiment runs; the proxy generates the new

GAMESS input file and transfers it to the Cray before starting the experiment.

7. **Make public and archive the results of the experiment.** Once our chemist has successfully completed the investigation and validated results, she makes the last computational experiment public, and flags the preliminary experiment on *chinook* to be archived in two months. Our chemist discovers that the hydrophobicity calculated by the successful computational experiment is corroborated by that measured by the laboratory experiment, and marks the computational experiment "confirmed". This confirmation creates a reference from the computational experiment to the laboratory experiment (and vice versa), so that other chemists can consult the confirming laboratory experiment for this structure optimization of ethylene.

Figure 6.1 summarizes the differences in support for computational chemistry experiment management offered (1) without database or proxy, (2) with an experiment database only, and (3) with the full infrastructure, i.e., database plus proxy. The proxy provides support needed for preparing, running and analyzing, and rerunning experiments — a gap not filled by the user interface and database alone. The proxy provides a uniform application program interface for the front-end across applications, a means of loading experiment data into the database, and a consistent view of each ongoing computational process. A number of practical details must be resolved before the proxy can support the scenario above scenario. This follow-on work involves engineering changes to the design and to the implemented prototype; they are specified below.

Assuming a proxy front-end, the current proxy design can support the scenario above with the exceptions noted below:

1. Substructure definition and structure-level searching. By substructure definition, we mean the ability to define substructures of the subject molecule. By structure-level and substructure-level searching we mean the ability to search the database of past experiments for experiments on molecules of like structure. Substructure definition and structure-level searching needs to be integrated with the data model and implemented before previous experiments on similar molecules can be retrieved. The current implementation searches only by molecule name.

	Currently	Database Only	Database + Proxy
1. Define molecule.	from scratch, with editor	can find structures of previous experiments	structures easily transformable
2. Consult previous experiments.	ad hoc	can find "like" experiments	can find "like" experiments
3. Annotate previous experiments.	ad hoc use file system	can annotate the public database	can annotate the public database
4. Choose input parameters.	ad hoc	can view "like" experiments	can also view and transform parameters
5. Run the experiment.	ad hoc	viewing like exp helps: • select machine • estimate resources	can also: • build & transfer input file • start & monitor experiment • transfer & capture output
6. Analyze results, adjust parameters.	ad hoc	side-by-side view of experiments in db	output auto-loaded to db easy to adjust parameters
6a. Rerun experiment, until successful.	ad hoc	ad hoc	potentially easy to replicate experiments
7. Make public and archive results.	ad hoc	must explicitly load results into database	automatically load results experiment validation

Figure 6.1: Support for managing computational experiments — Currently, with Database and with Proxy plus Database.

The absence of substructure definition and structure searching does not affect the operation of the proxy *per se*. Current research and development in molecular structure searching addresses these problems [45, 105, 121, 194],

and the CCDB could be extended to support these new techniques.

2. Support for the relationship between computational and laboratory experiments. A computational experiment can be driven by anomalies in the laboratory, or can be used as a precursor to a laboratory experiment.

   The current data model has not explored the structures of laboratory chemistry that would be required to set up more than a relationship that simply connects two specific experiments.

3. Provisions for annotation. Extending the *Experiment* class to provide for individual annotations of previous experiments would provide individual scientists a mechanism for storing and sharing their comments about others' experiments. Delcambre's current research in scientific databases addresses this issue [46].

In addition to the conceptual changes above, our analysis of the proxy infrastructure indicates that the following engineering issues would have to be considered before the system could be deployed in a practical setting:

1. Authorization and billing. For now, the proxy starts up a remote job without regard to whether the chemist has an account on that machine. Only the proxy need have an account. Obviously, if experiments were billed to individual chemists, or chemists' groups, then the database would have to keep track of who ran what when, and what resources were used; alternatively, the proxy could pass the user ID through to the network services, but this would require the database to store passwords, the proxy to provide user authentication and the network services to conduct a remote login.

   While critical to the deployment of the proxy infrastructure, implementation of authorization and billing was outside the scope of this thesis.

2. File transfer optimization. The user scenario calls for the transfer of input files generated by the database to the compute host, and of output files generated by the application back to the database machine. We have not optimized the proxy design with respect to file transfer. In particular, network traffic improvements could be attained in some cases by shipping objects, rather than text files, across the network.

We felt that the above changes constitute engineering rather than research issues, and that the usefulness of the proxy mechanism could be demonstrated without considering authorization, billing and network traffic considerations.

6.2 Data Services: The Computational Chemistry Model

In this section, we evaluate our data model's generality, extensibility and usability. First, to determine if our model was sufficiently general to cover a useful range of ab initio computational chemistry applications we asked our collaborators to indicate which objects were most important and most likely to exhibit variability across applications. We also asked which applications would be most representative with respect to text representation of those objects. Molecular structures and molecular orbitals were chosen as the most important complex objects for the domain, and Gaussian and GAMESS chosen as the representative applications. We then tried to represent the text equivalents of molecular structures and molecular orbitals for the Gaussian and GAMESS applications, and saw that our model and template structures were able to express those objects. Though hardly conclusive, these preliminary checks offer some evidence that our model and template structures are general enough to represent the ab initio computational chemistry objects of interest.

Secondly, to determine if our model is extensible to a wider range of chemistry domains, we reviewed our model with other researchers interested in developing an object-based analysis of chemical research. Other researchers from CAChe Scientific, Battelle Pacific Northwest Laboratories, and IBM Almaden Research joined us in an effort to develop extensible computer-based systems that help the experimental chemist conceive, conduct, analyze and report chemical experiments. Dubbed the "Chemistry Objects Research Architecture (CORA) Working Group, this group developed a conceptual model (Figure 6.2) to support systems that enable both laboratory and theoretical chemists to use molecular modeling and theoretical chemistry tools [118]. Similarities between our model and the CORA model become obvious as one changes the names of the CORA entities to those of our model, e.g., *chemical sample* to *molecule* and *list of projects* to *suite of experiment*. The CORA concepts of *plan* and *schedule* are analogous to the functions performed by the proxy infrastructure. The similarities between our model and the CORA model offer some evidence that our model could be extended to a wider range of chemistry than ab initio alone.

To measure the usability of our model, we asked computational chemist Dr. David Feller to determine whether he, an expert chemist, would find the model usable. We also reviewed our model with systems analysts at Battelle Pacific Northwest Laboratory who had conducted a user survey of computational chemists. Both Dr. Feller and the PNL systems analysts found our model adequate. Though these offer evidence that our domain model is satisfactory, Battelle's user surveys and our own reviews with the user and registrar scenarios indicate that the following extensions are needed before the system would be of practical use:

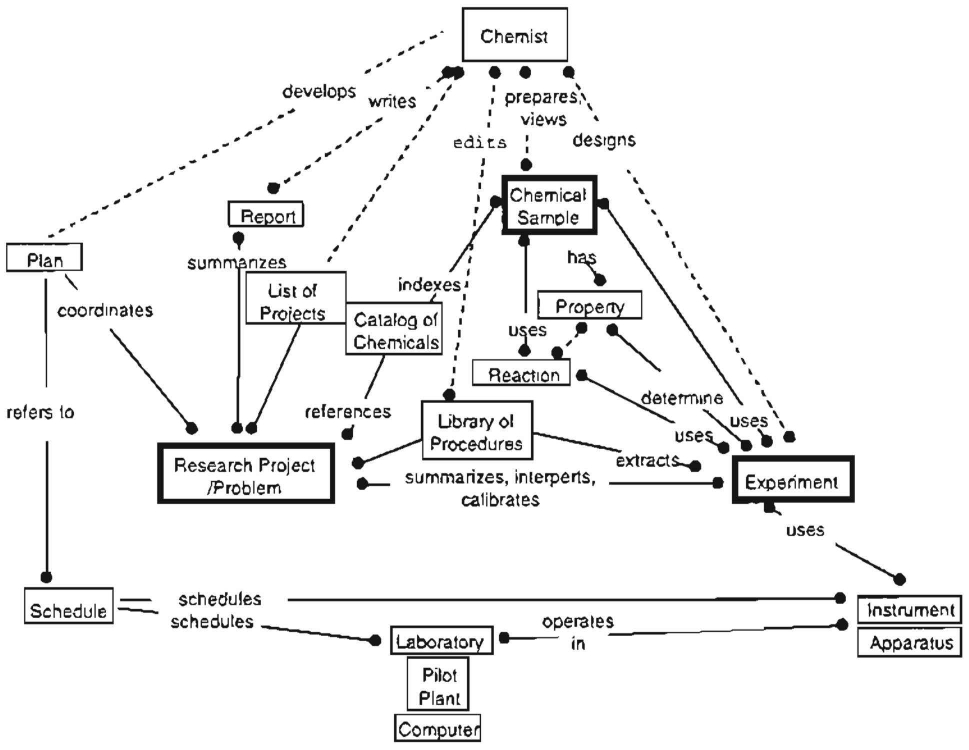


Figure 6.2: The CORA Working Group Chemistry Model

- Generalization of *Property*. Each observable property in the database is now explicitly modeled as a distinct database class or class attribute. The concept of property should be abstracted so that new properties can be defined by users dynamically, as instances of the property class.

- Additional ways of representing molecular structure. For now, molecular structure is represented only as Cartesian coordinates. While this is increasingly the representation of choice, most ab initio applications accept other molecular representations as input. Furthermore, many individual researchers have legacy data in those representations. Our model should be extended to represent molecular structure in internal coordinates, and should be modified to accept data using standard display formats such as that of the Brookhaven Protein Data Bank (PDB) [2].

- A more extensive implementation of the *laboratory experiment* class. While our model currently represents laboratory experiment and laboratory apparatus, we have not provided full support for comparing computational and laboratory experiments. Such support would be needed for connecting computational and laboratory experiments using the *confirms* relationship. The computational chemistry database should be extended to input laboratory data output by computerized laboratory equipment.

- Further specification and implementation of the extensions to *Experiment* for annotation, and to *Molecule* for molecular substructure identification and structure searching.

- Further specification of automated data migration. We do not provide for migrating unsuccessful experiment objects off-line. A chemist will likely wish to preserve only the final "successful" experiment in an investigation. We provide no automated mechanism for deleting or archiving unsuccessful experiments. Without such a mechanism, the database will grow inordinately large.

