The Parser Converter Loader:

An Implementation Of The Computational Chemistry Output Language (CCOL)

ABSTRACT

A necessity of managing scientific data is the ability to maintain experimental legacy information without continually modifying the applications that create and use the information. By facilitating the management of scientific data we hope to give scientist the ability to effectively use additional modeling applications and experimental data. We have demonstrated that an extensible interpreter, using a series of stored directives, allows the loading of data from computational chemistry applications into a generic database. Extending the interpreter to support a new application involves adding a list of directives for each piece of information to be loaded. This research confirms that an extensible interpreter can be used to load computational chemistry experimental data into a generic database. This procedure may be applicable to the loading and retrieving of other types of experimental data without requiring modifications of the loading and retrieving applications.

The Parser Converter Loader:

An Implementation Of The Computational Chemistry Output Language (CCOL)

By: Donald Randall Abel

(drabel@cs.pdx.edu)

January 19, 1995, Revision 9

Contents

List Of Figures

1Introduction

The primary focus of our work, the PCL interpreter, is part of an overall project to assist computational chemists with data and experiment management. Section 1.1 of the introduction surveys the overall problem. Section 1.2 surveys CCDB, which is our proposed solution to the problem. Section 1.3 describes the primary contribution of this thesis, namely the PCL interpreter.

1.1Problem Overview

The objective of the Computational Chemistry Database project (CCDB) at the Oregon Graduate Institute is to assist computational chemists with the management of data and experiments. This work is being led by Judith Cushing under the direction of David Maier with the assistance of Meenakshi Rao, and the author. Additional data on the CCDB project can be found in [2] and [5]. This objective is worthwhile because it will promote the leveraging of past research in future exploration. CCDB specifically addresses the difficulties of computational chemists. However, problems of the other computational sciences are similar.

Computational chemists mathematically model the characteristics of molecules. They accomplish this with powerful computers and specialized software. Results from promising experiment runs are implemented in the laboratory. Laboratory results are compared to the optimized molecular configuration predicted by the model. Computational chemists speed the development of new experiments by examining the optimized molecular configurations from experiment runs. Figure 1-1 shows the components found in a computational chemistry experiment.

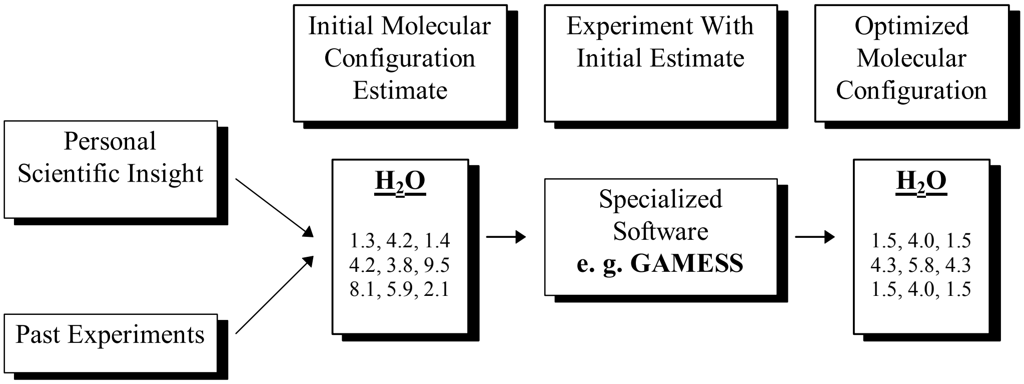


Figure 1-1 Components Of A Computational Chemistry Application

A molecular configuration is a set of parameters that mathematically describe a molecule and includes data on where specific atoms are located in the molecule and the nature of their bonding. A chemist estimates an initial molecular configuration either through personal scientific insight, or by referring to molecular configurations from past experiment runs. Chemists then submit these estimates to modeling experiment programs.

Computational Chemistry Modeling programs, such as GAMESS, Gaussian, and HONDO, use the molecular estimate to optimize the molecular configuration through iteration. Iteration is the process of repeatedly applying a calculation to an approximation in order to calculate a successively closer approximation. Each iteration step is referred to as an iteration. This process is shown in Figure 1-2. Under favorable circumstances the calculation will converge. In unfavorable situations the estimates will diverge. The process of having an experiment adequately optimize an initial molecular configuration is called an experiment run.

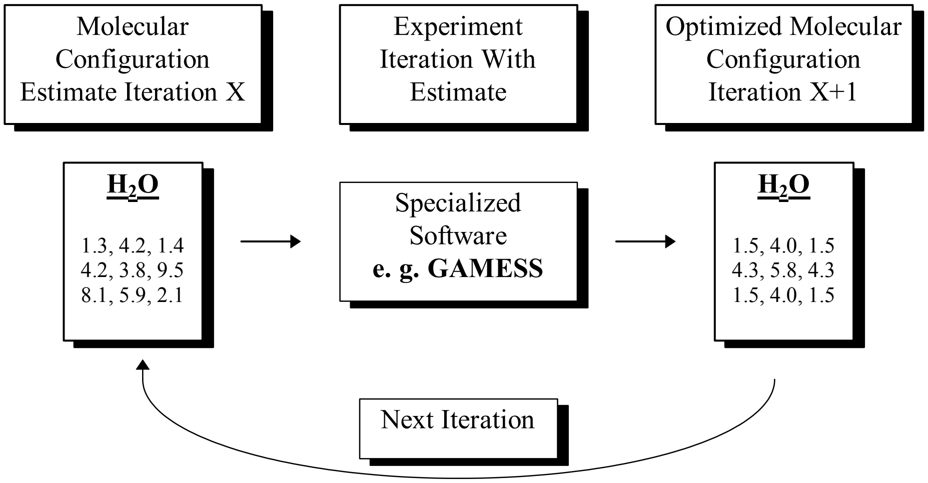


Figure 1-2 One Experiment Iteration

The process of performing an experiment is elaborate and involves numerous steps. We illustrate these steps in Figure 1-3. Before optimizing a molecular configuration estimate the computational chemist must select a program on which to perform the experiment. There are numerous programs that model molecules. Each program has features that distinguish it from other programs. Some systems quickly calculate the total energy of the molecule but run more slowly. Other programs have extremely accurate calculations. The scientist may, for example, first want to produce a experiment run with GAMESS and review the total energy results before calculating an optimized molecular configuration with the program HONDO. Our example will involve a experiment run of the molecule Ethylene and the GAMESS application.

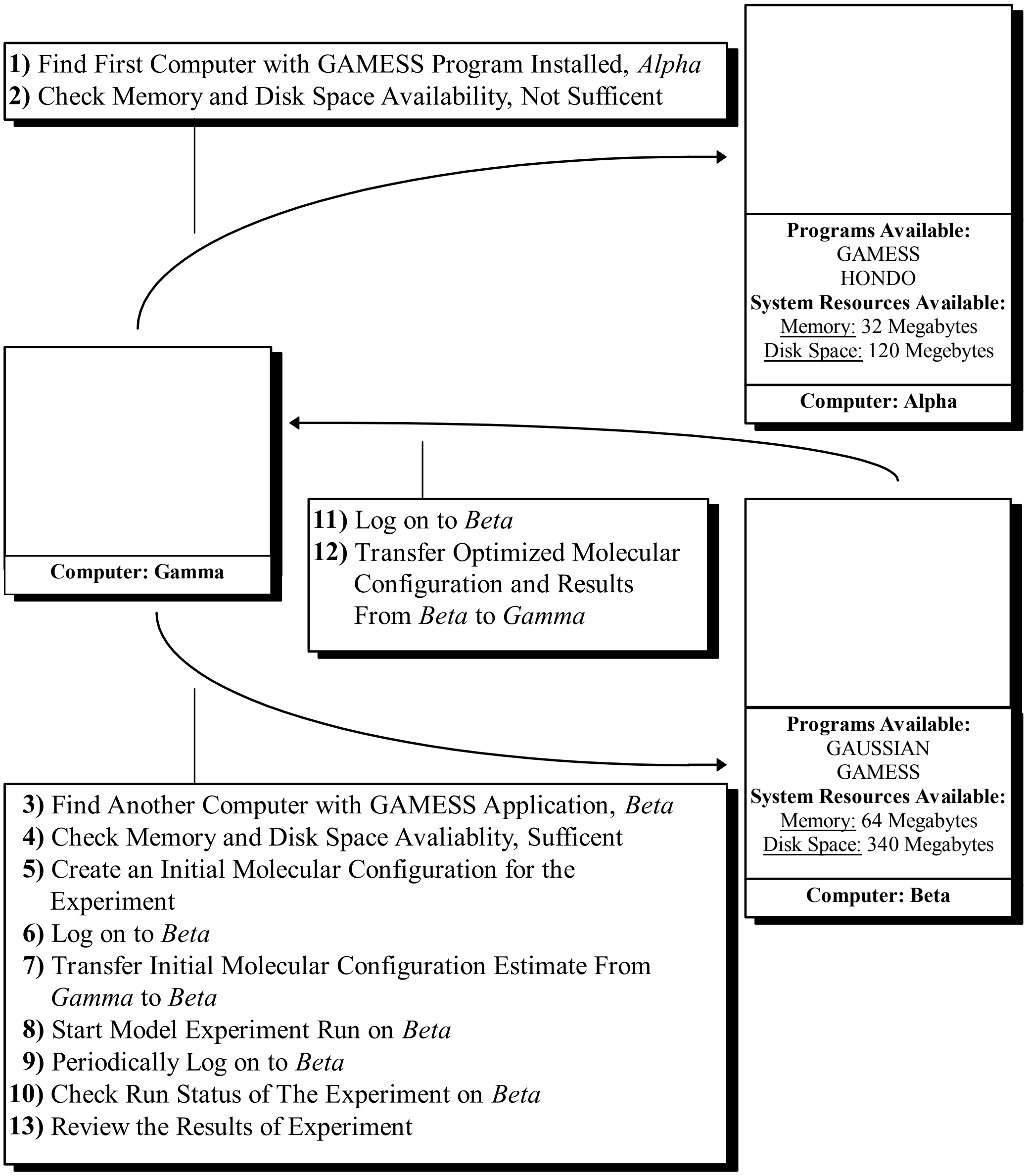

1) Find First Computer with GAMESS Program Installed, Alpha
2) Check Memory and Disk Space Availability, Not Sufficent
Programs Available:
GAMESS
HONDO
System Resources Available:
Memory: 32 Megabytes
Disk Space: 120 Megebytes
Computer: Alpha
Computer: Gamma
11) Log on to Beta
12) Transfer Optimized Molecular Configuration and Results From Beta to Gamma
Programs Available:
GAUSSIAN
GAMESS
System Resources Available:
Memory: 64 Megabytes
Disk Space: 340 Megebytes
Computer: Beta
3) Find Another Computer with GAMESS Application, Beta
4) Check Memory and Disk Space Avaliablity, Sufficent
5) Create an Initial Molecular Configuration for the Experiment
6) Log on to Beta
7) Transfer Initial Molecular Configuration Estimate From Gamma to Beta
8) Start Model Experiment Run on Beta
9) Periodically Log on to Beta
10) Check Run Status of The Experiment on Beta
13) Review the Results of Experiment

Figure 1-3 Example Steps Of An Experimental Run

The first step of an experiment involves locating a computer that has the necessary software, in our example GAMESS. In our example search for a system, we find that the computer named *Alpha* has GAMESS installed. Finding a computer that has the required program installed does not necessarily mean that the experiment run can be performed on that computer. The computational scientist must estimate the amount of disk space and memory required to perform the run. There are several factors that influence the amount of memory and disk space required: the program, the type of experiment run, the molecule being modeled, and the experiment input parameters. As with the molecular configuration estimates, a chemist can either arrive at the memory and disk space requirements through personal scientific insight or by referring to past experiments. This determination is shown as step two in Figure 1-3. By referring to past experiments we estimate that ours will require 48 Megabytes of memory and 8 Megabytes of disk space for results. Note that the computer *Alpha* had the GAMESS program installed, but, did not have enough memory to perform the run. In our example another computer needed to be located which had the GAMESS application installed, 48 Megabytes of memory, and 8 Megabytes of disk space. The computer named *Beta* is found to fulfill these requirements in steps three and four.

If all these requirements are confirmed, the molecular configuration estimate must be created and transferred to the computer that is performing the experiment run. In the example this requires logging in to the computer named *Beta* and transferring the molecular configuration estimate for the molecule Ethylene. In the example these are steps five, six, and seven. The molecular configuration estimate that is transferred contains data about the molecule being modeled. This data can be several megabytes in size. It includes estimates on where specific electrons are located in the molecule.