6.3 Computational Services: the Proxy

In Section 6.1 we addressed the issue of whether our infrastructure as a whole meets the functional needs of the chemist. In Section 6.2 we went on to consider the domain model, and asked if it was general, extensible and usable enough to deliver the data services required by our infrastructure. Here we evaluate the part of the infrastructure that models application programs and processes, i.e., the computational proxy itself. Whether the proxy is sufficiently efficient has been addressed along with implementation; we noted that the time needed to generate an input file and send it, and to send an output file back and parse it is virtually "noise" with respect to the time needed to complete the average computational experiment. Whether the proxy is effective has been addressed in Section 6.1 above. Here, we emphasize the extensibility of the design. We believe that the critical test for our infrastructure involves a determination of its extensibility and portability. Is the effort we have expended amortizable over a number of different applications for ab initio computational chemistry?

To answer this question, we first consider (Section 6.3.1) how easily new ab initio computational chemistry applications can be added to the system. An allied question is how easy it is to add support for computational applications running on "new" computer platforms. By appealing not to an end-user scenario, but to a registrar's scenario — the steps needed to add a new application to the system, we show the extensibility of our design, and determine what is needed to deploy the proxy and database beyond the computational environment explicitly supported by the prototype.

We then turn to the question of whether the *experiment* and *proxy* classes serve a useful range of experiment types and applications. In Section 6.3.2 we identify the range of application input and output we can support with the current design. Finally in Section 6.3.3 we ask whether the proxy design is portable to other database management systems than ObjectStore and other platforms than Sun.

6.3.1 Adding New Applications to the Infrastructure.

This section considers the effort required to add new applications to those already supported by the proxy, and to maintain existing applications across application program modifications. For the sake of simplicity, we assume that our methods for expressing the new application's input and output files are adequate; this question is explicitly addressed in Section 6.3.2.

For an application to be accessed from our infrastructure, the computational proxy must "know" what platforms and machines that application runs on, the application signatures, and how to map database objects to and from the application's input and output file formats. The process by which the proxy comes to "know" these facts is called *registration*. As described in Chapter 4, our aim is for application registration to be conducted by a chemist, the *registrar*, who knows the computational application well, but who need not be a programmer. The application registration facility is also used to update an old registration to match a new release.

In this section, we outline the process for registering an application. We ask whether the effort required to register and maintain an application with the proxy is justified by the benefit the proxy delivers. Below are the steps now required to register a new application:

1. Add the appropriate instance of *computational application* to the database. The user first adds the following information to an ASCII file: name of application, platform type(s) on which the application runs, application version, date installed. Then he or she runs a program to reload all application objects into the database.

2. Determine which chemical properties generated by the application one wishes to load into the database. If a property of interest is generated by the application but is not currently represented in the database, it must be added.

3. Determine the experiment types supported by the application. This involves studying the application to determine which of its options support the experiment types already defined. The registrar may wish to add new experiment types to the CCDB.

4. Write the input and output templates needed for each experiment type to describe

the input and output files for the application. If certain display types in the application cannot be handled by the template descriptions, the registrar must write appropriate input, display and conversion methods. As with application instances, the registrar must reload existing template instances to add new template instances.

5. If the application to be registered runs on a computer platform not already supported by the proxy, then one must add an instance of *computer platform* to the database and set up the network services component to support that platform. Adding a new computer platform is conceptually the most complex task in registering a new application, and must now be conducted by a programmer.[1] To add a computer platform to the database, one must:

   (a) Adapting a compute daemon program to the new platform. This probably involves about 100 lines of code.

   (b) Add an instance of *computer platform* to the database.

   (c) Add one or more instances of *particular computer* for that platform type to the database. In addition, one must associate a socket number to each new instance of *particular computer* on the computer where the Compute Monitor is running. On each new particular computer, one must similarly associate a socket number for the computer on which the Compute Monitor is running. Socket numbers are now hardcoded into the Compute Monitor.

6. Add an application signature to the infrastructure for each platform on which the application runs. The application signature for each application, i.e., its procedure call, is now hard-coded (not data-driven). Thus, the registrar (or a programmer) must modify the C program that generates the procedure call to the application. The locations of input and output files and the application signature are currently programmed into the part of the infrastructure that starts up experiments. Adding the application signature involves adding about 6 lines of code for each experiment type for each application.

[1]Currently, the proxy will work only with platforms running versions of UNIX that support socket communication.

The most tedious of the tasks above is preparing input and output templates for an application. Neither adding an application signature nor adding a new platform is difficult, though these activities require some familiarity with the infrastructure internals and with the C language. A half-day should be sufficient for a database programmer working with a registrar who knows the application. Adding a new platform type, however, requires the services of a systems programmer. Using the existing program as a model, adding a new platform would probably take a week to ten days.

After considering the registration procedure that we propose above, we observe that the following changes to the data model and the proxy infrastructure would make the system easier to manage:

1. Experiment type should be explicitly added (as an entity) to the database schema. "Experiment type" is now represented only as a text field in experiment. As we evaluated our current design and implemented our prototype, however, experiment type emerged as a prospective semantic device for comparing experiments and grouping application descriptors such as input and output templates.

2. Molecular property should be generalized, if possible, as a single database class. Our domain model currently specifies each chemical property of interest as a separately described entity. Whenever a new application is added to the system, the properties it calculates must already be represented in the domain model. If the domain model contained one general representation of chemical property, adding an application that calculated a "new" property would mean defining a new property in terms of the general representation. The registrar would not have to make a schema change.

3. The database loaders for *computational application*, *computer platform* and *particular computer* should be rewritten to add instances incrementally to the database, rather than having to reload all instances en masse. Clearly, reloading all applications when adding one is impractical for a working system. A data-entry program for these classes would be more appropriate than a program that loads classes from files. Such a program would be more easily written were the database system to provide an object-entry tool (just as most relational database systems include form-generation

facilities)[2].

4. A registrar user interface should be provided. Such a user interface would generate templates in our data description language from a graphical display of sample data structures.

5. As intimated in the discussion on network services in Section 4.4, the network services should provide a facility for registering platform types and platforms, and for calling application programs across the network. This facility would greatly simplify the steps in our application registration procedure that involve adding application signatures, computer platforms, and particular computers.

Even without the changes above, we conservatively estimate that it would take twelve days for a registrar to add a new application (running on a new platform). This estimate assumes that the registrar is already familiar with the syntax and semantics of that application, and that only a few major sets of experiment types are supported for each application. An additional ten days of systems-level programming is required to add a new platform under the current implementation. If the changes above were implemented, the time required for registrar and system programmer might be reduced by half.

The process for maintaining the proxy support of an application across new releases of the application will typically involve following each of the steps laid out above for new applications. At each step, the registrar must determine the extent to which the database description of that application must be modified for the new release. Adapting to a new release could be as simple as modifying an output template to add a new observable property as output, or it could involve adding numerous experiment types and computer platforms to the infrastructure.

Whether or not the benefit derived from using the proxy justifies the effort spent implementing the proxy infrastructure and registering each application depends on many factors such as: the frequency of use of computational applications at a site, the likelihood of a single user using several different platforms, and the level of user expertise with

[2]Servio Logic's *Geode* provides an "object-entry" tool.

respect to different computing environments and applications. The extent to which a large majority of experiments at a site can be expressed as manageable number of experiment types supportable by the proxy is also significant. It will not be practical to designate experiment types to fit all combinations of parameters to all applications. Our intuition is that the proxy will be beneficial in laboratories where most experiments run by non-expert users fit into less than a dozen experiment types, and where users are generally willing to define their experiments using the basic experiment types, modifying the experiment input files by hand if needed for more complex experiments.

6.3.2 Profile of Applications Readily Supported

Clearly the efficacy of the proxy depends on the expressiveness of the languages that describe input and output files. This section profiles the applications and experiment types supported by the proxy infrastructure, describing the range of inputs and outputs that can be represented by the proxy's templates.

The proxy's input file description language, CCIL, allows for describing input files composed of the following textual objects:

- Database objects formatted using the default display method from the schema or an alternate display method. There are no "automagic" higher-level display types such as matrix (as in the output file language). Minor control for complex types can be imposed by breaking up the type into its components, specifying the display format separately, and iterating over the members of a collection. For example, the user can define a display for molecules by formatting the molecule level information, and then using the ITERATE OVER command to display each atom according to a particular method or format.

- Textual objects generated from database objects using simple data transformations, e.g., a database value of "MP1" transformed to "1".

- Default values for database objects with null values for an application or experiment type.

simple textual objects given as single tokens, and complex textual objects such as molecular structure and molecular orbitals, even where the number of tokens differed across a sequence of textual objects of the same type.

6.3.3 Portability of the Proxy System

The proxy has been implemented using ObjectStore on Sun3's running the SunOS. Thus, for now, the infrastructure prototype handles only Sun3 workstations. Though we attempted to use general UNIX concepts in our design and portable procedures in our implementation, we have not tested the portability of the infrastructure to other platforms. As for porting the infrastructure to run on other platforms, the database management system, ObjectStore, runs on a variety of workstations. Thus, the major effort in porting the infrastructure to work on any of the platforms supported by ObjectStore lies in porting the Compute Monitor (written in C) to those machines.

Using a platform other than Sun as a compute host for computational applications is discussed above in Section 6.3.1, and requires porting the compute daemon. The infrastructure can support a platform if the socket interface is supported between the Compute Monitor and the new platform. In general, code using UNIX sockets is easily portable [186].

We have not as yet considered porting the proxy to a database management system other than ObjectStore. Porting it to another object-oriented database system would be practical as long as the programming language that interfaces with the database management system supports calls to C programs. Porting the infrastructure to a relational database system would be quite difficult, because we have taken advantage of the ability to implement behavior as part of the abstract data type definitions.

6.3.4 Concluding Remarks on Evaluating the Computational Services

To gauge how applicable the infrastructure is to other computational chemistry applications, we have considered the overhead required to support new applications and new platforms. With few or no changes the proxy design will service an application that has the following characteristics:

- Takes input and output from textual files.
- Runs on UNIX platforms that support sockets.

Obviously, of course, new applications that interface directly with the database can be easily supported.

6.4 Summary of the Validation

In this chapter we provided three measures of effectiveness in managing computational chemistry experiments, and we evaluated our proxy infrastructure against those measures. First, we showed that, as currently designed, the infrastructure (i.e., the computational database and proxy together) meets the needs of computational chemists as defined by a typical user scenario. Next we described why we believe our database design is general, extensible and usable. (1) We demonstrated generality by showing that the database can handle the most complex objects for representative applications. (2) We demonstrated extensibility by showing that our model is consistent with a more general domain model for chemistry. (3) We demonstrated usability by reviewing the system with computational chemists who use ab initio applications and with analysts developing support applications for computational chemists.

That the infrastructure is extensible to domains traditionally allied to ab initio chemistry strengthens this validation. Our evaluation of the infrastructure's computational services (i.e., the proxy itself) showed extensibility of the proxy by developing a profile of the applications and platforms that the infrastructure would support with little or no change. We conclude that the infrastructure is usable as designed for computational chemistry experiment management. However, the validation process suggests that certain improvements will enhance the proxy's value; these are summarized in the section on follow-on work in Chapter 7.

Chapter 7

Contributions, Lessons-Learned and Future Work

Our research has resulted in an infrastructure for computational experiment management that solves problems of data management and program interoperability. We focused on one particular domain of computational science (ab initio computational chemistry), believing that delving into one domain in detail and then generalizing would yield better results than surveying the field. In Chapter 2, we set the context for this work by describing research related to our own. We went on in Chapters 3 and 4 to show how the current computing environment available to computational scientists falls short of providing adequate support, and then articulated our design for the domain database and computational services that constitute our infrastructure. In Chapter 5, we described the prototype implementation that not only helped us refine that design but demonstrated its feasibility. Finally, in Chapter 6, we evaluated our infrastructure.

In this chapter, we conclude the discussion of our work. Section 7.1 describes the problem we set out to solve and roughly outlines our approach. We then go on to briefly summarize our solution to the file management and interoperability problems facing computational chemists in Section 7.2. Section 7.3 lays out our contribution to computer science research. With a more practical flavor, Section 7.4 recounts the lessons we learned — lessons that we hope constitute helpful feedback to those developing computational science applications and infrastructure. In Section 7.5 we suggest follow-on work to our research, indicating in Section 7.6 what work we ourselves intend to pursue. We conclude the thesis in Section 7.7.

7.1 The Need for an Infrastructure for Computational Experimentation

Advances in raw computing power over the last twenty years have enabled scientists from many disciplines to effectively model physical phenomena. Indeed, "computational science" is becoming a subdiscipline in its own right, of such complexity that not only writing, but using, the programs that implement these computational models of physical phenomena requires considerable training. Unfortunately, even highly trained computational scientists find themselves overwhelmed with the complexity of the computing environment in which they work.

A computational scientist typically uses five or six applications, distributed on a network of three or four different hardware and operating systems. Because input and output file formats for many applications are idiosyncratic, the scientist must deal with numerous file format conversions, sometimes writing programs to perform those conversions and usually having to explicitly invoke the format conversion programs. Because a typical scientific study may involve running hundreds of computational experiments, these scientists also must deal with complex file management tasks — moving files back and forth among several platforms and keeping track of hundreds of input and output files. The environment is further compounded by the fact that the applications are computationally intensive and "long-lived" — a single experiment takes anywhere from several minutes to several months to complete. In sum, computational scientists work in a complex laboratory consisting of semantically complex distributed applications that require significant file management, file format conversions, and the use of different programs running on a number of different platform types.

Traditionally, computer science research in the service of computational science has aimed to make the computations run faster. While there is of course still need for this, we also need to make computational applications easier to use — both for specialists and for users just now gaining access to these powerful tools. Our own research falls into this latter category, aiming to facilitate use of computational applications. We initially believed a database of past experiments for the computational scientist would adequately simplify the environment. Such a database would provide important examples to use as models for setting up new experiments as well as alleviating file management tasks facing scientists. However, our exploratory research revealed that a database to hold experiment data was not enough: we also needed to populate that database and to support the actual running of the applications. Making the database an integral part of the actual running of experiments could accomplish both goals, but to achieve this we needed to link the applications to the database. Because it is highly unlikely that these applications will be rewritten to connect directly with either a database system or any particular infrastructure software (at least in the near future), we treated them as legacy applications, albeit legacy applications for which new versions are released once or twice yearly. The nature of these "legacy applications", however, precluded interfacing them to the database using naive encapsulation techniques.

Motivated by these observations, we designed an infrastructure for experiment management that consists of a domain-specific experiment database and an object-oriented interface relating it to the applications. Our infrastructure is "middleware" — sitting between a user interface and the standalone computational applications installed on a distributed network of compute hosts. Because we believe the key to interoperability is a common conceptual view of the underlying data, and because of the value we place on maintaining persistent records of ongoing experiments that can be made available to cooperating distributed user applications, we built this "middleware" within a database tradition. The infrastructure in effect extends a database management system to support computation services.

To demonstrate the feasibility of our approach, we implemented a prototype. We chose an object-oriented database system as the implementation vehicle because we believed that the complex scientific data with which we dealt could best be modeled by object-oriented structures.

7.2 Computational Proxies: An Infrastructure for Experiment Management

Our solution to computational science interoperability problems consists of a single domain-specific information model to facilitate comparability of experimental data and of an object-oriented persistent structure to maintain information about application programs and experiment processes. We implemented our infrastructure as a database to provide persistent records of ongoing and past experiments. We chose object-oriented technology for implementation because of the high level of complexity of the information model — we did not feel that current record-based database technology could model these structures effectively.

Our data-centered infrastructure supports computation as well as data management. The domain-specific database provides database services that meets the twofold needs of individual scientists: keeping track of experiment inputs and outputs for one's own work, and referring to previous experiments that can be used as models for future experiments. We developed conceptual and information models for the domain of ab initio computational chemistry; from the information model we prepared logical and physical designs, subsequently implementing the Computational Chemistry Database in an object-oriented database system, ObjectStore.

The other component of our infrastructure provides computation services, and essentially consists of an abstract data type, the computational proxy, that models scientific programs and processes within an object database. We have developed computational proxies specifically for the domain of ab initio computational chemistry, but believe the resulting infrastructure is applicable to computational science in general and will help simplify the complex computing environment in which computational scientists find themselves.

A computational proxy is an object-oriented abstract data structure — with accompanying functions and database services. A proxy represents, within the database, an active process running a scientific application (usually on a remote computer). Proxies model not only application programs (computational characteristics and interface), but also the invocation and execution of those programs. Because proxies model the execution of programs, they are especially useful for controlling lengthy application processes — users can consult a proxy to determine the status of a computation or tell the proxy to stop a particular computation without having to log on to the remote computer that is running that computation. Proxies launch and monitor experiments, generate data input to the experiment from the database, and capture experimental results. We have endeavored to provide the computational proxy's functions while hiding semantic and syntactic differences between applications and environments. Because the proxies maintain persistent records of on-going experiments in the database, those records are readily accessible to the scientist even if the computer that is running the experiment is temporarily inaccessible.

Our ultimate goal for the proxy mechanism is to provide ways to register computational applications declaratively with the database, without writing special purpose programs. We made progress toward this goal with the conception of templates. Templates are written specifically for a particular application by a user (the registrar) who is very familiar with that application. Input templates contain a specification of an application's input file designed so that the proxy can interpret the template in the context of experimental data and generate an input file for that application. We have specified a language for defining input templates called the Computational Chemistry Input Language (CCIL). In our design and implementation of output templates we have made further progress toward the goal of registering applications declaratively. Output templates describe output files for a subset of computational chemistry applications and guide the proxy's parser in loading experimental results into the database. We have described a language for output template specification, the Computational Chemistry Output Language (CCOL), and implemented the corresponding interpreter, the Parser-Converter-Loader (PCL).

Concomitant with the goal of declaratively defining computational applications to the database is our second goal, running those applications automatically — without requiring the user to intervene manually. We believe that the scientist need be familiar with no more than one operating environment, but still be able to run experiments in many other environments. We have good progress toward this goal. The infrastructure we have designed and implemented includes a component called the Compute Monitor. When an experiment is scheduled by the user, the proxy passes a message to the Compute Monitor, which passes the input file to the machine on which the experiment is to run and in turn passes a message to that machine to invoke the chosen application. Another infrastructure component, the compute daemon, resides on each computer where applications are run. A compute daemon receives and carries out the Compute Monitor's requests to start, query and stop experiments. While we have implemented programs for message passing and remote process control (via UNIX sockets), we believe that a more viable implementation would use a vendor-supplied network service. Because (to our knowledge) no appropriate network service yet exists, we have defined the requirements for such a service.

Our third goal for this research is to provide a suitable migration path for the legacy applications with which we work. Both input to and output from these applications is currently effected via ASCII files. In the proxy infrastructure, we generate an input file for each experiment and copy it to the compute host on which the experiment is to run. Experiment results are captured by parsing the application's output file and loading objects into the database. Eventually, we advocate modifying the applications themselves to read and write objects directly. However, we realize that this goal is not feasible until considerable practical experience helps determine what object interface is appropriate for this domain.

7.3 Research Contributions

Computational proxies offer functionality not now provided in database systems, fulfilling key requirements for database use by computational scientists. The infrastructure we propose can be used in a migration path from stand-alone legacy applications to an integrated environment where users can invoke applications distributed across a network without having to learn syntactic idiosyncrasies of multiple applications and command languages. As important as the seamless launching of applications is the possibility of comparing outputs from different programs, or of using the output of one program as the input to another, without having to write file conversion programs. Our infrastructure can also be used directly by applications that can read and write structures conformable to our database.

Our information model has been validated as intuitive by an expert computational chemist and systems analysts at Battelle Pacific Northwest Laboratory who have conducted a preliminary needs analysis for a similar database. Thanks to a related conceptual modeling effort (the CORA project), we have come to believe that the model is generalizable to laboratory chemistry, and that it is intuitive to chemists already working within the lab chemistry tradition. Finally, we have some evidence that the model is sufficiently general to serve most non-expert use of three representative code packages. (Non-expert use is expected to increase dramatically over the next five years.) We also suspect that generating input files and parsing output files for the general cases we define can provide first-cut input files and some support for loading results into a database for expert users.

We have demonstrated the feasibility of the infrastructure (database plus proxy) by implementing its salient features in a commercially available object-oriented database system (ObjectStore). The performance of the prototype database itself is adequate for generating inputs from the database and loading outputs into the database. Our implementation of the output template interpreters has suggested that output file parsers are amenable to automated construction and thus further validates the infrastructure. We have illustrated the proxy's utility by comparing the amount of programming required to interface computational applications to a database with and without the proxy.

The major contributions of our work are described in the sections that follow. Its more general significance lies in the evidence it offers of the utility of a unifying data model as a first step toward solving interoperability problems and of the promise of a data-centered approach to simplifying a heterogeneous computing environment for the end user.

A Conceptual Data Model for Computational Chemistry.

Even though much of the information contained in a conceptual data model is implicitly understood by those who work within a given domain, it is usually not written down and certainly not written down in a manner accessible to computer science researchers. In this situation, subtle differences arise between how individuals or research groups define specific entities or relationships. Researchers from other sub-domains of the same discipline, or from other disciplines will not have internalized the model. Even if some sub-domains use the same terms, the terms may carry different meanings.