The programs that model a molecule use this data to calculate characteristics of the molecule and the iterative modeling can continue for days. The molecular model run can now be started with the molecular configuration estimate. The GAMESS program is started with the Ethylene molecular configuration estimate in step eight. Depending on initial configuration estimates, the program can converge to an answer quickly or not at all. To ensure that the experimental run is converging toward an answer, long running experiments must be periodically checked. In steps nine and ten, the experiment run is checked by logging into the computer named *Beta* and browsing the intermediate results of the run. We assume the intermediate results of our example run indicate that it is converging.

At some point the experiment run completes or is terminated by the computational chemist. The scientist then transfers the experimental data to the original computer and reviews the experimental data. Experiments may need to be rerun several time before deciding that the results are adequate. When checking on the experiment run for a second time we find that the experiment has finished. Steps eleven, twelve, and thirteen transfer the experimental results back to our original computer for analysis.

If this experiment run was a success, the scientist may desire to model the molecule with another program in an attempt to gather additional data. Using an optimized molecular configuration estimate from a previous experiment run will help the new model converge quickly. However, accomplishing this sharing of data is difficult. Currently the modeling programs do not share a common file format or common representation of the data in the molecular configuration estimate. For example, one program may represent an atom’s location as three coordinates in three-space, while another program may represent it as a length and two angles. While one representation can be transformed into another, they cannot be used interchangeably. Such differences prevent one program’s results from being directly used in another program. The computational scientist has two choices: convert and format the experiment run into a structure that the new program will be able to use, or enter new estimates for the new program leaving behind the results of previous work.

Cushing [2] notes computational chemists manage large experimental data from many different runs. A computational chemist could have tens of experiment runs progressing and the results of hundreds of experimental runs. Our goal is to facilitate and automate the management and reuse of experiment run data. By automating the management of this scientific data the computational chemists will be more effective. In addition, the sharing of experiment run data can have a synergistic effect on other researchers by simplifying the exchange of scientific data.

1.2CCDB - Proposed Solution

We now discuss our proposal for the management and reuse of this scientific data. We then demonstrate our proposal with the example experiment run above.

Cushing's Ph.D. Thesis [2] has suggested that object-oriented databases and "computational proxies" be used to manage computational scientific data. A computational proxy consists of two parts: computational services and data services. These components can be seen in Figure 1-4.

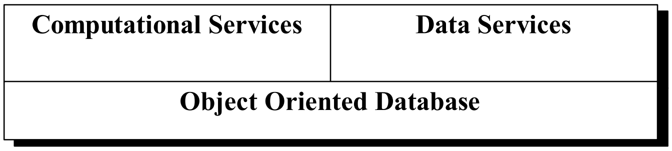


Figure 1-4 Components Of Computational Proxy

The services provided are requested by a computational scientist through a graphical user interface, a CCDB client. This is shown in Figure 1-5. This client allows the scientist to specify at a high level what experimental data is desired. For example, a scientist could request that the client retrieve experiment runs that involve Ethylene. The client is responsible for breaking down complex high level requests into simpler requests of services provided by the computational proxy. The client plays an important role in facilitating the work performed by the computational scientists. However because, it is not a part of the computational proxy its specification and design will not be discussed here.

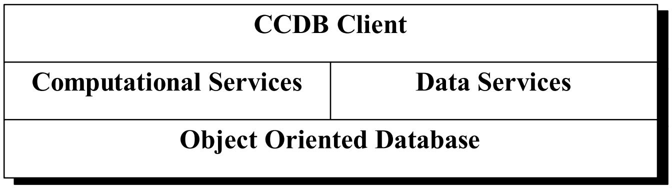


Figure 1-5 Components Of Computational Proxy With CCDB Client

We will now provide a brief summary of each portion of the proxy. After this explanation we will clarify our points with an example. The example will show the interaction a chemist would have with the CCDB client when performing the experimental run from Figure 1-3.

The data services are the first part of the computational proxy. The data services are responsible for managing a database of experiment runs. One service provided is making optimized molecular configurations from past experiment runs available as input to new runs. We refer to this service as a Molecule Configuration Dump. The converse of this service is placing the optimized molecular configurations from complete experiment runs into the database for later review and reuse. This functionality is referred to as a Molecule Configuration Load. In addition to these services, data on specific molecules in the database can be requested from the manager. The last service is named Database Queries. An example would be to have the data services retrieve experiment runs involving the molecule Ethylene. After reviewing data on the requested experimental runs, the chemist can select one experimental run to serve as the initial molecular configuration estimate to a new run. These three services are shown in Figure 1-6.

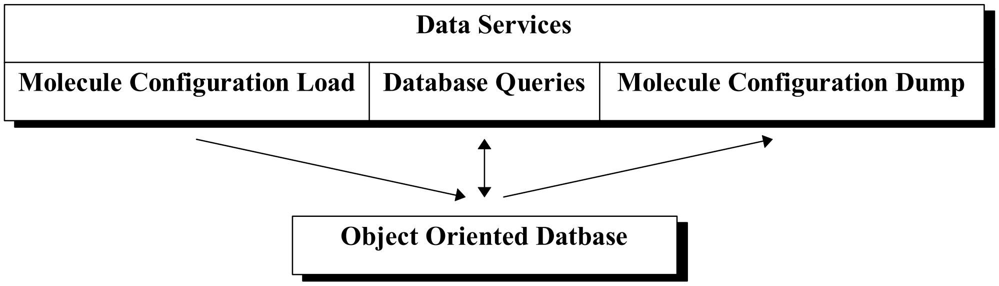


Figure 1-6 Data Services

The process of loading and dumping molecule configuration data sounds straight-forward; however, it is not a simple task. If data is to be transferred between an application and the database, the formats of this data must be the same. There are two possible situations when the two formats do not match. The first is when the data created by an application program does not match the format of the database. The other situation is when the experimental data stored in the database is not in the format required by the application.

In general, the loading and dumping of experiment run data may require several conversions. These conversions are performed in the molecule configuration load and dump components of the data services. Experiment run data in the database may need to be converted into a format that is expected by the application performing the experiment run. On the other hand, an optimized molecular configuration may need to be changed into a form that matches the database schema before being loaded into the database. A method of providing this functionality was described in Cushing's work [2]. Two languages were described that allow the specification of how data is to be converted. One language, called the Computational Chemistry Input Language (CCIL), specified how the experiment run data in the database needed to be formatted so that it could be used as a molecular configuration estimate for a run. The other language, the Computational Chemistry Output Language (CCOL), specified how the data services would parse optimized molecular configurations so that this data could be put in the database. The CCIL language is used to leverage the data of past experiment runs and facilitate new runs. The CCOL is used to return the results from experiment runs into the database. Figure 1-7 shows where the CCIL and CCOL fit in the data services.

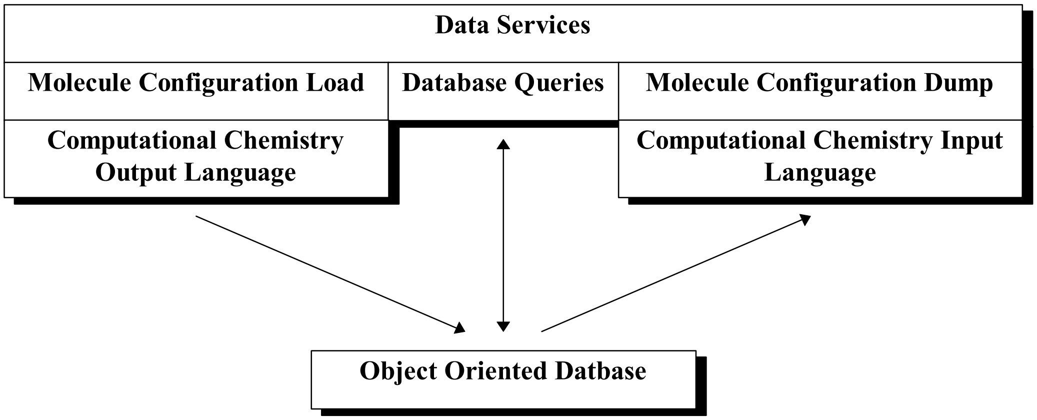

Data Services
Molecule Configuration Load
Database Queries
Molecule Configuration Dump
Computational Chemistry Output Language
Computational Chemistry Input Language
Object Oriented Datbase

Figure 1-7 Data Services Including Chemistry Languages

The computational services are the second part of the computational proxy. The computational services are responsible for starting, querying, and, stopping experiment runs. When a chemist has specified that an experimental run is to be performed, the computational services will locate a host computer on which the experiment can be performed. The computer must have the application installed, and must also have enough memory and disk space to conduct the experiment. Once this has been confirmed the initial molecular configuration data is requested from the data services. This data is then transferred to the host computer and the experiential model run started. Periodically, a chemist may request the status of a experiment run in order to check that it is proceeding correctly. The computational services will retrieve the current status of the experiment run and make it available to the scientist. The last portion of the computational services are the administering of completed experiment runs. When a experiment run completes, the computational services retrieve the optimized molecular configuration results and instructs the data services that the results can be placed in the database. The chemist can review the results when they have been placed in the database. Figure 1-8 presents the salient points of the computational services. Additional data on the computational services can be found in Rao [5].

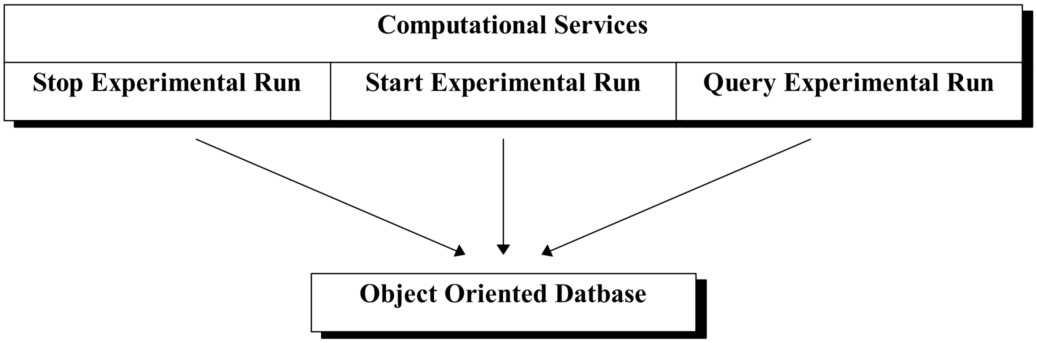
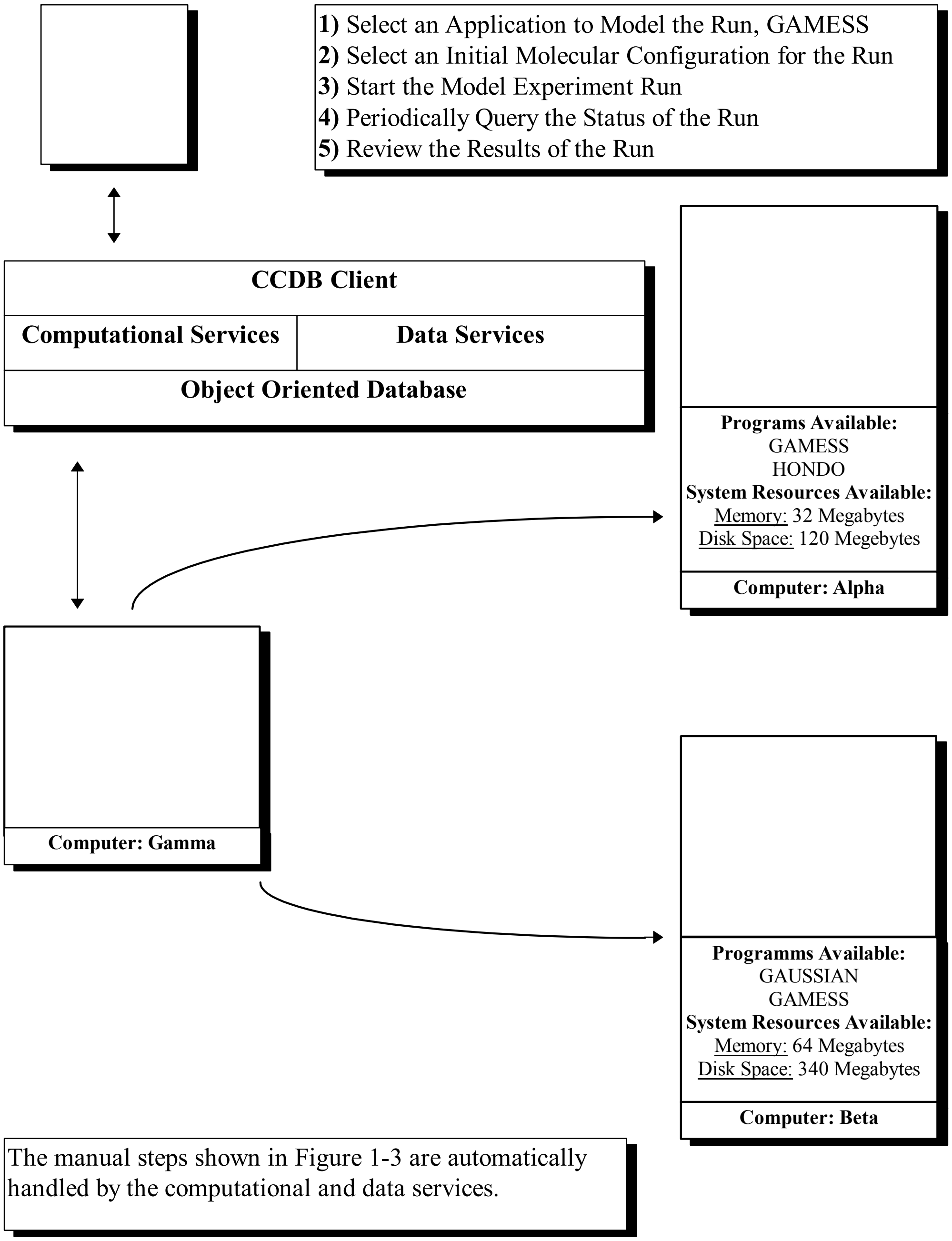