Making explicit the conceptual data model implicitly used in the sub-discipline sets the semantic foundation for both interdisciplinary work and data interchange between programs and people. Without a conceptual data model common to the application programs commonly used in the sub-discipline, the syntactic differences between these programs cannot be resolved. Without a conceptual data model common to the application-domain supported by the proxy, for example, we would not be able to define experiment types across applications because we could not establish semantic correspondence between properties output by two different applications. Without a common conceptual data model, it would be scientifically unwise to use the output of one application as the input to another, or to compare two outputs of the same name from two applications. The benefits above derive from a common conceptual model whether or not it is actually implemented as a common representation; where the model is implemented, we can resolve data incompatibilities among n formats with writing $2n$ transforms.[1]

Without a conceptual data model common to the application programs commonly used in the sub-discipline, neither the semantic content nor syntactic form of data produced in one laboratory would be fully discernible to scientists working in other laboratories. As important, without such a model it is difficult to write programs to import data accurately from outside laboratories.

Our data model for ab initio computational chemistry contributes to the possibility of a future data interchange standard not only for computational chemistry but for the molecular sciences in general [118].

The Proxy Structure and Mechanism

The idea of modeling a computational process as a database construct suggests possibilities for similar innovation by other researchers seeking ways to provide integrated computing

[1]While it is true that data differences between programs can be resolved where both semantic and syntactic correspondences have been defined, one needs n^2 data conversion routines to resolve syntactic differences among n programs.

environments. An important characteristic of computational science applications is their complex inputs and long execution times. Representing both the application program and an invocation of that program as objects allows us the flexibility we need (1) to separate supplying inputs to the program from scheduling its execution, (2) to run the program asynchronously from the database client session, and (3) to control and monitor the application process. The naive wrapper approach, on the other hand, does not separate invocation from execution. Our separation allows fine-tuning (or replicating) invocations and controlling execution. A wrapper can only superficially filter input and output, and cannot monitor an experiment [117].

Declarative Specification of Program Interfaces

The mechanisms we have developed to define application interfaces declaratively, without writing new programs, contribute to the research areas of program and database interoperability. The interfaces to the application programs in the domain with which we are dealing are typically ASCII files. Using our Computational Chemistry Output Language (CCOL), a user (the registrar) can write output templates that define the output file formats for these applications. The proxy interprets output templates in the context of a particular experiment object to parse an output file and load experiment results. The Computational Chemistry Input Language (CCIL), based on our experiences with the CCOL, is a first step toward a similar declarative specification for input file formats that would guide automated input file generation. The CCIL suggests possibilities for object-oriented database report writers.

The direct value of the CCOL and CCIL to our own infrastructure is their provision of declarative specifications to help automate the interface between the database and application program, avoiding tedious and error-prone special-purpose format translations.

More generally, the languages and mechanisms in our infrastructure suggest (1) a basis for layout languages and parsing mechanisms that could be incorporated into object-oriented class libraries, and (2) an alternate way of approaching some schema integration tasks, i.e., resolving syntactic differences among semantically equivalent data.

The Prototype Database Itself

The database we developed not only confirms the feasibility of our design, but also provides an infrastructure for further research in integrating intelligent user interfaces with scientific systems and in developing distributed database and operating system support for this application domain.

The proxy enables building a database that can serve as a repository of past experiments. The user can draw upon this database directly or indirectly (through the user interface) in setting up parameters and or estimating resource requirements for prospective experiments. The computational chemistry database thus establishes an empirical basis upon which both human users and intelligent user interfaces might make rational scheduling decisions. With data about past experiments, one could experiment with heuristics for parameter selection, and policies and mechanisms for distributed scheduling.

7.4 Lessons Learned

We readily admit that computational proxies are most useful in domains where a relatively stable conceptual model covers a large majority of the applications in use and that proxies do not provide a solution for integrating all applications with object databases. However, we do think that proxies are applicable in other scientific domains and likely to be useful in areas beyond those. Indeed, the approach seems appropriate to any application areas where legacy programs have complex input and output structures, where computations are lengthy, where there are a variety of possible execution platforms and where it is desired to support intelligent interfaces for non-expert users. Because relatively powerful computers are becoming ubiquitous, and public communications networks are making even more powerful computers widely available, the use of computational tools is becoming widespread. Once usable only by experts in large relatively rich laboratories with considerable support staff, computational applications are becoming available to less well-endowed or to isolated researchers. Thus the use of existing, arcane computational tools and the demand for tools that make them easier to use are likely to increase — before the tools themselves become easy to use.

Below we offer advice to others embarking on projects similar to ours. We think this advice is relevant to those managing the evolution from legacy applications running stand-alone on heterogeneous platforms to an integrated domain-specific system of application services. We emphasize three aspects of our work as applicable to building infrastructure to support legacy systems and integrate applications or databases into a single framework:

1. The importance of a domain-level data model. As pointed out in Section 7.3 above, a common conceptual data model is critical to integrating diverse applications or existing schemas. Without agreement at the conceptual level, syntactic differences and format differences cannot be resolved. Even worse, any two applications may have subtle semantic differences that will render the comparison of results scientifically meaningless, even though that comparison may be plausible on the surface.

2. A clear division of labor between system components. Our infrastructure has defined one plausible division of labor among database, network services, user interface and application. The database should not only provide distributed database services but also support displaying and transforming data of different syntactic formats. The database can provide computation services as well, but these should be implemented in consort with a network service that spans the operating systems and architectures needed by and available to the computational scientist. The database is the logical place to maintain domain-specific information, including information about previous runs to pass to the network services (or the user) for scheduling decisions.

3. Our experience with object-oriented technology. We found that semantic data models do not capture behavioral aspects well, whereas an object-oriented framework gave us several mechanisms for coping with the complexity of our conceptual model. These mechanisms included grouping attributes into a single object, modeling behavioral aspects directly, using class hierarchies, and customizing abstract data types to a particular application.

   We also found object-oriented languages and database systems quite appropriate for expressing and implementing our information model. In particular, we could map conceptual level classes, operations and hierarchies directly into counterparts in the database's data definition language without a lot of encoding. The OODBS provided a data manipulation language that allowed us to implement many operations without recourse to an external application language. This behavioral capability is also useful for building routines to convert to and from particular formats. In spite of these advantages, however, we found several shortcomings in the current object-oriented modeling tools, languages and database systems during our work. These are summarized below in Section 7.5.

Inevitably, the demand for increasing the usability of computational tools will lead to the tools themselves being modified. While we do not expect that all the important legacy applications in any given domain will be modified to interface directly with a particular database system or even with a particular conceptual data model, building versions of those applications to read and write standard data structures requires somewhat less drastic changes. Indeed, data interchange formats for several scientific domains are being developed [21, 63, 129, 153, 183]. One could first use proxies to interface the database to standalone input and output files of major experiment types, and then build a proxy interface to read standard file interchange formats directly from the application. Thus proxies would offer a staged migration from standalone heterogeneous applications to a domain-specific database available to many applications.

7.5 Follow-on Work

This section identifies three areas of follow-on work suggested by our research: (1) engineering changes to the computational chemistry applications and to our own infrastructure, (2) engineering changes to the object-oriented development vehicles we used, and (3) research opportunities identified by our work. After discussing follow-on work we turn in Section 7.6 to the portions of it that we ourselves suggested by our research, intend to pursue as future work.

7.5.1 Engineering Changes to the Applications and Infrastructure

Work defining our infrastructure has led us to believe that the computational chemistry community should consider adopting a common object interchange schema standard rather than a common file interchange format. Given a common semantic-level definition of objects such as molecular structures or basis sets, syntactic variations among programs can be handled with syntactic transforms. If these applications write distinct *objects* as output rather than text, or in addition to it, the need for complex parsing of output text is alleviated.

We emphasize again here that the infrastructure we have designed is an application interface framework; it is not meant as an end-user interface to computational applications. An The end-user interface under development at PNL will provide a browser for our database and expert system advice for setting input parameters. We would also like to see improved support for registering new proxy types. An application registrar interface would provide a more effective way of loading application instances and writing templates than is now available.

In the course of evaluating our work, we observed that certain engineering changes to the design would render the infrastructure more effective in practise. Although some features of our conceptual model were not directly required in the implementation to demonstrate our research on computational support, they are needed for a practical system. These include substructure definition and structure searching, the generalization of the *property* class, facilities for experiment annotation and migration, and a closer integration of laboratory experiment to computational experiment. Security and accounting features must be added, and some benchmarks performed to determine if optimization of file transfer activities should be pursued. In addition to these enhancements to the database, the infrastructure's Compute Monitor could be rewritten to take advantage of future network services that (as they become available) will provide a higher abstraction of application process control than is now available. Finally, we recommend a further prototype implementation effort to write an input file generator as per our design and to refine the output file parser. This last task might uncover additional research questions.

7.5.2 Engineering Changes to Object-Oriented Tools

As we said above in Section 7.4, we were in general pleased with the choice of object-oriented technology as a vehicle for this research. However, object-oriented tools and technology are still in their infancy, and we have identified changes we think would facilitate efforts such as ours.

The first is improving the system design and conceptual modeling methodology that is available. As for the second, we recall for the reader our recommendations in Chapter 6 for improving the object-oriented language (C++) and database system (ObjectStore). Many of these desiderata fall into the category of providing a richer semantics or typing system. With a richer type system that allows for easily defining syntactic variants, we believe that some problems now classified as "schema integration" issues could be more simply resolved as "syntactic" differences.

Program generators for parsing external files and displaying database objects constitute our third recommendation for improving object-oriented development environments. While the parser generators and display functions specialized for scientific applications may be too specialized to warrant commercial development, general methods and tools would help many fields in addition to scientific domains. We believe that program generators specialized for creating textual renditions of data stored in a database can and should be developed. Such "layout languages" would provide in effect report writers that could generate formatted data from the database to be used for reports and papers, as well as input files. We emphasize that both kinds of program generators must associate objects in the database with new display formats for that data and must be easy enough for scientists to use without learning arcane syntax.

The fourth area ripe for object-oriented tool development is database support for mechanisms that would simplify the scientist's search space during experiment construction. We would like to see deductive query support available to an expert-system component of the user interface. A deductive query mechanism would complement the type extensibility features of object-oriented databases — a researcher could, we believe, use deductive mechanisms together with the object typing facility to enforce semantic integrity constraints at the experiment level. Deductive capabilities in the database would provide support for having one application input parameter constrain another, determining whether input parameters are compatible or extracting options from the database to fill certain input slots. For example, once the molecule that is the target of a computational experiment is chosen, the only sensible basis sets are those with basis functions for every atomic element in the molecule. Also, chemists need to query the domain database on the basis of approximate rather than exact matches. In advising a user on basis-set selection, we want to scan the database for "like experiments"; however, the measure of similarity needs to be type dependent, varying for molecule, chemical property, and so forth. We believe that a database query facility incorporating work on fuzzy sets could provide searches on "families" of molecules, for example.