Figure 1-8 Computational Services

These two parts of the computational proxy form an data infrastructure that automates the management and reuse of experiment run data. The computational proxy accomplishes our goal of facilitating and automating the management and reuse of experiment run data.

The computational proxy infrastructure can help alleviate the difficulties associated with the management of scientific data. Working through the example originally shown in Figure 1-3 with the proposed infrastructure will help demonstrate its usefulness. As previously mentioned, the computational chemist does not directly use the computational and data services. The interaction is carried out through an intermediate piece of software, the CCDB client. For the purpose of this example we will assume that this interface is available.

Figure 1-9 shows the steps required to allow a experiment run to be performed with the computational proxy infrastructure. The first step in performing a experiment run is the selection of the application that will computationally model the molecule. When selecting an application the CCDB client will request that the data services retrieve the names of all the applications available. The data services will then query the database. All the appropriate applications names will be returned to the client. The client will show the applications names in a list that the computational chemist can browse. The application selection is accomplished by making a choice from this list.



1) Select an Application to Model the Run, GAMESS
2) Select an Initial Molecular Configuration for the Run
3) Start the Model Experiment Run
4) Periodically Query the Status of the Run
5) Review the Results of the Run

The manual steps shown in Figure 1-3 are automatically handled by the computational and data services.

Figure 1-9 Steps Of Proposed Experimental Model Run

At this point the molecular configuration estimate is needed. (See step two.) The scientist can request data on previous experiments in the database. This is accomplished by entering a query in the client. A limited type of requests can be made in a data manipulation language like SQL [3], or, Query By Example [9] and then mapped to an object-oriented query. The CCDB client sends a request to the data services and the requested data is retrieved. This experimental data can then be shown to the scientist and reviewed. Once an acceptable molecular configuration estimate is selected by the scientist, the client asks the data services to retrieve the data from the database. When this data is found it is in the database's format. The data services will look in the database for the CCIL instructions that explain how to convert the data into a form readily acceptable by the application. This molecular configuration estimate is then converted and presented to the client in a file to be used as input to the application. The scientist is allowed to review and modify the data in the file.

After the computational chemist has completed browsing and modifying the molecular configuration estimate, the experiment run can be started. The computational services can then begin the steps necessary for locating a suitable location for the run.

The computational services first will query the database and locate where the requested application is installed. These sites are potential run locations. The run location list can then be further limited by reviewing the memory and disk space requirements of the experimental run. To do this the computational services will locate the experimental run that was used to create the molecular configuration estimate and use its final memory and disk space usage. Each computer in the list of potential run locations is queried to see if it meets the memory and disk space requirements. Once a computer is located that fulfills the requirements for the run, it is selected as the host computer. Additional requirements such as current load could be used to select a computer. Additional selection criteria can help the balancing of experiment runs across a network of machines, but this optimization is not central to the required functionality.

Once the experiment run has been started, a proxy of the experiment is placed in the database. The proxy is a place holder that contains current data on the partially completed experiment.

During the course of the application's running of the experiment, the computational chemist may wish to check that the run is converging. The chemist can start the CCDB client and request a list of currently running experiment runs. A group of experiment runs can then be selected and the status of each requested. In order to retrieve the status of a experiment run, the client sends a request to the computational services. The computational service can review the data in the proxy and locate the computer that is computing the model. The computational services will transfer the current output of the run along with additional data, such as the CPU time accrued. This data will be used to update the status of the proxy and then presented to the client for review.

If, after reviewing the experimental run, the scientist deems that the run should be terminated, the run can be selected and stopped. This would be accomplished similarly to how the status of an experimental run was requested.

When the model run finishes, the computational services are notified that the run has completed. The results of the experiment run are then transferred back to the computer holding the database. The computational services then requests that the data services load the data into the database. The data services will look in the database for the CCOL instructions that explain how to convert the data into a form readily acceptable by the database. After this conversion is complete the scientist is notified that the experiment run has completed and the results can be reviewed.

One might infer from the above discussion that the above scenario introduces many new steps in managing experiment runs. However, a computational chemist using the CCDB client has a small amount of work to manage an experiment. Reviewing the steps required to produce a experiment run with the computational proxy, the computational chemist must:

1. Select an application to model the run.
2. Select an initial molecular configuration for the run.
3. Start the experiment run.
4. Periodically query the status of the run.

5. Review the results of the run.

Compare the above steps to the procedure currently required:

1. Select an application to model the run.
2. Locate a computer with the application installed.
3. Check the memory and disk space availability.
4. Create an initial molecular configuration estimate for the experiment.
5. Log on to the computer.
6. Transfer the initial molecular configuration estimate to the computer.
7. Start the experiment run.
8. Periodically log on to the computer.
9. Check the status of the experiment.
10. Log on to the computer.
11. Transfer the optimized molecular configuration and results back.
12. Review the results of the experiment.

As computing resources become more and more inexpensive, the number and size of the experiment runs that computational chemists desire to conduct will increase. In the years to come the problem of scientific data management and data sharing will be exacerbated.

By allowing chemists to share data about modeling experiments, past optimized molecular configurations can be used to give new experiment runs better initial molecule configurations. The leveraging of the data from past experimental runs will allow new runs to converge more quickly.

The above overview has specifically discussed computational chemistry. However, the situation for other computational sciences is similar. Data management and data sharing can benefit these areas.

1.3The PCL Interpreter

We now focus on the implementation of an interpreter for the language (CCOL) which transforms output from specific applications to a generic database format.

Figure 1-10 shows how output from specific applications are transformed to and from the object-oriented database schema. The PCL is responsible for transforming the application specific output file into a equivalent generic format and placing it into the database.

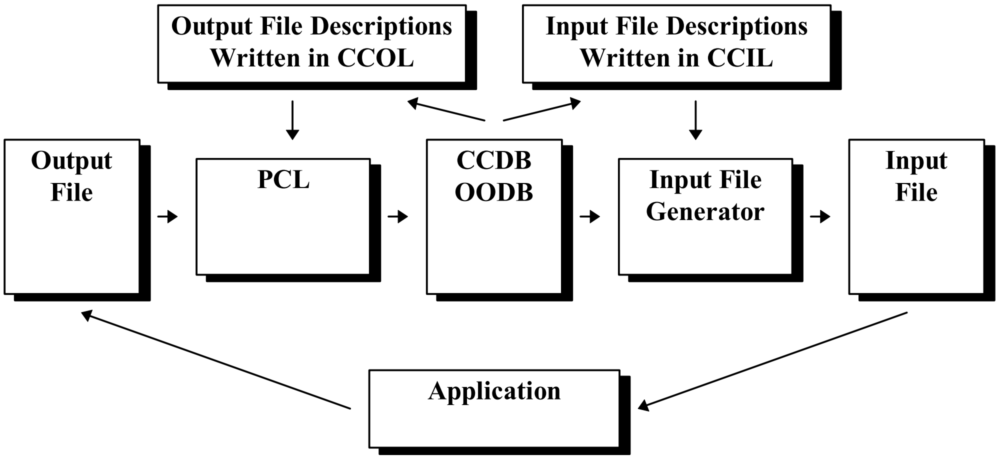

Output File Descriptions Written in CCOL
Input File Descriptions Written in CCIL
Output File
PCL
CCDB OODB
Input File Generator
Input File
Application

Figure 1-10 CCDB Computational Languages

The PCL performs this translation by consulting a list of interpreter directives in the database for the particular application. These directives express the type of information, the location of the information in the output file, and any required conversion functions that are to be performed on the information before being placed in the database. The PCL uses these directives to load experimental data in a five step conversion process.

These five steps are the creation of data representation, locating of data, reading of data, converting of data, and the loading of data. The first step, the creation of the data's representation, involves allocating storage for the data. The amount of storage allocated is declared in the interpreter directive for the particular application. The second step , the locating of data, entails positioning a parsing cursor by searching for specific patterns in the output file. Reading data, the third step, involves loading data from the current parsing cursor location into the allocated storage. The fourth step is converting the data into a format that matches the generic database format. Placing the information into the database is step five.

This thesis describes an implementation of the CCOL language called the Parser Converter Loader (PCL) and part of the data services. The goal of this work is to address the problem of loading incompatible experiment run file formats into the database. This work is central to the ability of the CCDB project to reuse experimental data. The thesis is organized as follows: Chapter Two discusses work related to the PCL. Chapter Three offers the functional requirements and specification of the PCL project. The design of the PCL is discussed in Chapter Four. Chapter Five considers the C++ implementation of the project. Evaluations and conclusions from the project are explained in Chapter Six. Chapter Seven provides analysis and retrospective of the PCL project. Future work is contemplated in Chapter Eight.

2Related Work

2.1Data Reuse

When reviewing research literature we found two approaches to the problem of reusing data from one application as input to another: preventive and permissive.

The preventive approach is that data is stored in compatible formats. In order to prevent the problems associated with incompatibilities, application and platform independent file formats are described and standardized for specific conceptual models. These independent file formats are called Data Interchange Formats (DIF) [1]. Examples of these standardized formats are the "Chemical Exchange Format" for Chemistry, the "Abstract Syntax Notation One" for Genetics, and the "Planetary Data System" for Space Mission Data.

The permissive approach accepts that data may be stored in incompatible formats. The data is converted into the format required by an application by a conversion program. The conversion program can be a customized program, that converts only from one specific file format to another, or a generic conversion program that can convert data in one file format to a common file format. An example of a conversion program with a conversion language is the EXPRESS system developed by Shu and Housel and others [7] at IBM.

The two approaches explained above attempt to solve the problem of data reuse using one or more agreed-upon conceptual models. The conceptual model explains what connotations can be associated with each data file. An example would be the meaning associated with atomic mass. The mass could be for a particular isotope of an atom, or the average mass of all the isotopes of the atom in its natural state. The Data Interchange Formats (DIF) have a conceptual model clearly defined in the specification of the file format. This specification states what data is represented in the file and the semantics of that data. The conversion programs also have a unifying conceptual model. The conversion programs are less stringent than the DIF in how the data is represented in the file. The conversion programs require that the data be conceptually compatible. The conversion programs deal with the problems associated with converting the representation of the data.

It is important to note that without an agreement on a conceptual model neither of these approaches will work. Data items may have several meanings and possible interpretations. Some of these interpretations may be contradictory or may lead to different results based on the interpretation. For example, if we wanted to calculate the mass of carbon found in a sample, some additional data about the carbon atoms in the sample is necessary. Does the sample contain only one particular isotope of carbon or does the sample contain the carbon isotopes in their natural proportions? The results calculated will be different based on which meaning we associate with the data.