The reification of computations, programs and resources as objects in the database expands the context in which deductive capabilities could be applied, enhancing the above possibilities. Given computations, programs and resources as objects, one can reason over events and resources.

7.5.3 Follow-on Research Opportunities

We suggest three categories of follow-on work with respect to the computational proxy itself: (1) extensions of the conceptual model, (2) enhancements to the proxy structure, and (3) use of the proxy structure to rationalize scheduling of resources in a distributed or parallel computing environment. Section 7.6 covers future research opportunities we ourselves will pursue.

1. Extensions of the conceptual model. The conceptual data model that we have developed is appropriate for ab initio computational chemistry. Extensions that would make the data model useful to other sub-domains include laboratory apparatus, semi-empirical computational applications and an extensible property class.

2. Enhancements to the proxy structure to manage series of experiments. In the computational chemistry domain (and in other scientific domains), scientists often want to perform collections of related runs where one or more parameters are varied over

some range. For example, one might vary the distance and orientation between two molecular fragments in order to generate an energy surface. The capability to set up an experiment "script" from which a series of experiments can be generated would be very helpful for such needs.

It seems a direct extension of our work to construct "meta-proxies" that represent collections of executions and implement experiment "scripting". Another common pattern of activity is stringing together computations by different programs, with the output of one being used as the input to the next. Modeling such chains of computations as sequences of proxies seems another natural enhancement. Clearly this would require the specification of a scripting language.

3. Enhancements to handle non-computational applications. Proxies are currently aimed at programs with a "batch" interface, where all input data are available at the start of a run, and output files are available at completion. If the proxy infrastructure could be extended to applications, such as molecular editors, that are more interactive than computational applications, their effectiveness as tools for computational science would be enhanced.

4. Use of the proxy structure to rationalize scheduling of resources in a distributed or parallel computing environment. There are two ways that proxies could help interfacing to distributed experiments or to experiments running on parallel computers:

   (a) Interface the proxy to be able to run an experiment on distributed or parallel systems. Once ab initio applications are rewritten so that a single experiment can be run on distributed or parallel systems, the proxy should be extended accordingly. Running ab initio applications on such systems will likely introduce additional complexity for the user such as specifying how to configure the parallel processors for a particular experiment. Extensions of the proxy infrastructure and user interface could do much to help in this regard. These extensions would not necessarily mean that the proxy need talk to multiple processes; in the future, it should be possible for the proxy to talk to a single point of control (a global scheduler).

(b) Providing information about likely resource needs of experiments by extrapolating from past resource use. As applications are rewritten to run on distributed or parallel systems, a global scheduler could make use of more information about resource requirements than users can now typically provide. Our study of the data structures and computations suggests that certain interesting characteristics of the computational process may be predictable from the input data, given an adequate sampling of previous computations on similar input data. Maintaining a database of previous runs, and "mining" those past experiments to provide hints from the proxy on setting up new experiments could prove useful if the proxy could pass such information to the applications.

7.6 Future Work

We decided to limit the scope of this dissertation to a particular sub-domain of chemistry because we felt that more could be accomplished initially by exploring one domain in detail than generalizing about many. We wanted to base our work on detailed empirical observations about the objects of interest (programs and data) in a particular sub-domain. However, we recognize that our contribution will be even more valuable if it is generalized to domains other than ab initio computational chemistry and to other target languages and database systems than C++ and ObjectStore. To that end, we are continuing our work by exploring whether (1) prospective or current users of Dr. Thomas Marr's genomic database application *Genome Topographer* also need access to computational biology applications, and (2) whether these applications have a profile sufficiently similar to ab initio computational chemistry applications to warrant use of the proxy infrastructure. If so, we shall explore generalizing the proxy infrastructure such that it could be implemented for *Genome Topographer*, implemented in the object-oriented database system GemStone. Developing a proxy for *Genome Topographer* would involve extending GemStone to provide computational services as we have extended ObjectStore. The extensions would allow for parameterized interface of the database to existing computation-intensive genomic applications running on a variety of platforms [41, 122].

The research we have proposed has two objectives: (1) To demonstrate that our work on computational proxies is generalizable to scientific domains other than computational chemistry, and serviceable as a model for the development of a database interface with so-called "legacy" applications and newer stand-alone computational programs running on a variety of computing platforms. (2) To make computational operations as well as genomic data available to molecular biology researchers.

In addition to our planned and funded work extending the idea of proxies to other domains, we note that our collaborators Dr. David Feller, Mr. D. Michael DeVaney, Ms. K. Schuchardt, Dr. Tom Keller, and others in the Molecular Science Research Center at Pacific Northwest Laboratory have embarked on a five-year implementation of a Computational Chemistry Database. We look forward to opportunities for further collaboration, to consider how to deploy the proxy mechanism in their ambitious effort, and to share further our own design efforts and implementation experiences. Computational scientists at the Molecular Science Research Center are also actively rewriting several computational chemistry applications to improve their performance. Dr. Thom Dunning has indicated interest in suggestions about building more flexible interfaces to those programs that we might offer from this work.

7.7 Final Remarks

We conclude that the proxy construct is an effective mechanism for solving the problems of program interoperability that plague computational science and for improving the accessibility of computational applications. While it is probably not possible to automate the construction of proxies for every parameter of every application in a given domain, it is possible to automatically generate interfaces for a significant and useful subset of experiment types. Our proxy infrastructure, when used in consort with a viable domain model, provides a viable migration path from standalone heterogeneous applications to a shared distributed database environment for computational science.

We are pleased to offer this contribution to the democratization of hitherto arcane and elusive yet powerful scientific tools. We are also convinced that the methodology we used in this research — detailed examination of the tools and data of a particular group of users — suggests a viable methodology for using empirical observation in computer science research. The "empirical" world where we gathered our primary data and that we used to corroborate our findings was a collection of artifacts, application programs, and data generated by those applications programs, as well as observations about the real-world use of those artifacts. We sincerely hope that our work will make that empirical world a better place for scientists to work.

Bibliography

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

[2] E. E. Abola, F. C. Bernstein, S. H. Bryant, T. F. Koetzle, and J. Weng. Protein data bank. In *Crystallographic Databases - Information Content, Software Systems, Scientific Applications*, pages 107–132. Data Commission of the International Union of Crystallography, Bonn/Cambridge/Chester, 1987.

[3] H. Afsarmanesh, D. McLeod, D. Knapp, and A. Parker. An extensible object-oriented approach to databases for VLSI-CAD. In *Proceedings of VLDB-85*, pages 13–24. Morgan Kaufmann, 1985.

[4] F. H. Allen. The cambridge structural database as a research tool in chemistry. In Z. B. Maksic, editor, *Modelling of Structure and Properties of Molecules*, pages 51–66. Horwood, 1987.

[5] M. Arango, D. Berndt, N. Carriero, D. Gelernter, and D. Gilmore. Adventures with Network Linda. *Supercomputing Review*, 3(10), October 1990.

[6] M. Atkinson, F. Bancihon, D. J. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database manifesto. In *Proc. of the Conference on Deductive and Object-Oriented Databases (DOOD)*, pages 40–57. Elsevier Science Publishers, 1990.

[7] K. Baclawski, R. Futrelle, C. Hafner, et al. Data and knowledge bases for biological papers and techniques. In W. W. Chu, A. F. Cardenas, and R. K. Taira, editors, *Proceedings of the AAAS Workshop on Advances in Data Management for the Scientist and Engineer — NSF Scientific Database Projects*, pages 23–28. Department

of Computer Science, University of California, Los Angeles, CA 90024, February 1993.

[8] R. A. Bair. Battelle Pacific Northwest Laboratory, Richland, WA. Personal communication, 1992.

[9] P. J. Barclay and J. B. Kennedy. Modelling ecological data. In H. Hinterberger and J. C. French, editors, *Sixth International Working Conference on Statistical and Scientific Database Management (SSDBM)*, pages 77–93. Departement Informatik, ETH, Zurich, June 1992.

[10] C. Baum, B. Balzer, et al. Domain specific software architectures – Command and control. In R. N. Taylor and W. L. Scherlis, editors, *DARPA Software Technology Conference*, pages 215–222, Los Angeles, 1992. Defense Advanced Research Projects Association, Arlington, VA, Meridian Corp.

[11] J. L. Bell. *Data Structures for Scientific Simulation Programs.* PhD thesis, University of Colorado, Boulder, CO, 1983.

[12] J. L. Bell and G. S. Patterson, Jr. Data organization in large numerical computations. *The Journal of Supercomputing*, 1:105–136, 1987.

[13] F. C. Bernstein, T. F. Koetzle, G. J. B. Williams, E. F. Meyer, Jr., M. D. Brice, J. R. Rodgers, O. Kennard, T. Shimanouchi, and M. Tasumi. The protein data bank: A computer-based archival file for macromolecular structures. *J. Mol. Bio.*, 112:535–542, 1977.

[14] P. A. Bernstein. Middleware: An architecture for distributed system services. Technical Report CRL 93/6, Cambridge Research Lab, Digital Equipment Corporation, Cambridge, MA 02139, March 1993.

[15] J. Biskup and H. H. Bruggemann. The personal model of data — towards a privacy oriented information system. Technical report, Hochschule Hildescheim, Hildescheim, West Germany, 1987.

[16] A. J. Bleasby and J. C. Wootton. Construction of validated, non-redundant composite protein sequence databases. *Protein Engineering*, 3(3):153–159, 1990.

[17] B. I. Blum. *Software Engineering, A Holistic View.* Oxford University Press, 1992.

[18] Computer Science and Technology Board. *Computing and Molecular Biology: Mapping and Interpreting Biological Information, a CSTB Workshop.* NRC, Washington, DC, 1990.

[19] G. Booch. *Object-oriented Design with Applications.* Benjamin Cummings, 1991.

[20] D. Bopegedera. The Evergreen State College, Olympia, WA. Personal Communication, 1992-1993.

[21] P. Bourne. *Proceedings of the First Macromolecular Crystallographic Interchange Format (CIF) Tools Workshop.* Sponsored by NSF at Columbia University, New York, October 15-17, 1993.

[22] C. Burks. Genbank: Current status and future directions. Technical Report LA-UR-89-1154, Los Alamos National Laboratory, Los Alamos, NM, April 1989.

[23] CAChe Scientific. *CAChe: Modeling for the Experimental Chemist.* Beaverton, OR, 1990.

[24] Cambridge Scientific Computing, Inc. *Chem3D.* Cambridge, MA, 1989.

[25] J. A. Campbell. *Chemical Systems.* Freeman, San Francisco, 1970.