2.1.1Legacy Applications And Legacy Data

Two major differences between the approaches taken by the conversion programs and the DIFs are how legacy systems and legacy data are handled. Many of the programs used in the scientific community are legacy applications. Some applications have been used for tens of years and their particular file formats are well known by their users. During the life of the application numerous experiment runs have been performed using them. These experiment runs collectively form a warehouse of legacy experimental data.

The DIF approach to data reuse would require that all the applications of a particular application type be changed to use the new standard DIF. While this seems plausible, there may be several reasons why an application author may not make such a change. First, the change to the application to support DIF may not be not trivial. Second, an author might not support DIF because of a concern that users may migrate to another program if they can easily transfer previous experimental results. Some pools of scientific data are vast and have been accruing for tens of years. Users with a large number of past experiments would not want to lose this data. An application author would need to create a conversion program that would translate the past experiments into the new DIF format.

The conversion program approach requires writing a program that can translate data in one file format to another file format. The conversion program allows use of the currently existing legacy application without modification. The data from past experiment runs produced by these legacy applications are available for reuse by having the conversion program manipulate the data.

2.2Conversion Programs -- EXPRESS

Conversion programs allow data created by one application to be translated into other formats and then used by other applications. As described above, a conversion program can be a customized program, that can convert only from one file format to another, or a generic conversion program that can have an input and output file format described. The customized programs are more common because they are easier to design and create than generalized conversion programs. The price of this simplicity is paid when the customized conversion program must support additional file formats. Cushing [2] estimates that a general conversion program for a particular sub-domain will begin to save development time over a customized program after support for the fourth file format is added. The computational sciences use several different types of applications for modeling and visualization. For this reason we will focus on work that involves generic file conversion solutions.

The EXPRESS system developed by Shu and Housel [7] is an example of a generic conversion application. EXPRESS is a system that transforms data in a hierarchical format from one form into another. EXPRESS's primary use was to migrate data from a flat file or hierarchical database into a relational database. The two main design points of EXPRESS were to allow its use with minimal training and to efficiently use the computer resources while transforming the data. The goal of allowing the system to be used with minimal training is achieved through the two transformation languages, DEFINE and CONVERT. These languages are used to describe the transformations that need to be applied to the data. The languages are non-procedural and thus specify what transformations should occur, rather that state how the transformations should occur. This allows the user to express the transformations in a natural way that is much easier than traditional programming languages.

Because EXPRESS was expected to load large amounts of data, the efficiency of the system was a major concern. The efficient use of computer resources was achieved through concurrency and compilation. Concurrency was used to allow non-dependent transformations to begin processing while other transformations were completing. In addition, non-procedural descriptions of the transformations were compiled into a program. This compilation allow the conversion to run more quickly that an interpreted description.

2.3Conversion Applications -- The PCL

Like EXPRESS, the PCL is an example of a generic conversion program. The PCL is a system that transforms data from one format to another, and loads data into an object-oriented database. The PCL's primary use is to allow the reuse of data in legacy applications. The main design point of the PCL is to allow the system to adapt to new applications, or to new releases of old applications.

The goal of allowing the system to adapt to new applications is achieved by making the program table driven. Entries in three tables are used to control the transformation of data from one format to another. We refer to these entries as "directives" because they direct the transformation. The three types of directives are: creation, parsing, and conversion. The three types of directives are used to adapt the PCL system to new applications. This adaptation is performed by adding creation, parsing, and conversion directives for an application to the PCL tables.

2.4A Comparison Of EXPRESS And The PCL

The PCL was influenced by the design of EXPRESS. This is to be expected as the purpose of the two systems is similar. However, there are several differences that make a comparison of the two systems interesting. We will focus on three requirements and the design trade-offs they caused. The three requirements involved: the type of data transformed, the amount of data transformed, and how often the transformation is performed.

The type of data transformed by the two systems is different. EXPRESS support's the manipulation of basic business data, for example text and simple numeric values. The PCL transformations support the manipulation of scientific data. This type of data has complex hierarchies and is heavily interconnected. Both systems require that the transformed data to be available for later reuse. EXPRESS uses a relational database to accomplish this goal. The need to support highly interconnected complex hierarchies caused us to select an object-oriented database for our repository. The PCL includes functions that change data with one syntax into data with another syntax but equivalent semantics. For example, a function could be written that converts a location from Cartesian coordinates to Polar coordinates.

The two systems transform different amounts of data. EXPRESS is optimized to transform large amounts of data from one file format to another. In an effort to facilitate this conversion two steps were taken: the use of concurrency and the compilation of conversion instructions. The PCL system converts smaller amounts of experimental data and is concerned with the system's ability to adapt to new application formats. The speed of the conversion was a secondary concern for two reasons. One reason was that the amount of data being converted was relatively small, on the order of several megabytes. The other reason was that producing experimental results takes days or weeks of computation and, a few additional minutes during the conversion was deemed insignificant. For these reasons we selected an interpreter to execute our conversion instructions and delayed the contemplation of concurrency during the conversion process. An additional benefit of using an interpreter was that the PCL could be moved to different hardware platforms without needing to change the source code.

The specification and execution of the transformation occur with different frequencies in the two systems. Shu and Housel [7] note that “[i]n practice database conversion is not a ‘one shot’ process. Rather, application systems and their data are moved gradually as the application programs are rewritten.” Conversions in the PCL occur whenever a experiment completes. This can occur tens of times per day, which is much more frequently than anticipated in the EXPRESS system. The EXPRESS system expects the specification and execution of the conversion to occur several times. The PCL, on the other hand, expects changes to specifications of the conversion to occur seldom and the conversions to be invoked frequently. These differences lead to diverse conversion languages. The authors of EXPRESS, expecting the conversion language to be specified often, created non-procedural languages. On the other hand, expecting that the conversion language would be specified less often, we believed that a procedural language would be adequate for a prototype conversion program.

2.5Alternative Systems

We considered if existing pattern-matching tools could reformat the experimental output so that it could be loaded by the computational proxy. Several tools such as PERL[10] and AWK[11] were considered. Both programs were able to handle the reformatting necessary for single-valued objects, however, the scripts to handle the reformatting of complex objects become elaborate. The other problem we encountered was that we saw no direct method of linking reformatted objects generated by PERL and AWK with database objects without creating an intermediate language. For these reasons we did not use alternative pattern-matching tools.

The PCL combines an interesting mixture of ideas: database conversion and loading, complex and highly interconnected scientific data models, and support for unmodified legacy application and data. This blend of ideas permits several design tradeoffs explained above. While the PCL has similarities to existing systems, it addresses the problem of scientific data reuse in several unique and innovative ways.

3The PCL Functional Requirements And Specification

In this section we will explain several types of data incompatibility. This discussion helps clarify what incompatibilities can addressed by a software system. After this discussion, we contrast a customized method with the PCL method of loading incompatible experiment run data into the database. Once the PCL method is presented we will discuss the importance of a shared conceptual model to the PCL solution.

3.1Conceptual, Data Model and Physical Incompatibility

When attempting to address the problems of data incompatibility it is important to define what data level we are discussing. Maier in a paper entitled *Object Data Models For Shared Molecular Structures*[4], defines three levels of data incompatibility: the conceptual, data model, and physical levels. The conceptual level can be thought of as the connotation of terms and concepts. An example is the meaning of the atomic mass of an element. The atomic mass can be thought of as an average of all the isotopes of an element or as the mass of a particular isotope. The data model conveys how a conceptual idea is represented. An example is how a bond between two elements can be represented. The bond can be represented as a pair of Cartesian coordinates or it can be represented a pair of Polar coordinates. Each of these representations contains the same data (semantically), it is merely represented differently (syntactically). The final level of incompatibility is the physical level. The physical level is the way data is stored in the computer system. An example of this type of incompatibility is the different byte orderings that computers use. We discuss how each level relates to the PCL below.

The PCL addresses two of these three, namely the physical and data model levels. The conceptual level is not addressed by the PCL, but we assume that a common conceptual model can represent the inputs and outputs of the application of interest.

Agreement at the conceptual level is a precursor to any attempt at supporting informational model or physical compatibility. Maier states "There is no point in discussing physical compatibility of data if there is fundamental disagreement on the meaning or interpretation of that data." [4]

Incompatibility at the data model level is addressed by the PCL directives. Creation directives allow the creation of application-specific representations of conceptual structures. Parsing directives allow the parsing of data into these representations. Finally, conversion directives allow these representations to be transformed into a common type maintained in the database. These directives all assume common conceptual structures.

The CCDB as a whole deals with incompatibility at the physical level when data is retrieved from the database. The database converts stored data into the byte ordering required by the requesting computer system. The PCL need not address incompatibility at the physical level since it parses the ASCII files output by the application.

3.2Customized Loading Of Experimental Data To A Database

In this section we describe the functional requirements and specification of the Parser Converter Loader (PCL). The PCL is an implementation of the Computational Chemistry Output Language (CCOL) as shown in Figure 3-1. The goal of the PCL is to load incompatible experiment run file formats into the database.

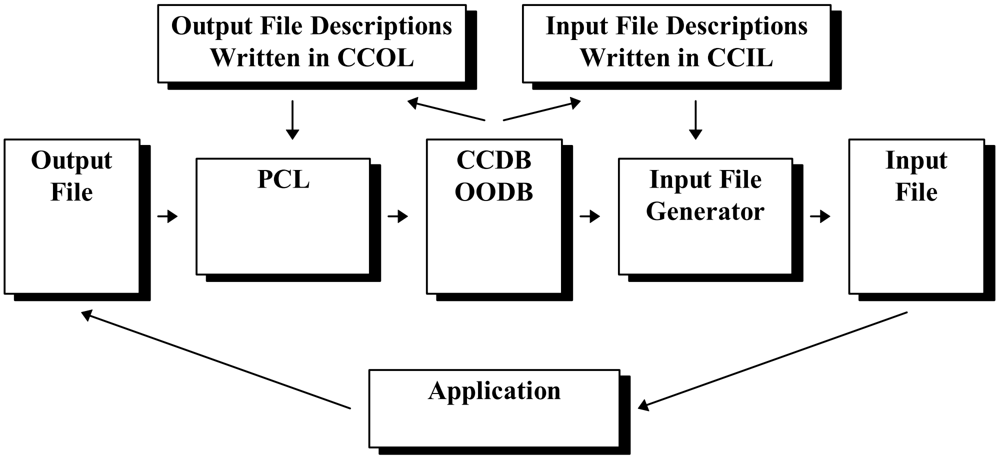

Output File Descriptions Written in CCOL
Input File Descriptions Written in CCIL
Output File
PCL
CCDB OODB
Input File Generator
Input File
Application

Figure 3-1 CCDB Computational Languages

After listing the requirements for the PCL we will explain each in more detail. The requirement of the PCL system is that it retrieve data from the output file produced by a model run and place this data into the database for reuse, see Figure 3-2 below.

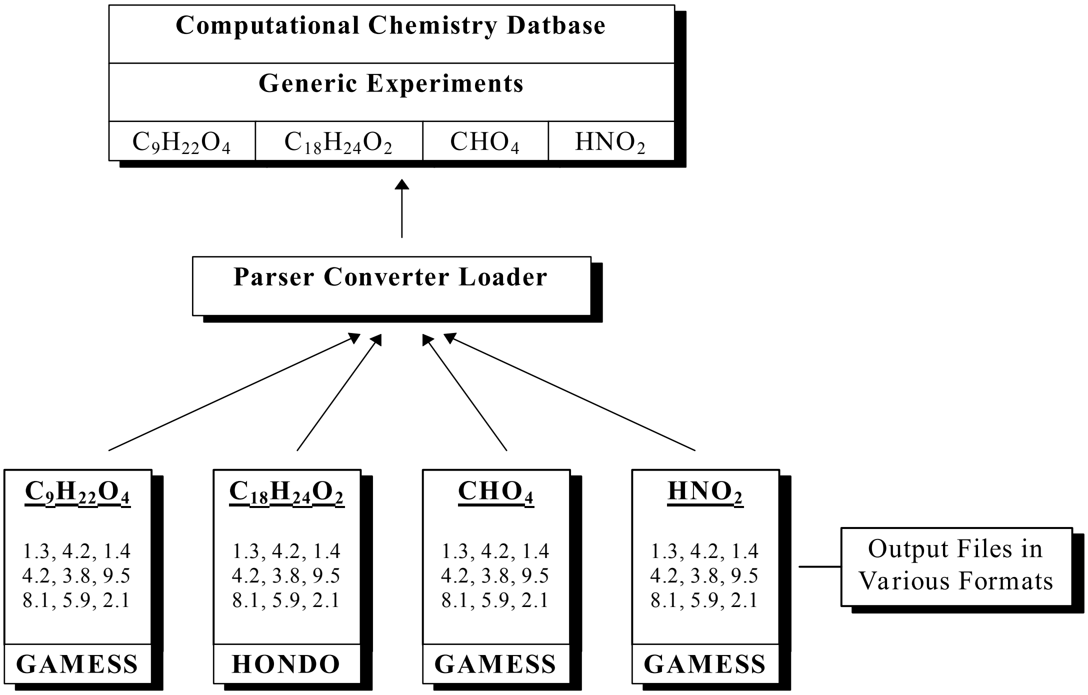


Figure 3-2 Parsing, Loading, and Converting Experiment Run Data From Several Applications

The loading of experimental data could be achieved by writing a customized database loader for each computational application. Each database loader would place data generated by a particular computational application into the database. However, this requires that a new database loader program be written whenever a new application is used. Notice that conceptually all the loader programs would share similar processing needs. The database loaders can be thought of as making this transformation in the five generic steps shown in Figure 3-3 through Figure 3-7.

3.2.1Customized Creation Of Data Representation

The first step in loading the experimental data into the database involves allocating storage to hold experimental data that is in the application's format before it is placed into the database. This can be seen in Figure 3-3. In our example, the GAMESS experiment has five attributes; molecule name, application name, three arrays X, Y, and Z. Each of these has a domain associated with it. The molecule name and application name have the domain of string, and the three arrays have the domain array of double with three elements each.

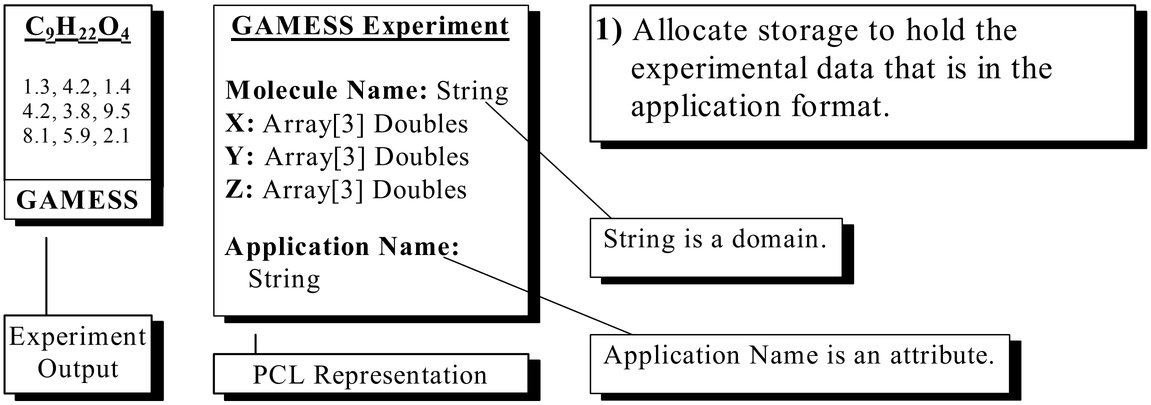


Figure 3-3 Step One -- Customized Creation Of Data Representation

3.2.2Customized Locating Of Data

Step two entails locating the data to be placed in the database. The data is located in the experiment run output. This data is usually located by finding a particular keyword or title. In our example in Figure 3-4 the string desired was the first string in the file and no positioning was required. The arrow in the experiment output points to the data that has been located in the GAMESS experiment run.

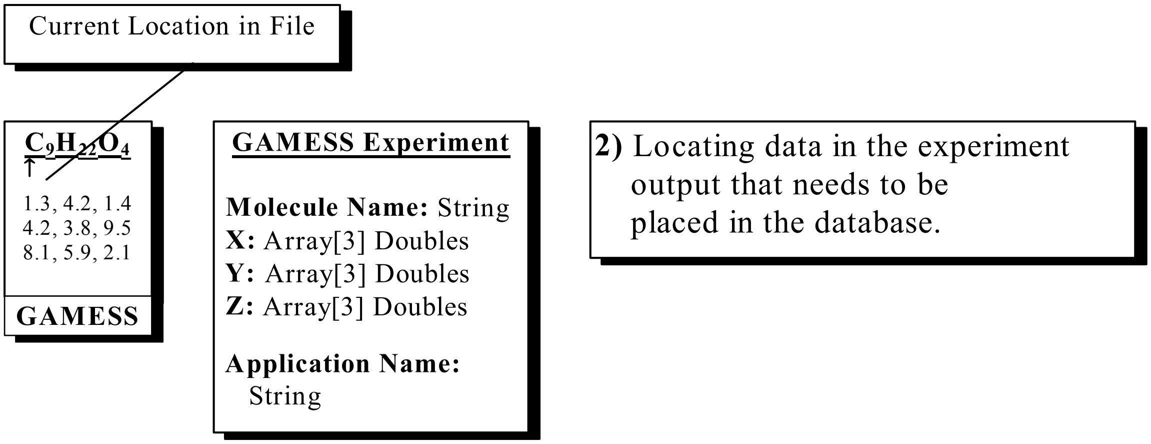


Figure 3-4 Step Two -- Customized Location Of Data

3.2.3Customized Reading Of Data

In step three the data located is placed in the area allocated. The process of locating data in the experiment output and then placing it in the allocated space continues until all the data have been located and read. Figure 3-5 shows the completed results of out searching and loading.

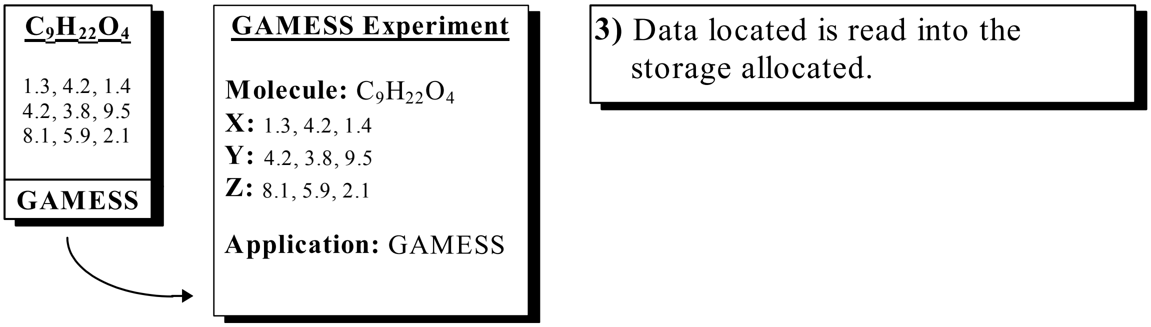


Figure 3-5 Step Three -- Customized Reading Of Data

3.2.4Customized Converting Of Data

The application's representation of data may not agree with the representation in the database schema. In these cases a conversion needs to be applied to transform the data into the format required for loading it into the database. Even if the domain of the application attribute and the database attribute match (i.e. have the same type) there may need to be conversion, For example simply changing the units of measure for a reading. Conversion functions can be arbitrarily complex. The conversion of attributes in the application's representation into the format in the database's schema occurs in step four. Figure 3-6 shows the completed conversion process. The molecule name and application name do not require any changes and are carried forward. The three arrays, X, Y, and Z, however, are converted into the domain of the database schema.

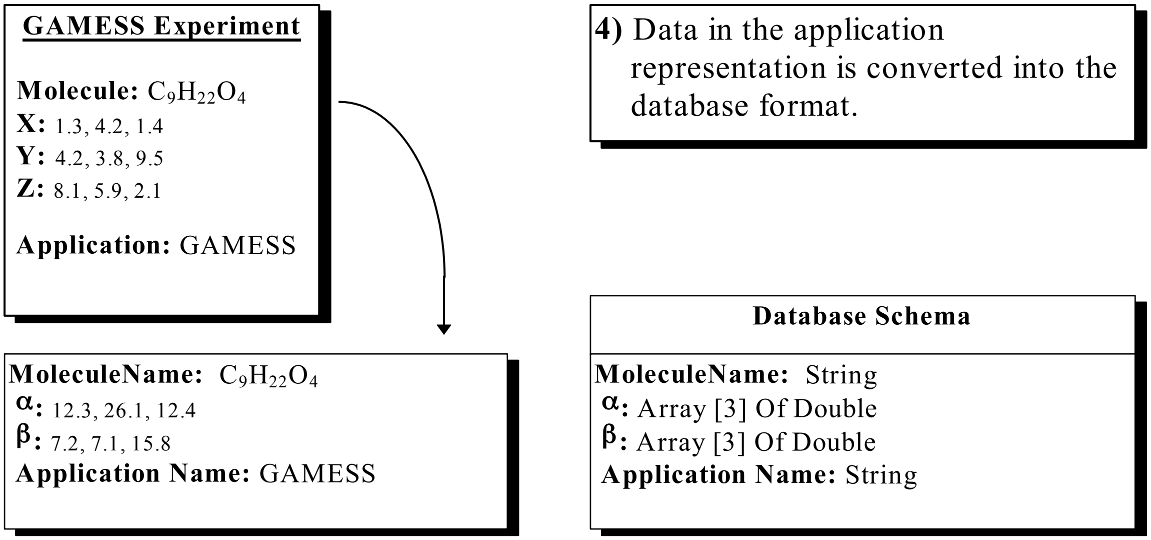


Figure 3-6 Step Four -- Customized Data Conversion

3.2.5Customized Loading Of Data

The fifth step is loading the converted data into the database. This can be seen in Figure 3-7.

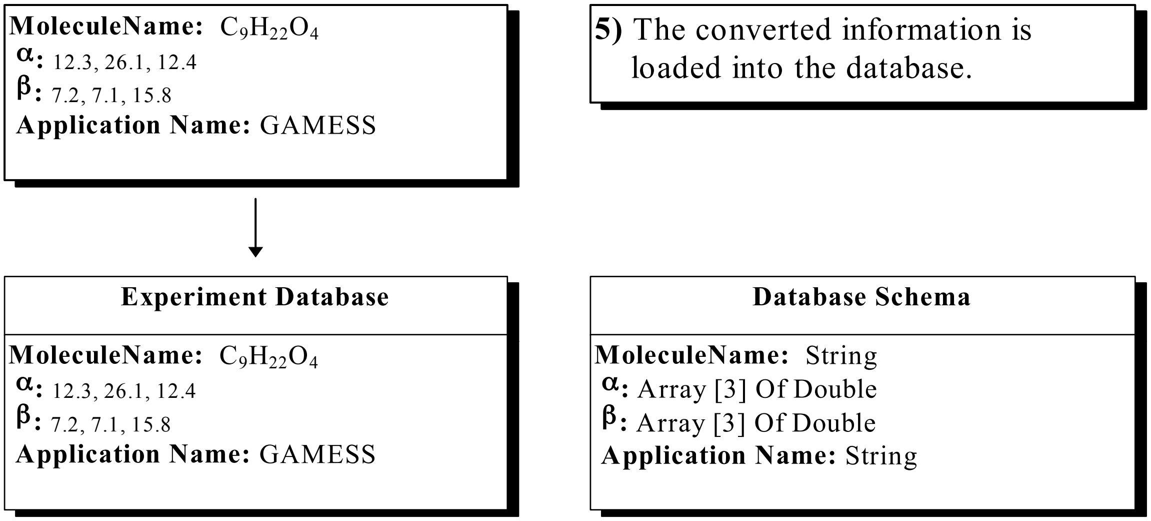

MoleculeName: $C_9H_{22}O_4$
α: 12.3, 26.1, 12.4
β: 7.2, 7.1, 15.8
Application Name: GAMESS
5) The converted information is loaded into the database.
Experiment Database
MoleculeName: $C_9H_{22}O_4$
α: 12.3, 26.1, 12.4
β: 7.2, 7.1, 15.8
Application Name: GAMESS
Database Schema
MoleculeName: String
α: Array [3] Of Double
β: Array [3] Of Double
Application Name: String

Figure 3-7 Step Five -- Customized Loading Of Data

The five customized steps of loading experimental data are summarized in Figure 3.8.

1. Customized Creation Of Data Representation
2. Customized Locating Of Data
3. Customized Reading Of Data
4. Customized Converting Of Data
5. Customized Loading Of Data

Figure 3-8 Five Customized Steps Of Loading Experimental Data

3.3The PCL Loading Of Experimental Data To A Database

In order to avoid writing numerous loader programs, we decided to factor out the common functionality of all such potential programs. Application-specific formats would be communicated to this single program via “directives”. Directives are instructions to the PCL that control the loading of experimental data. This generalization is the conceptual basis of the PCL. The PCL processes directives that control each of the five steps listed above for specific applications. The PCL has specific directives that instruct it how to allocate storage. It also has directives that instruct it how to locate data in the experiment run and how to convert data from one type to another. These directives control how the PCL loads data from a experiment run into the database of computational experiments.