[26] M. J. Carey et al. The EXODUS extensible DBMS project: An overview. Formerly unpublished technical report from Computer Sciences Department, University of Wisconsin, Madison, WI. In S. B. Zdonik and D. Maier, editors, *Readings in Object-oriented Database Systems*, pages 474–499. Morgan Kaufmann, 1990.

[27] R. G. G. Cattell. *Object Data Management.* Addison-Wesley, 1991.

[28] P. P. Chen. The entity-relationship model — toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[29] R. M. Chervin. High performance computing and the grand challenge of climate modeling. *Computers in Physics*, 4(3):234-238, May/June 1990.

[30] W. W. Chu, A. F. Cardenas, and R. K. Taira. A knowledge-based multimedia medical distributed database system — KMeD. In W. W. Chu, A. F. Cardenas, and R. K. Taira, editors, *Proceedings of the AAAS Workshop on Advances in Data Management for the Scientist and Engineer — NSF Scientific Database Projects*, pages 2–7. Department of Computer Science, University of California, Los Angeles, CA 90024, February 1993.

[31] W. W. Chu, I. T. Ieong, R. K. Taira, and C. M. Breant. A temporal evolutionary object-oriented data model and its query language. In *Proceedings of the Very Large Database Conference VLDB*, 1992.

[32] E. F. Codd. A relational model for large shared data banks. *CACM*, 13(6):377–387, 1970.

[33] E. F. Codd. *The Relational Model for Database Management Version 2.* Addison-Wesley, 1990.

[34] L. Coglianese, D. Batory, K. Bellman, D. Gries, D. McAllester, R. Selby, et al. An avionics domain-specific software architecture. In R. N. Taylor and W. L. Scherlis, editors, *DARPA Software Technology Conference*, pages 211–214, Los Angeles, 1992. Defense Advanced Research Projects Association, Arlington, VA, Meridian Corp.

[35] D. I. Cooke-Fox, G. H. Kirby, and J. D. Rayner. Computer translation of IUPAC systematic organic chemical nomenclature: Introduction and background to a grammar-based approach. *J. Chem. Info. Comp. Sci.*, 29:101–5, 1989.

[36] D. I. Cooke-Fox, G. H. Kirby, and J. D. Rayner. Computer translation of IUPAC systematic organic chemical nomenclature: Development of a formal grammar. *J. Chem. Info. Comp. Sci.*, 29:106–12, 1989.

[37] D. I. Cooke-Fox, G. H. Kirby, and J. D. Rayner. Computer translation of IUPAC systematic organic chemical nomenclature: Syntax analysis and semantic processing. *J. Chem. Info. Comp. Sci.*, 29:112–18, 1989.

[38] J. O. Coplien. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley, 1992.

[39] J. B. Cushing. 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.

[40] J. B. Cushing, D. Hansen, D. Maier, and C. Pu. Connecting scientific programs and data using object databases. *Bulletin of the Technical Committee on Data Engineering*, 16(1):9–13, March 1993.

[41] J. B. Cushing and E. Kutter. Computational proxies for genomic databases. The Evergreen State College, Olympia, WA, NSF proposal. Funded in August of 1993, September 1992.

[42] J. B. Cushing, D. Maier, and M. Rao. Computational chemistry database prototype. Technical Report CS/E-92-002, Department of Computer Science and Engineering, The Oregon Graduate Institute of Science & Technology, Portland, OR, 1991.

[43] J. B. Cushing, D. Maier, M. Rao, D. M. DeVaney, and D. Feller. Object-oriented database support for computational chemistry. In H. Hinterberger and J. C. French, editors, *Sixth International Working Conference on Statistical and Scientific Database Management (SSDBM)*, Zurich, June 1992. Departement Informatik, ETH, Zurich.

[44] E. R. Davidson and D. Feller. Basis set selection for molecular calculations. *Chemical Review*, 86:681–696, 1986.

[45] Daylight Chemical Information Systems. *The Daylight Toolkit.* Irvine, CA, 1991.

[46] L. Delcambre. Oregon Graduate Institute of Science & Technology, Portland, OR. Personal communication, 1994.

[47] J. Diederich and J. Milton. Creating domain specific metadata for scientific data and knowledge bases. *IEEE Transactions on Knowledge and Data Engineering*, 3(4):421–434, December 1991.

[48] J. Dozier. Looking ahead to EOS: The Earth Observing System. *Computers in Physics*, 4(3):248–259, May/June 1990.

[49] B. Dubrovsky. Universal data access for time series analysis. *Pixel*, pages 42–44, March/April, 1991.

[50] M. Dupuis. *HONDO-8 User's Guide.* IBM Center for Scientific and Engineering Computations, Kingston, NY, 1990.

[51] D. Feller. Computational Chemistry Input Assistant (CCIA) prototype. Extensional Computational Chemistry Group, Battelle Pacific Northwest Laboratory, Richland, WA, 1993.

[52] D. Feller. Battelle Pacific Northwest Laboratory, Richland, WA. Personal communication, December 1990–February 1995.

[53] D. Feller, C. M. Boyle, and E. R. Davidson. One-electron properties of several small molecules using near Hartree-Fock limit basis sets. *Journal of Chemical Physics*, 86(6):3424, 1987.

[54] D. Feller and E. R. Davidson. Basis sets for *ab initio* molecular orbital calculations and intermolecular interactions. In K. B. Lipkowitz and D. B. Boyd, editors, *Reviews in Computational Chemistry*, pages 1–43. VCH, 1990.

[55] D. Feller et al. *MELDFX User's Guide.* Molecular Science Research Center, Battelle Pacific Northwest Laboratory, Richland, WA, April 1991.

[56] T. E. Ferrin, B. S. Couch, C. C. Huang, E. F. Pettersen, and R. Langridge. An affordable approach to interactive desktop molecular modeling. *J. Mol. Graphics*, 9:27–32, March 1991.

[57] T. E. Ferrin, C. C. Huang, L. E. Jarvis, and R. Langridge. The MIDAS database system. *J. Mol. Graphics*, 6:2–12, March 1988.

[58] T. E. Ferrin, C. C. Huang, L. E. Jarvis, and R. Langridge. The MIDAS display system. *J. Mol. Graphics*, 6:13–27, March 1988.

[59] J. W. Fickett and C. Burks. Development of a database for nucleotide sequences. In M. S. Waterman, editor, *Mathematical Methods for DNA Sequences*, pages 1–35. CRC Press, 1990.

[60] J. C. French, A. K. Jones, and J. L. Pfaltz. NSF scientific database management workshop. Technical Report TR-90-21, University of Virginia, Charlottesville, VA, August 1990.

[61] J. C. French, A. K. Jones, and J. L. Pfaltz. NSF scientific database management workshop (panel reports and supporting materials). Technical Report TR-90-22, University of Virginia, Charlottesville, VA, August 1990.

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

[63] D. W. Fulker. Unidata strawman for storing earth-referencing data. *Proceedings of the Seventh International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology*, pages 210–217, 1991.

[64] GAMESS user's guide. Department of Chemistry, North Dakota State University, Fargo, ND, 1990.

[65] E. Garfield. Chemico-linguistics: Computer translation of chemical nomenclature. *Nature*, 192:192, 1961.

[66] E. Garfield. Manfred Kochen: In memory of an information scientist pioneer qua world brain-ist. *Current Contents*, 25(JUN):3–14, June, 1989.

[67] G.A. Geist and V. S. Sunderam. Network based concurrent computing on the PVM system. *Concurrency: Practice and Experience*, 4(4):315–339, June 1992.

[68] S. P. Ghosh. Statistical relational model. In M. Rafanelli, J. C. Klensin, and P. Svensson, editors, *Fourth International Working Conference on Statistical and Scientific Database Management (SSDBM)*, pages 338–355. LNCS 339, Springer-Verlag, June 1988.

[69] H. H. Goldstine. *The Computer from Pascal to Von Neumann.* Princeton University Press, 1993.

[70] N. Goodman. An object-oriented DBMS war story: Developing a genome mapping database in C++. In W. Kim, editor, *Modern Database Management: Object-Oriented and Multidatabase Technologies.* ACM Press, New York, 1994.

[71] P. M. D. Gray, N. W. Paton, G. J. L. Kemp, and J. E. Fothergill. An object-oriented database for protein structure analysis. *Protein Engineering*, 3(4):235–243, 1990.

[72] T. P. Green and J. Snyder. DQS: A Distributed Queuing System. Technical report, Supercomputer Computations Research Institute, Florida State University; Pittsburgh Supercomputing Center; Center for High Performance Computing, The University of Texas System, March 1993.

[73] J. Gribbon. *In Search of Schrödinger's Cat.* Bantam, 1984.

[74] H. Gunadhi and A. Segev. Temporal query optimization in scientific databases. *Data Engineering (Special Issue on SSDBMS)*, 13(3):27–34, September 1990.

[75] R. Gupta and E. Horowitz, editors. *Object-oriented Databases with Applications to CASE, Networks, and VLSI/CAD.* Prentice Hall, 1991.

[76] L. M. Haas et al. Starburst in mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990.

[77] E. Haber, Y. Ioannidis, and M. Livny. Foundations of visual metaphors for schema display. *Journal of Intelligent Information Systems*, 3(3/4):263–298, July 1994.

[78] D. Hansen. Oregon Graduate Institute of Science & Technology, Portland, OR. Personal communication, 1994.

[79] D. Hansen, D. Maier, J. Stanley, and J. Walpole. Object-oriented heterogeneous database for materials science. *Scientific Programming*, 1(2):115–131, 1992.

[80] D. M. Hansen and D. Maier. Using an object-oriented database to encapsulate heterogeneous scientific data sources. In J. F. Nunamaker and R. H. Sprague, editors, *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, pages 408–417. IEEE Computer Society Press, January 1994.

[81] K. A. Healy et al. IDEF1X working group, IEEE draft standard 1320.2, 1994.

[82] S. R. Heller. The chemical information system and spectral databases. *J. Chem. Inf. Comput. Science*, 25:224–231, 1985.

[83] S. R. Heller, editor. *The Beilstein OnLine Database.* Symposium Series 436, ACS, 1990.

[84] C. Hightower and R. Schwarzwalder. A comprehensive look at materials science databases. *DATABASE*, 14(3):42–53, April 1991.

[85] W. D. Hillis and G. Steele, Jr. Data parallel algorithms. *CACM*, 29(12):1170–1183, December 1986.

[86] B. Hodges and C. Thompson. Texas Instruments workshop on application integration architectures. Technical report, National Institute of Standards and Technology Publication, February 8-12, 1993.

[87] C. C. Huang, E. F. Pettersen, T. E. Klein, T. E. Ferrin, and R. Langridge. Conic: A fast renderer for space-filling molecules with shadows. *J. Mol. Graphics*, 9:230–236, December 1991.