An important goal of the PCL is that it be extensible. If the idea of factoring common functionality out of numerous loader programs is to prove fruitful, the PCL must be able to adapt easily to new applications and to changes in applications. If at all possible, adaptations should be accommodated through the modification of the directives given to the PCL, rather than through PCL code modifications. Code additions may be necessary if new conceptual attributes or domains are introduced by a new application. Sections 3.3.1 through 3.3.5 constitute a program specification for the five steps shown in Figure 3-3 through Figure 3-7:

3.3.1The PCL Creation Of Data Representation -- Creation Directives

The allocation of storage for the experiment data is controlled by creation directives. Creation directives are instructions used to specify, to the PCL, how much space needs to be allocated for an attribute. This storage space is used to hold the experimental data that is in the application's format while it is being placed into the database. The creation directives allow the allocation of storage for each experiment type to be uniquely defined and controlled. The changing of the creation directives will allow the PCL to adapt to new application types.

3.3.2The PCL Locating Of Data -- Parsing Directives

Locating data in the output file is controlled by parsing directives. Parsing directives are instructions used to specify, to the PCL, how to locate data in the output file for the experiment. The PCL maintains a current token location in the experiment run output. As parsing directives are processed, the current token location is updated appropriately. The parsing directives allow data for each experiment type to be uniquely defined and controlled. Changing parsing directives allows the PCL to adapt to new application types.

3.3.3The PCL Reading Of Data

The loading of data is implicitly performed by the PCL. When all the parsing directives for an attribute have been processed, the PCL automatically reads in the attribute. When reading in a data element the PCL uses the current token location to retrieve text.

3.3.4The PCL Converting Of Data -- Conversion Directives

The conversion of data to be placed in the database is controlled by conversion directives. Conversion directives are instructions used to specify, to the PCL, the transformations that need to be applied to a data element. If the representation and meaning of the application data does not agree with the database schema, conversion directives define the transformations to convert the data into the format required by the database. The conversions allow a common semantics between the application and the repository.

Once these transformations are completed, the application-specific data has been converted into the database format. This form is the same as the database schema and can be directly loaded into the database.

3.3.5The PCL Loading Of Data

The loading of the converted data is implicitly performed by the PCL. When all the conversion directives for an attribute have been processed, the PCL automatically places the attribute into the repository.

3.3.6The PCL Loading Experiment Run Data -- Example

Now that we have listed the five steps involved in loading experimental data we will work through an example. The example will include the CCDB infrastructure steps that precede the start of the PCL and will include how the PCL loads a experiment run into the database. Of the five steps listed above, three are central to the extensibility of the PCL, while the other two are automatically invoked and are not configurable. The three central stages are the processing of the creation, parsing, and conversion directives. These three stages are equivalent to steps one, two, and four listed above. We will refer to them as the directive processing stages. They can be seen in Figure 3-9 through Figure 3-11.

Recall from earlier that when a experiment run is complete, the computational application creates a file that contains the results of the run. This output file is then transferred by the computational proxy from the computer that determined the results to the system that contains the repository. Once the results from the experiment model run have been successfully returned where the PCL resides the PCL program is started. The PCL program is charged with the responsibility of parsing, converting, and loading the results of the experiment and placing this data into the database for reuse. For this example, we assume that a GAMESS application has successfully completed and that the experiment run data has been transferred back to be loaded by the PCL.

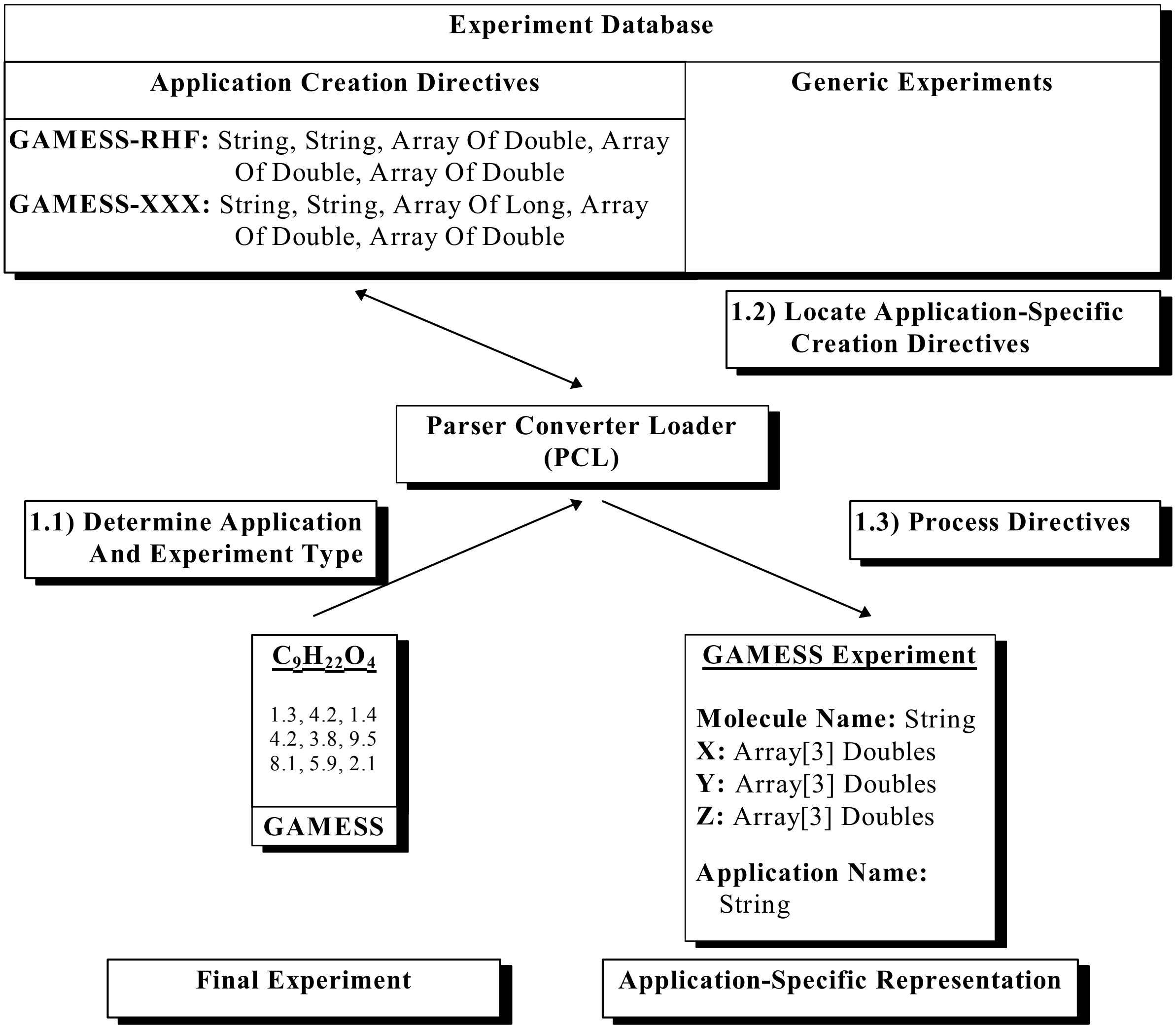

Experiment Database
Application Creation Directives
Generic Experiments
GAMESS-RHF: String, String, Array Of Double, Array Of Double, Array Of Double
GAMESS-XXX: String, String, Array Of Long, Array Of Double, Array Of Double
1.2) Locate Application-Specific Creation Directives
Parser Converter Loader (PCL)
1.1) Determine Application And Experiment Type
1.3) Process Directives
$C_9H_{22}O_4$
1.3, 4.2, 1.4
4.2, 3.8, 9.5
8.1, 5.9, 2.1
GAMESS
GAMESS Experiment
Molecule Name: String
X: Array[3] Doubles
Y: Array[3] Doubles
Z: Array[3] Doubles
Application Name: String
Final Experiment
Application-Specific Representation

Figure 3-9 Creation Directive Processing Stage For The Loading Of A GAMESS Experiment Run

The first stage involves creating an application-specific representation of the run. In our example the application run was performed by the application GAMESS. The type of application that produced the final experiment and the type of experimental run is passed to the PCL by the CCDB proxy. This is shown in Figure 3-9 as stage 1.1. It may be possible to infer this data directly from the output. This was not done because the data is readily available in the proxy. In stage 1.2 of Figure 3-9 the creation directives for the particular application and experiment type are located in the database by the PCL. These directives are entered into the database once by an scientist well versed with the applications whom we call the "registrar". Creation directive data must be provided for each possible application and experiment type combination supported by that application.

The creation directives located in the database explain the application-specific representation of the data contained in the experiment run. We see that the GAMESS application representation in Figure 3-9 has five attributes. The attributes are Molecule Name, Application Name, X, Y, and Z, with domains string, string, and three arrays of three doubles respectively. Once the application and the experiment type are located in the database, each associated creation directive is processed by the PCL. Processing these directives produces the application-specific representation of the experiment run data. The results of this processing are shown in stage 1.3 of Figure 3-9. The constructed application-specific representation of the experiment can now be filled.

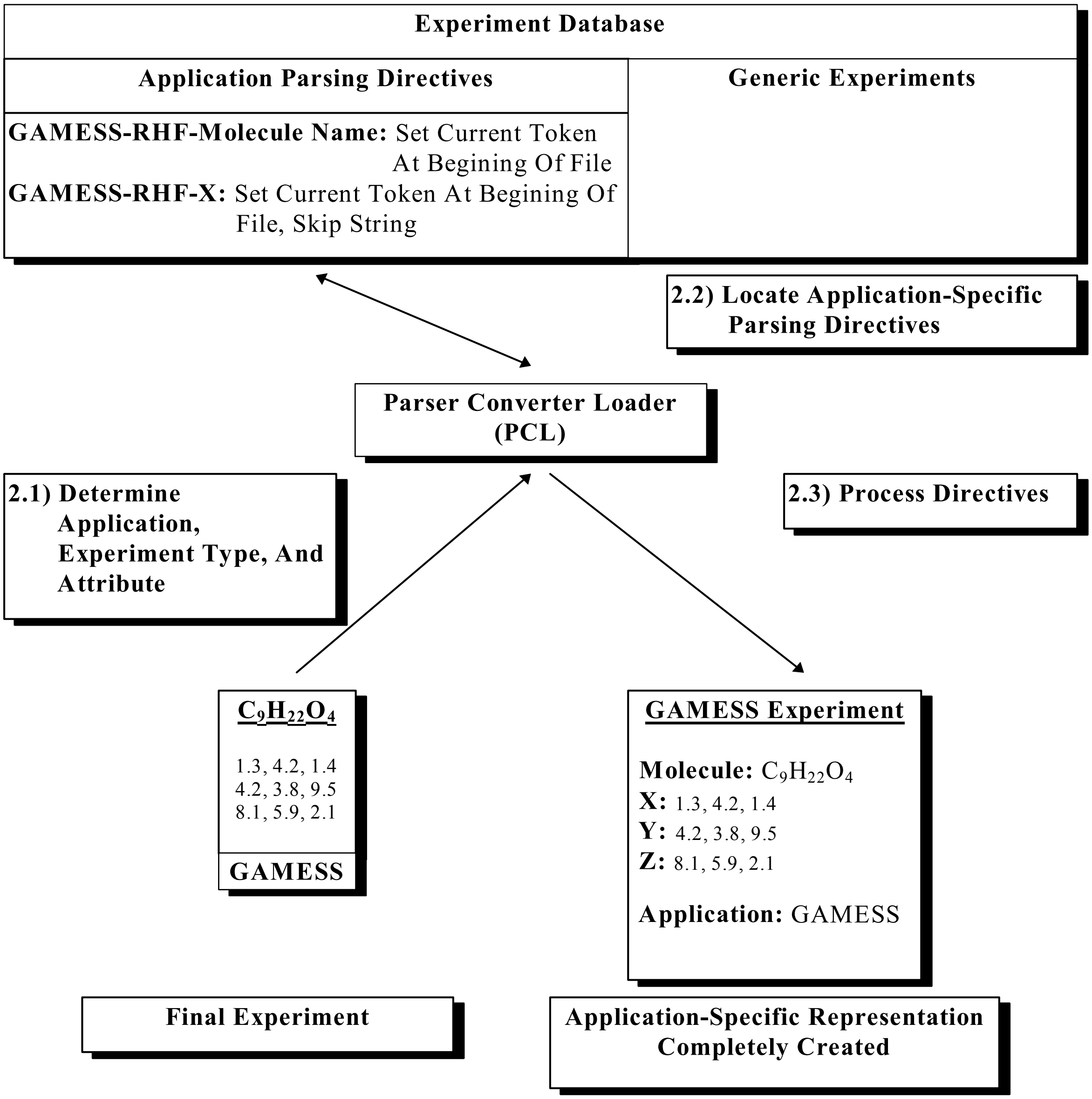

Experiment Database
Application Parsing Directives
Generic Experiments
GAMESS-RHF-Molecule Name: Set Current Token At Begining Of File
GAMESS-RHF-X: Set Current Token At Begining Of File, Skip String
2.2) Locate Application-Specific Parsing Directives
Parser Converter Loader (PCL)
2.1) Determine Application, Experiment Type, And Attribute
2.3) Process Directives
$C_9H_{22}O_4$
1.3, 4.2, 1.4
4.2, 3.8, 9.5
8.1, 5.9, 2.1
GAMESS
GAMESS Experiment
Molecule: $C_9H_{22}O_4$
X: 1.3, 4.2, 1.4
Y: 4.2, 3.8, 9.5
Z: 8.1, 5.9, 2.1
Application: GAMESS
Final Experiment
Application-Specific Representation Completely Created

Figure 3-10 Parsing Directive Processing Stage For The Loading Of A GAMESS Experiment Run

The second stage of loading computational experiment data involves the locating and parsing of data in the output file. In Stage 2.1 of Figure 3-10 the PCL is instructed to locate and parse a particular attribute. The complete explanation of how the PCL is instructed to locate and parse a experiment attribute is discussed in the PCL design section. Assume, for the moment, that the PCL is instructed that a particular attribute needs to be parsed. In order to locate and parse an attribute the PCL requires three pieces of data: the application that produced the final experiment, the type of run, and the name of the attribute. The PCL will use these three pieces of data to find the parsing directives required to locate and parse the attribute. In Stage 2.2 of Figure 3-10 the parsing directives for the particular application, experiment type, and attribute are located in the database by the PCL. The parsing directives, like the creation directives, are entered into the database once by the registrar. Parsing directive data needs to be provided for each application, experiment type, and attribute combination supported by the application.

The parsing directives located in the database explain how to locate and parse each attribute contained in the experiment run. Once the application, experiment type, and attribute are located in the database, each associated parsing directive is processed by the PCL. By consulting the database we can see that the GAMESS application, performing a RHF experiment type, with the attribute “Molecule Name” has one parsing directive. This directive is “Set Current Token At Beginning Of File”. When this directive has been processed by the PCL the “Molecule Name” attribute is ready to be loaded. By looking at the final experiment run in Figure 3-10 we can validate the correctness of this directive. If we processed the directive we would be at the beginning of the file. We would then read the domain type of the attribute, which is a string. The molecule name would be retrieved correctly.

The processing of these directives produces the parsed application-specific representation of the experiment run data. The results of this processing are shown in Stage 2.3 of Figure 3-10. The parsed application-specific representation of the run can now be converted, if necessary.

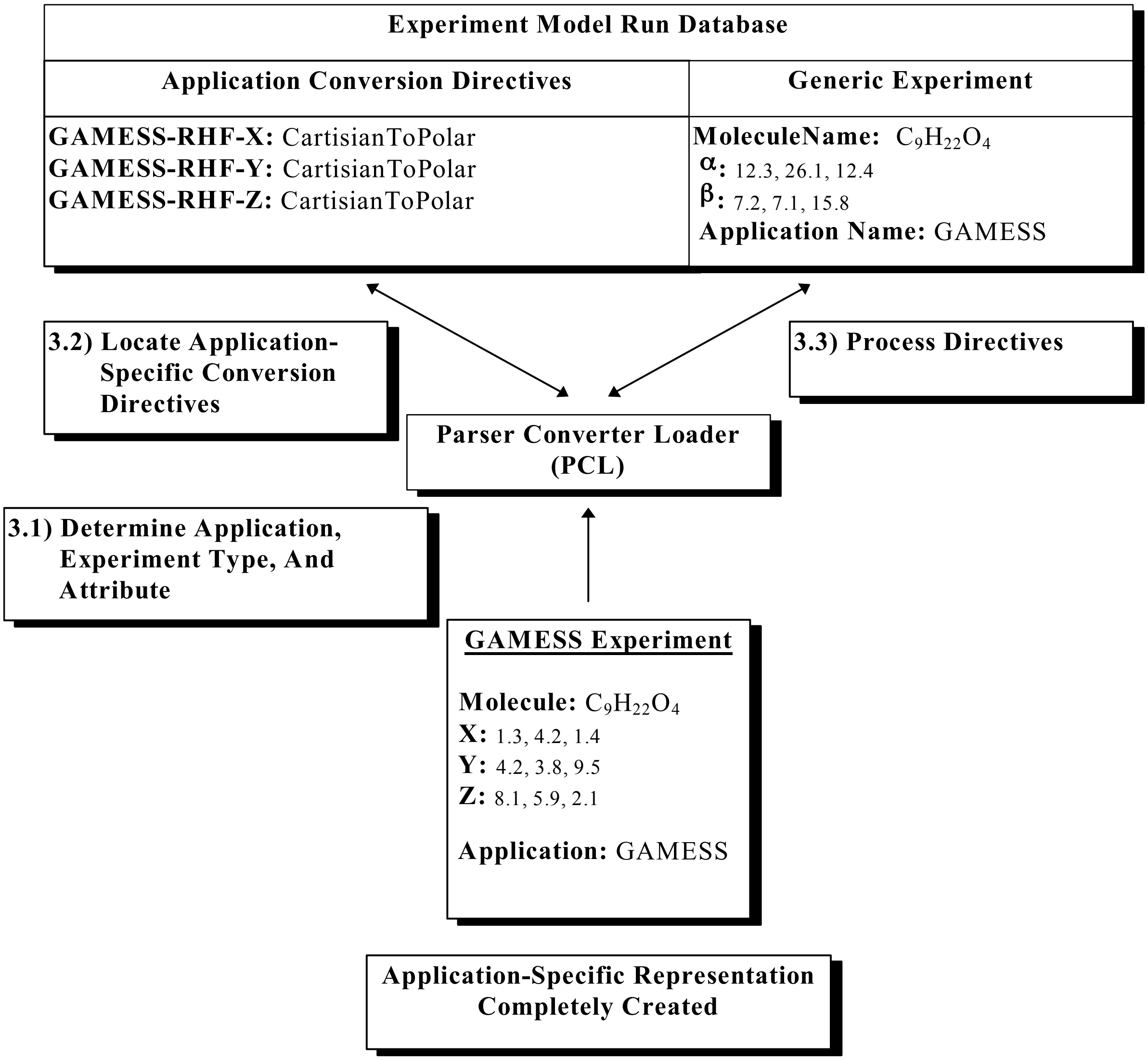

Experiment Model Run Database
Application Conversion Directives
Generic Experiment
GAMESS-RHF-X: CartisianToPolar
GAMESS-RHF-Y: CartisianToPolar
GAMESS-RHF-Z: CartisianToPolar
MoleculeName: $C_9H_{22}O_4$
α: 12.3, 26.1, 12.4
β: 7.2, 7.1, 15.8
Application Name: GAMESS
3.2) Locate Application-Specific Conversion Directives
3.3) Process Directives
Parser Converter Loader (PCL)
3.1) Determine Application, Experiment Type, And Attribute
GAMESS Experiment
Molecule: $C_9H_{22}O_4$
X: 1.3, 4.2, 1.4
Y: 4.2, 3.8, 9.5
Z: 8.1, 5.9, 2.1
Application: GAMESS
Application-Specific Representation Completely Created

Figure 3-11 Conversion Directive Processing Stage For The Loading Of A GAMESS Experiment Run

The third and final stage involves converting and loading the parsed application-specific representation of the experiment run. In Stage 3.1 of Figure 3-11 the PCL is instructed to convert and load a particular attribute. Again, the complete explanation of how the PCL is instructed to convert and load a experiment attribute is discussed in the PCL design section. Assume that the PCL is instructed that a particular attribute needs to be converted.

In order to load an attribute into the database the PCL first checks if the data needs to be converted into another form. Checking for this conversion requires three pieces of data: the application that produced the final experiment, the type of experimental run, and the attribute. The PCL will use these three pieces of data to locate the conversion directives required to convert and load the data. In Stage 3.2 of Figure 3-11 the conversion directives for the particular application, experiment type, and attribute are located in the database by the PCL. The conversion directives like the creation and parsing directives are entered into the database once by the registrar. Parsing directive data needs to be provided for each application, experiment type, and attribute combination.

The conversion directives located in the database explain what conversions to apply to each attribute contained in the experiment run. Once the application, experiment type, and attribute are located in the database, each associated conversion directive is processed by the PCL. By consulting the database we can see that the GAMESS application, performing a RHF experiment type, with the attribute X has one conversion directive, Cartesian To Polar. When this directive has been processed by the PCL a new database object is created, the X attribute is converted into polar coordinates, and this value is loaded into the database.

In some cases the generic format may match the application-specific format, and thus no conversion is required. In these cases no conversion directives are required. A new database object is created, the value of the attribute is copied and this value is loaded into the database. An example of an attribute that does not require any conversion directives would be the molecule name. The GAMESS application representation of a molecule name and the database's match.

The processing of these directives completes the loading of the experiment run data into the database. The result of this processing is shown in stage 2.3 of Figure 3-11. The experiment run data can now be queried and reused in other experiment runs.

3.4The PCL And Data Incompatibility

3.4.1The PCL And A Conceptual Model

We now discuss why, when placing the results of experimental data from several different programs into a database, all the programs must share a common conceptual model. Without a common conceptual model the PCL would not be able to construct application-specific representations of conceptual structures, as there would be no common structure. The PCL would not be able to parse or convert these objects because the common structure is used when performing these operations. Even if we assume that these limitations could be addressed and this data could be loaded into the database, we now have the problem of retrieving data with no common semantics. Data without a meaning is clearly useless. Our work supports the conclusion that, “The key to extensible computer-based chemistry systems and shared molecular structures lies in a common conceptual model” [4].

The conceptual model is central to the ability to share meaningful data across applications, but does not excessively limit those applications. The shared schema represents the common data and theoretical basis that binds the applications. For example, in the CCDB project all modeling programs must agree, at a conceptual level, that a molecule has an energy and a collection of atoms of which it is comprised. The applications must also agree that atoms are conceptually composed of an atom location and an atom type. This agreement on a conceptual form does not describe or limit how the components are actually modeled in the computational program. For example, all the models can agree that an atom has an atom location. Each model can represent this location in any way it sees fit. For example, the representation can be in polar or Cartesian coordinates. Cushing notes in [2] that this common application domain is not easily defined due to subtle nuances in the implied meaning of conceptual ideas.

3.4.2Conceptual Model Support For Data Model Compatibility

A central component of the PCL is the database schema. When created by the user the schema represents a conceptual model (e. g. molecule). The PCL uses the conceptual model to create a compatible data model. The PCL accomplishes this by using the conceptual model as a template for the data model. The data model template is used by the creation directives. The conceptual schema describes the generic object, while the creation directives describe the particular representation of that conceptual object. The creation directives are a statement of the data model in the CCOL. The conceptual schema represents the conceptual objects of which the PCL creation directives create complete application instances. In this manner the conceptual model is mapped to a particular application data model. This mapping provides a level of indirection required to support several data models on top of a single conceptual model.

The ability to support several data models on top of a single conceptual model allows the experimental results from specific applications to be deposited into a generic repository in a common format. Once in this store, experimental data can be view, queried, and applied to a specific problem. A computational scientist who wishes to run an experiment, can browse or query the databases for experiments involving molecules of a similar class or type. Once these experiments have been located the scientist can be used to produce initial guesses of the optimized molecular structure of a molecule. Meaningful data can be gleaned from this warehouse of experimental data because of the unified application schema.

4The PCL Design

In this section we will discuss the design of the PCL system. We will specifically discuss why we chose to design and implement a generic parser rather than several application specific parsers. We will also discuss how the PCL system supports the creation, parsing, and conversion of data for arbitrary computational applications.