[88] Z. Huang, P. Svensson, and H. Hauska. Solving spatial analysis problems with GeoSAL, a spatial query language. In H. Hinterberger and J. C. French, editors, *Sixth International Working Conference on Statistical and Scientific Database Management (SSDBM)*, pages 1–17. Departement Informatik, ETH, Zurich, June 1992.

[89] IBM Visualization Systems. *IBM Visualization Data Explorer*. Thomas J. Watson Research Center, Yorktown Heights, NY, 1994.

[90] H. V. Jagadish. Incorporating hierarchy in a relational model of data. *Proceedings ACM SIGMOD*, 18(2):78–87, June 1989.

[91] R. Jain. Workshop report on NSF workshop on visual information management systems. Technical report, Computer Science and Engineering Division, University of Michigan, Ann Arbor, MI, 1993.

[92] G. A. Jirak. Aurora Dataserver, a data management system for visualization applications. *IEEE Visualization '93*, 1993.

[93] S. Jobs. Openstep: A NeXt computer product. Panel on Interconnectivity, *Object-World*, San Francisco, 1994.

[94] T. Joseph and A. F. Cardenas. PICQUERY: A high level query language for pictorial database management. *IEEE Transactions on Software Engineering*, 14(5):630–638, 1988.

[95] H. K. Kaindl. An integrated information system for the bench chemist. *Chemical Information: Information in Chemistry, Pharmacology and Patents*, pages 63–70, September 1989.

[96] W. J. Kaufmann, III. *Supercomputing and the Transformation of Science*. Scientific American Library, 1993.

[97] T. Kazic and S. Tsur. Modeling and simulating biological processes as logical enterprises. In W. W. Chu, A. F. Cardenas, and R. K. Taira, editors, *Proceedings of the AAAS Workshop on Advances in Data Management for the Scientist and Engineer — NSF Scientific Database Projects*, pages 16–22. Department of Computer Science, University of California, Los Angeles, CA 90024, February 1993.

[98] T. Keller. Battelle Pacific Northwest Laboratory, Richland, WA. Personal communication, 1992.

[99] B. W. Kernighan and D. M. Richie. *The C Programming Language.* Prentice-Hall, 1978.

[100] J. C. Klensin and R. M. Romberg. Statistical data management requirements and the SQL standards. In M. Rafanelli, J. C. Klensin, and P. Svensson, editors, *Fourth International Working Conference on Statistical and Scientific Database Management (SSDBM)*, pages 19–38. LNCS 339, Springer-Verlag, June 1988.

[101] H. F. Korth and A. Silberschatz. *Database System Concepts.* McGraw Hill, 1986.

[102] J. C. Kotz and K. F. Purcell. *Chemistry and Chemical Reactivity.* Saunders, 1987.

[103] J. K. Labanowski. List Coordinator for the Computational Chemistry Electronic Bulletin Board (CHEMISTRY-REQUEST@osc.edu), Ohio Supercomputer Center, 1224 Kinnear Rd, Columbus, OH 43212-1163.

[104] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *CACM*, 34(10):50–63, October 1991.

[105] E. Lander, R. Langridge, and D. Saccocio. Computing in molecular biology: Mapping and interpreting biological information. *IEEE Computer*, 24(11):6–13, November 1991.

[106] E. Lang, T. Forster, and C. von der Lieth. The integration of the Cambridge crystallographic data files into the relational information network of the German cancer research center. In J. Gasteiger, editor, *Software Development in Chemistry 4, Proceedings of the 4th Workshop on Computers in Chemistry*, pages 43–50. Springer-Verlag, November 1989.

[107] R. H. Lathrop, T. A. Webster, and T. F. Smith. ARIADNE: Pattern-directed inference and hierarchical abstraction in protein structure recognition. *CACM*, 30(11):909–921, November 1987.

[108] A. J. Lawson. The Lawson similarity number. In S. R. Heller, editor, *The Beilstein OnLine Database*, pages 143–155. Symposium Series 436, ACS, 1990.

[109] J. Lederberg. Digital communications and the conduct of science: The new literacy. *Proc. IEEE*, 66:1314–1319, 1978.

[110] S. Letovsky, R. Pecherer, and A. Shoshani. Scientific data management for human genome applications. *Data Engineering (Special Issue on SSDBMS)*, 13(3):51, September 1990.

[111] I. N. Levine. *Quantum Chemistry*. Allyn and Bacon, 1983.

[112] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[113] P. Lyngbaek and D. McLeod. A personal data manager. *Proceedings of the International Conference on Very Large Databases*, August 1984.

[114] R. Lysakowski, editor. *First International Symposium on Computerized Chemical Data Standards: Databases, Data Interchange, and Information Systems, May 5-7, 1993*. STP 1214, American Society for Testing and Materials (ASTM), 1994.

[115] P. Machin. Programming aspects of crystallographic data files: Interactive retrieval from the cambridge database. In G. M. Sheldrick, C. Kruger, and R. Goddard, editors, *Crystallographic Computing 3: Data Collection, Structure Determination, Proteins, and Databases*, pages 106–118. Clarendon Press, Oxford, UK, 1985.

[116] D. Maier. Why isn't there an object-oriented data model? In G. X. Ritter, editor, *IFIP 11th World Computer Congress – Information Processing '89*, pages 793–798. Elsevier Science, August – September 1989.

[117] D. Maier and J. B. Cushing. Treating programs as objects: The computational proxy experience. Invited paper. In *Conference on Deductive and Object-Oriented Database (DOOD'93)*, pages 1–12. LNCS 760, Springer-Verlag, December 1993.

[118] D. Maier, J. B. Cushing, D. Hansen, M. Rao, et al. Object data models for shared molecular structures. In R. Lysakowski, editor, *First International Symposium on*

Computerized Chemical Data Standards: Databases, Data Interchange, and Information Systems. STP 1214, American Society for Testing and Materials (ASTM), 1994.

[119] D. Maier, P. Nordquist, and M. Grossman. Displaying database objects. In L. Kerschberg, editor, *Proceedings of the 1st International Conference on Expert Database Systems*, pages 59–74. Benjamin-Cummings, 1987.

[120] D. Maier and J. Stein. Development and implementation of an object-oriented DBMS. In P. Wegner and B. Shriver, editors, *Research Directions in Object-Oriented Programming*, pages 355–392. MIT Press, 1987.

[121] H. Maier and D. Walkowisk. Chemical substructure search on CD-ROM. In J. Gasteiger, editor, *Software Development in Chemistry 4, Proceedings of the 4th Workshop on Computers in Chemistry*, pages 1–9. Springer-Verlag, November 1989.

[122] T. Marr. The Genome Topographer, Cold Spring Harbor Laboratory, Cold Spring Harbor, NY. Personal Communication, 1994.

[123] D. McLean and E. Replogle. Archem. Technical report, IBM Almaden, 1992.

[124] D. McLeod. *High Level Expression of Semantic Integrity Specifications in a Relational Data Base System.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA 02139, 1977.

[125] W. Moor. *Schrödinger: Life and Thought.* Cambridge University Press, 1992.

[126] B. Moreland. Microsoft's Cairo, a distributed operating system product. Microsoft Corporation, Bellevue, WA. Personal comminication., 1995. See also Sun Microsystems' satellite broadcast, "Sunergy #10," of *Object World* Panel on Distributed Systems, July 28, 1994, Mountain View, CA (URL: http://www.sun.com/sunergy/).

[127] R. S. Mullikan. Spectroscopy, molecular orbitals, and chemical bonding. *Scientific American*, 157(3784):13–24, July 7 1967.

[128] N. Nadkarni and G. Parker. A profile of forest canopy science and scientists - who we are, what we want to know, and obstacles we face: Results of an international survey. *Selbyana*, 15:38–50, 1994.

[129] National Institutes of Health. ASN — An interchange format for genomic sequence data. Bethesda, MD, 1994.

[130] A. Nebel, G. Olbrich, and R. Deplanque. The Gmelin information system: The connection between handbook and database. In J. Gasteiger, editor, *Software Development in Chemistry 4, Proceedings of the 4th Workshop on Computers in Chemistry*, pages 51–56. Springer-Verlag, November 1989.

[131] J. R. Nicol, C. T. Wilkes, and F. A. Manola. Object orientation in heterogeneous distributed computing systems. *IEEE Computer*, 26(6):57–67, June 1993.

[132] Object Design. *ObjectStore*. Burlington, MA, 1991.

[133] T. O'Brien. Hewlett-Packard's client/server strategy. *Distributed Computing Monitor*, 9(4):3–23, 1994.

[134] Office of Standards and Technology. Implementation of the flexible image transport system. Technical report, NASA/OSSA, November 1991.

[135] F. Olken. Physical database support for scientific and statistical database management. Technical Report LBL-19940 (rev2), Lawrence Berkeley Laboratories, Berkeley, CA, May 1986.

[136] F. Olken. Scientific and statistical data management research at LBL. Technical Report LBL-21623, Lawrence Berkeley Laboratories, Berkeley, CA, June 1986.

[137] J. A. Orenstein. Object Design, Inc., Burlington, MA. Personal Communication, 1993.

[138] N. W. Paton and P. M. D. Gray. Identification of database objects by key. In *Advances in Object-Oriented Database Systems*, pages 280–285. LNCS 334, Springer-Verlag, 1988.

[139] N. W. Paton and P. M. D. Gray. Optimising and executing DAPLEX queries using Prolog. *The Computer Journal*, 33(6):547–555, 1990.

[140] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci. (U.S.)*, 85(8):2444–2448, April 1988.

[141] R. Pecherer, R. Robers, R. Olken, and R. Robbins. Goals and objectives of the human genome project (panel discussion). In *Proceedings ACM SIGMOD*, May 1990.

[142] G. M. Pesyna. *Computerized Structure Retrieval and Interpretation of Mass Spectra: The Design and Evaluation of a Probability Based Matching System Using a Large Data Base.* PhD thesis, Cornell University, Ithaca, NY, 1975.

[143] R. Pike et al. Plan 9 from Bell Labs. *Research Note, Bell Laboratories*, July 1990.

[144] G. Popek and B. Walker. *The LOCUS Distributed System Architecture.* MIT Press, Cambridge, 1985.

[145] J. A. Pople and D. L. Beveridge. *Approximate Molecular Orbital Theory.* McGraw-Hill, 1970.

[146] C. Pu, K. P. Sheka, L. Chang J. Ong, A. Chang, E. Alessio, I. N. Shindyalov, W. Chang, and P. E. Bourne. PDBtool: A prototype object oriented toolkit for protein structure verification. Technical Report CUCS-048-92, Department of Computer Science, Columbia University, New York, 1992.

[147] G. D. Purvis, III. CAChe Scientific, Inc., Beaverton, OR. Personal Communication, 1993.

[148] K. Qiu, N. I. Hachem, M. O. Ward, and M. A. Gennert. Providing temporal support in database management systems for global change research. In H. Hinterberger and J. C. French, editors, *Sixth International Working Conference on Statistical and Scientific Database Management (SSDBM)*, pages 274–289. Departement Informatik, ETH, Zurich, June 1992.

[149] M. Rafanelli and F. L. Ricci. A visual interface for statistical entities. *Data Engineering (Special Issue on SSDBMS)*, 13(3):35–44, September 1990.

[150] M. Rafanelli and A. Shoshani. STORM: A statistical object representation. *Data Engineering (Special Issue on SSDBMS)*, 13(3):12–18, September 1990.

[151] M. Rao. Computational proxies for computational chemistry: A proof of concept. Master's thesis, Department of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, Portland, OR, to be published, 1995.

[152] E. Replogle. Corporate Research Laboratory, IBM, Almaden, CA. Personal Communication, 1992.

[153] R. K. Rew and G. P. Davis. NetCDF: An interface for scientific data access. *IEEE Computer Graphics & Applications*, 10(4):76–82, July 1990.

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

[155] L. Rosenblum. Scientific visualization research laboratories. *IEEE Computer*, 22(8):68–101, August 1989.

[156] L. A. Rowe and M. Stonebraker. The Postgres data model. In *Proceedings of VLDB-87*, pages 461–473. Morgan Kaufmann, 1987.

[157] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modeling and Design.* Prentice Hall, 1991.

[158] J. R. Rumble, Jr. and F. J. Smith. *Database Systems in Science and Engineering.* Adam Hilger, 1990.

[159] L. Salem. *Marvels of the Molecule.* VCH, 1987.

[160] M. Santori. An instrument that isn't really. *IEEE Spectrum*, 27(8):36–39, August 1990.

[161] K. Schuchardt. Battelle Pacific Northwest Laboratory, Richland, WA. Personal communication, 1992.

[162] A. Segev and A. Shoshani. The representation of a temporal data model in the relational environment. In M. Rafanelli, J. C. Klensin, and P. Svensson, editors, *Fourth International Working Conference on Statistical and Scientific Database Management (SSDBM)*, pages 39–61. LNCS 339, Springer-Verlag, June 1988.

[163] Servio Corporation. *GemStone Programming Guide.* Portland, OR, June 1994.

[164] L. G. Shapiro and S. L. Tanimoto. Panel on visual database systems. In *IEEE Symposium on Visual Programming Languages*, pages 62–71, August 1993.

[165] L. G. Shapiro, S. L. Tanimoto, J. F. Brinkley, J. Ahrens, R. M. Jakobovits, and L. M. Lewis. A visual database system for data and exeriment management in model-based computer vision. In *Proceedings of the Second CAD-Based Vision Workshop*, pages 64–72, February 1994.

[166] A. Shoshani, F. Olken, and H. K. T. Wong. Characteristics of scientific databases. In *Proceedings of the Tenth International Conference on VLDB*, pages 147–159, August 1984.

[167] N. C. Shu, B. C. Housel, R. W. Taylor, S. P. Ghosh, and V. Y. Lum. EXPRESS: A data EXtraction, Processing, and REStructuring System. *ACM Transactions on Database Systems*, 2(2):134–174, June 1977.

[168] M. J. Sienko and R. A. Plane. *Experimental Chemistry (6th edition).* McGraw-Hill, 1984.

[169] J. M. Smith and D. C. P. Smith. Database abstractions: aggregation and generalization. *ACM Transactions on Database Systems*, 2(2):105–133, 1977.

[170] E. Soloway, W. Martin, and K. Abotel. Computer-based support for scientific data analysis. In W. W. Chu, A. F. Cardenas, and R. K. Taira, editors, *Proceedings of*

the AAAS Workshop on Advances in Data Management for the Scientist and Engineer — NSF Scientific Database Projects, pages 119–124. Department of Computer Science, University of California, Los Angeles, CA 90024, February 1993.

[171] J. F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine.* Addison-Wesley, 1984.

[172] T. M. Sparr, R. D. Bergeron, L. D. Meeker, N. Kinner, P. Mayewski, and M. Person. Integrating data management, analysis and visualization for collaborative scientific research. Technical Report 91-10, University of New Hampshire, Durham, NH 03824, May 1991.

[173] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1991.

[174] A. Szabo and N. S. Ostlund. *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory.* McGraw-Hill, 1989.

[175] Fred Tabbutt. The Evergreen State College, Olympia, WA. Personal Communication, 1992-1993.

[176] A. Tanenbaum. *Operating Systems: Design & Implementation.* Prentice-Hall, 1987.

[177] D. Tasker. The problem space e-book and the model 5w CASE tool. Self-published, Sydney, Australia, 1994.

[178] R. N. Taylor and W. L. Scherlis, editors. *DARPA Software Technology Conference*, Los Angeles, 1992. Defense Advanced Research Projects Association, Arlington, VA, Meridian Corp.

[179] M. A. Thompson. Battelle Pacific Northwest Laboratory, Richland, WA. Personal communication, 1992.

[180] J. M. Thornton and S. P. Gardner. Protein motifs and database searching. *Trends in Biochemical Sciences (TIBS)*, 14(7):300–304, July 1989.

[181] D. Tsichritzis and A. Klug. The ANSI-X3-SPARC DBMS framework. *Information Systems*, 3:173–191, 1978.

[182] D. C. Tsichritzis and F. H. Lochovsky. *Data Models.* Prentice-Hall, 1982.

[183] Unidata Program Center. *NetCDF User's Guide, v 1.11.* University Corporation for Atmospheric Research, Boulder, CO, March 1991.

[184] C. Upson, T. A. Faulhaber, Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam. The Application Visualization System: A computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, 9(4), July 1989.

[185] L. Wall and R. L. Schwartz. *Programming perl.* O'Reilly & Associates, 1990.

[186] J. Walpole. Oregon Graduate Institute of Science & Technology, Portland, OR. Personal communication, 1995.

[187] S. S. Walther and R. L. Peskin. Object-oriented visualization of scientific data. *Journal of Visual Languages and Computing*, 2(1):43–56, 1991.

[188] M. Waterman. Foreword. *Bulletin of Mathematical Biology*, 51(1):1–4, 1989.

[189] A. Weinand. Taligent, Inc., Cupertino, CA 95014-2233. Personal communication, 1995. See also URL: http://www.taligent.com.

[190] R. F. E. Weissman. In search of the scholar's workstation: Recent trends and software challenges. *Academic Computing*, 3:28–31, 59–64, September 1989.

[191] A. J. Westlake and I. Kleinschmidt. The implementation of area and membership retrievals in point geography using SQL. *Data Engineering (Special Issue on SS-DBMS)*, 13(3):4–11, September 1990.

[192] T. Winograd and F. Flores. *Understanding Computers and Cognition.* Ablex, 1986.

[193] R. Wirfs-Brock. *Designing Object-Oriented Software.* Prentice-Hall, 1990.

[194] J. L. Wisniewski, L. Goebels, and A. Lawson. AUTONOM: Automatic generation of IUPAC-names from structural input. In J. Gasteiger, editor, *Software Development*

in Chemistry 4, Proceedings of the 4th Workshop on Computers in Chemistry, pages 19–29. Springer-Verlag, November 1989.

[195] E. Zass. Reaction databases in a university chemistry department – online or in-house? In J. Gasteiger, editor, *Software Development in Chemistry 4, Proceedings of the 4th Workshop on Computers in Chemistry*, pages 243–253. Springer-Verlag, November 1989.

[196] S. B. Zdonik and D. Maier, editors. *Readings in Object-oriented Database Systems*. Morgan Kaufmann, 1990.

Biographical Note

Judith Bayard Cushing completed the research reported in this dissertation while on leave from her position as Member of the Faculty at The Evergreen State College. Evergreen is an innovative state-supported four-year liberal arts college located in Olympia, Washington, whose curriculum is organized primarily as full-time, year-long, team-taught interdisciplinary programs of study. There, Ms. Cushing developed the software enginnering and database program *Student Originated Software*, known from 1983 to 1987 as *The Business of Computers*. Needless to say, this program contains a considerable database component. Ms. Cushing also teaches general computer science courses, and particularly enjoys teaching a first programming course based on William Clayson's work on visual design using Logo. She has organized interdisciplinary colloquia on computing and social responsibility, cognitive science, scientific applications, and computing and the arts.

Before coming to Evergreen in 1982, she worked for 15 years in a variety of professional capacities in information systems, including as systems and application programmer, systems engineer, and computer center director. She has worked for large multinational companies (IBM and Texas Instruments), universities (Université de Bordeaux III, Cornell, and Southwestern Medical School), and small start-up firms (Compagnie Internationale de TéléInformatique in Paris, France, and the Public Health Automated Medical Information System (PHAMIS) in Seattle). Her undergraduate work at The College of William and Mary in Williamsburg, Virginia, focused on mathematics until she discovered the Greek philosophers. She continued to study philosophy at Brown University in Providence, Rhode Island, completing a master's thesis in June of 1969 in the philosophy of science.

As a result of four years working with Prof. David Maier at the Oregon Graduate

Institute, Ms. Cushing currently concentrates teaching and research on the software engineering of distributed scientific applications and databases. Specific application areas of interest include computational chemistry, molecular biology, geographical information systems and forest canopy spatial structures, and she is principal investigator or co-principal investigator for National Science Foundation and Murdock Charitable Trust grants in these areas. Ms. Cushing has a particular concern for developing and maintaining quality computer science education at four-year liberal arts colleges, and is a principal investigator for a three-year National Science Foundation educational infrastructure effort to integrate recent computer science research results into an interdisciplinary undergraduate curriculum.

Married to fellow computer scientist and Evergreen faculty member John Aikin, Judith Bayard took with him in 1988 the name "Cushing". Together they often hike both the Pacific Northwest and the American Southwest, and enjoy downhill and cross-country skiing anywhere they can find good snow. Each summer they raise more raspberries than they and their friends can eat. This thesis completed, Ms. Cushing intends to rekindle her passion for cooking foods of China, France and India, and to finally learn to bake sourdough bread. She hopes to log more hiking and skiing miles than had been possible while commuting between the Rose City and Olympia, finally join the Town Cats Knitting Circle, and plant some roses.

Ms. Cushing is a member of the ACM, the IEEE, and Sigma Xi; she was born Judith Ellen Johnson, in Baltimore, Maryland, USA, June 1, 1946.