4.1Extensibility In A Conversion System

During the design phase of the project, the specification described in Chapter Three was analyzed, and a design was outlined, and validated. The specification is outlined again in Figure 4-1.

1. Creation of Data Representation
2. Locating of Data
3. Reading of Data
4. Converting of Data
5. Loading of Data

Figure 4-1 Five Functional Specifications For The PCL System

When designing the PCL we considered whether extensibility should be a major concern. An extensible system will over the long term require less development effort than customized conversion applications. The primary cost saving an extensible (i.e. generic) system affords is a decrease in development and testing costs. Once a generic conversion program is developed, the cost of supporting a new computational application is incremental. The customized conversion program approach, on the other hand, requires major components of the system to be redesigned, rewritten, and re-tested. As support for additional applications are required, the incremental cost of development for an extensible system overtakes the cost required for several customized conversion systems.

Dr. Judith Cushing is an experienced developer of complex computer systems. In her thesis [2] she claims, based on her experience, that the initial development of a generic conversion system would take 16 weeks. One week of additional work would be required to modify table entries for each additional application. The customized conversion system was estimated to take eight weeks for the first application. Four weeks of development time would be required to develop additional customized conversion applications. Figure 4-2 shows the amount of time required to support different numbers of applications. The asterisk '*' denotes the development break even point, where the development cost of a customized system overtakes the cost of a generic system.

Number Of Applications Supported	Generic Conversion Application	Customized Conversion Application	Development Time Comparison
1	16 Weeks + 1 Week	8 Weeks	17-8
2	1 Week	4 Weeks	18-12
3	1 Week	4 Weeks	19-16
4	1 Week	4 Weeks	20-20 *
5	1 Week	4 Weeks	21-24
6	1 Week	4 Weeks	22-28

Figure 4-2 Development Time Comparison For Generic And Customized Conversion Systems

From Figure 4-2, we see that a generic conversion system is less expensive to develop than a customized conversion application when support for four or more computational applications are required. We expect that the computational legacy systems will continue to be used and that the ability to easily use different applications will drive the scientist's desire to transfer this data to even more modeling and visualization applications. This will increase the need for additional applications support.

For the above reasons we made extensibility a major concern in our design. We wanted the ability to add support for new computational applications to the PCL without requiring changes to, and recompilation of the source code. This goal was achieved by using a table-driven approach. Our system design centered on using a table of directives that control how the different steps in the conversion process are performed. Before discussing how the table of directives are used in the conversion progress we will explain conceptual and base objects.

4.2Conceptual And Base Objects

As explained in Chapter Three, the PCL requires a conceptual model that is shared among all the computational applications. The PCL's goal of data reuse requires us to resolve syntactic differences among particular applications with a shared conceptual model. At one level there is the conceptual model, with which all the applications agree. On the other hand, the data model level describes different implementational representations of the data described by the conceptual model. (Computational applications may represent data differently.) Figure 4-3 shows the difference between the conceptual model and the informational model of an atom. All three representations shown in the data model can be used to represent an atom uniquely.

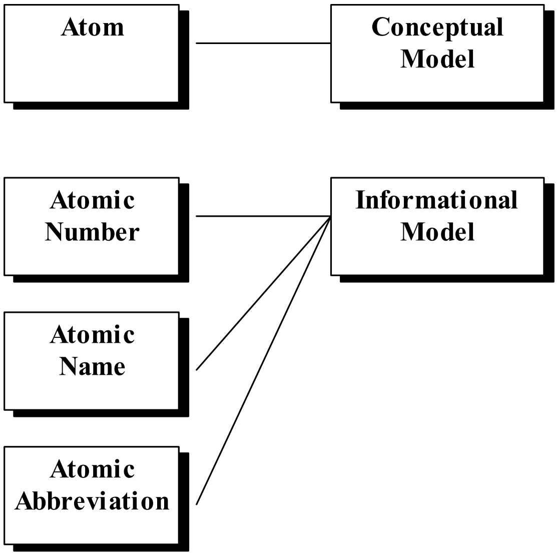


Figure 4-3 Conceptual Model And Informational Model For "Atom"

We call each abstraction “within” the conceptual model a conceptual object. Figure 4-4 shows an example of a molecule conceptual object. A conceptual object can be composed of other conceptual objects. A conceptual object that is a sub-component of another conceptual object X is called an attribute of X. For example, in Figure 4-4, the attributes of the “Molecule” conceptual object are “Name” and ”Final Energy”.

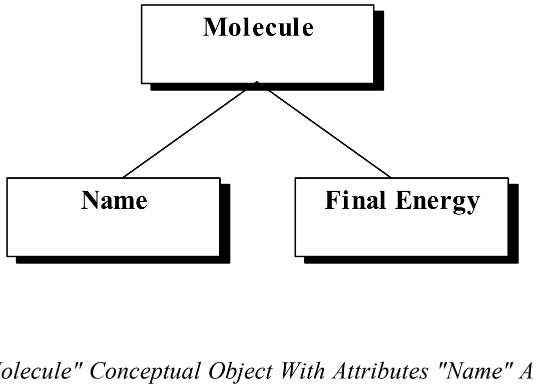


Figure 4-4 The "Molecule" Conceptual Object With Attributes "Name" And "Final Energy"

All computational applications represent the conceptual schema with conceptual objects connected in analogous structure. We refer to this as a conceptual object hierarchy. Each computational application can physically represent the conceptual objects differently in the informational model. We will now discuss how conceptual objects are represented.

The physical representation of conceptual objects are described by base objects. Base objects are physical storage locations used to contain data. Base objects are used to represent the particular data model an application uses to represent a conceptual object. For example, Figure 4-5 shows two base objects, integer and double.

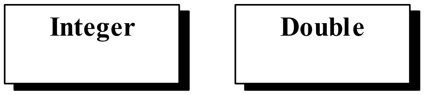


Figure 4-5 An Example Of Two Base Objects

These two types of objects can be used to allow a conceptual object's representation to be associated with arbitrary base objects. With this flexibility, the physical representation of a conceptual object can be changed for different applications. Figure 4-6 shows an example of how four different computational applications might represent the atom conceptual object.

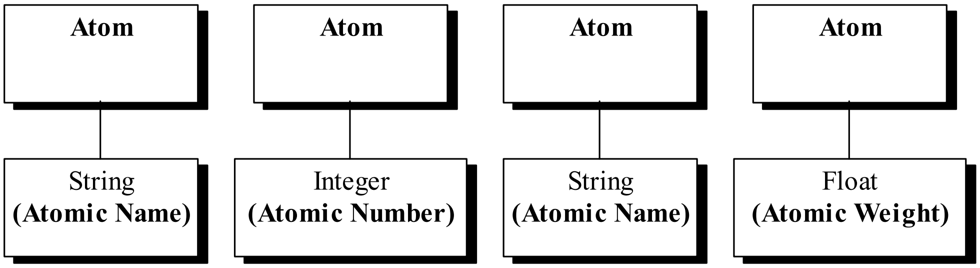


Figure 4-6 Conceptual Object "Atom" With Four Possible Base Object Representations

We designed these two types of objects so that an application can create an arbitrary physical representation for the conceptual objects and thus support the required extensibility. These two types of objects can be used to create a conceptual object in the form represented by a particular computational application.

4.3Operation Of The PCL

We will now discuss how the components of the PCL operate, after which we will discuss each directive type in detail.

Once a computational experiment has been transferred back to the host computer by the computational proxy, the PCL is started. The computational proxy then notifies the PCL of the output filename, the computational chemistry application that produced the results, and the type of computational experiment that was conducted. The PCL uses this data to initialize itself.

The PCL then allocates space for conceptual objects associated with the conceptual model. The hierarchy of conceptual objects is permanent and all objects are allocated as persistent database objects. This hierarchy forms the structure on which the application-specific representation is hung. Figure 4-7 shows a simple hierarchy of conceptual objects for a molecule. At this point in the processing of the computational experiment results, the hierarchy of conceptual objects does not have an application-specific representation.

The following discussions involve a single hierarchy of conceptual objects. However, the PCL is not limited to a hierarchy with a single root node. Multiple hierarchies of conceptual objects would be processed as if each were a single hierarchy. The root object of each hierarchy would merely need to be processed as described below.

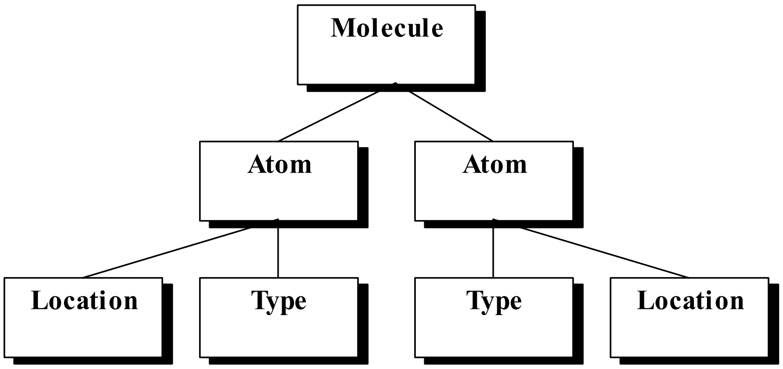


Figure 4-7 Conceptual Object Hierarchy For A Molecule With Two Atoms

The process of loading data involves five steps elucidated earlier in Chapter Three:

1. Creation of Data Representation.
2. Locating of Data.
3. Reading of Data.
4. Converting of Data.
5. Loading of Data.

4.3.1Operation Of The PCL Creation Directives

We will use a single conceptual object -- molecule -- as our example, and generalize the operation of the PCL in the next section. The PCL processing begins by invoking the load function for the root of the conceptual object hierarchy. The molecule object then starts the first step in the loading process. This step is the creation of the application representation for each attribute of the molecule. To accomplish this task the molecule needs to find out how the computational application that created the experiment output represents a molecule. The molecule object does not have the data needed to make this determination and defers this decision to the PCL by invoking the PCL look-up-creation-directive function and passing it the conceptual object. The PCL knows the computational application and experiment type used to create the output file, because the computational proxy passed this data to it when it was started. Figure 4-8 shows the process of locating the creation directives for the conceptual object molecule.

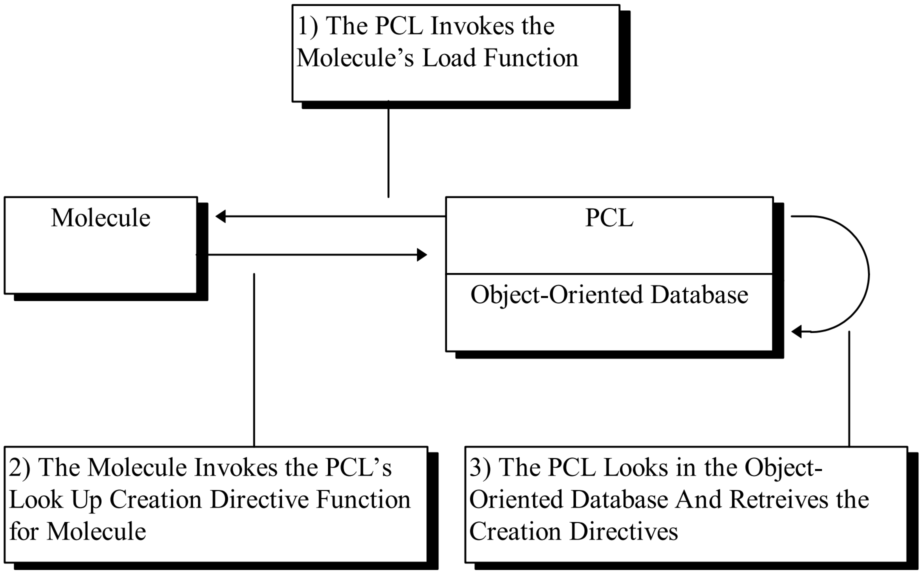

1) The PCL Invokes the Molecule's Load Function
Molecule
PCL
Object-Oriented Database
2) The Molecule Invokes the PCL's Look Up Creation Directive Function for Molecule
3) The PCL Looks in the Object-Oriented Database And Retreives the Creation Directives

Figure 4-8 Creation Directive Look Up For A Molecule

The PCL looks up the representation of the molecule in the database using three pieces of data. (1) It retrieves a list of creation directives. (2) Each creation directive is processed by allocating transitory space for a new base object of the type described in the directive. (3) The transitory space allocated is converted and saved in persistent storage when the application representation is converted into the database format. This new base object is then attached to the conceptual object, and the PCL function returns. Figure 4-9 shows this procedure.

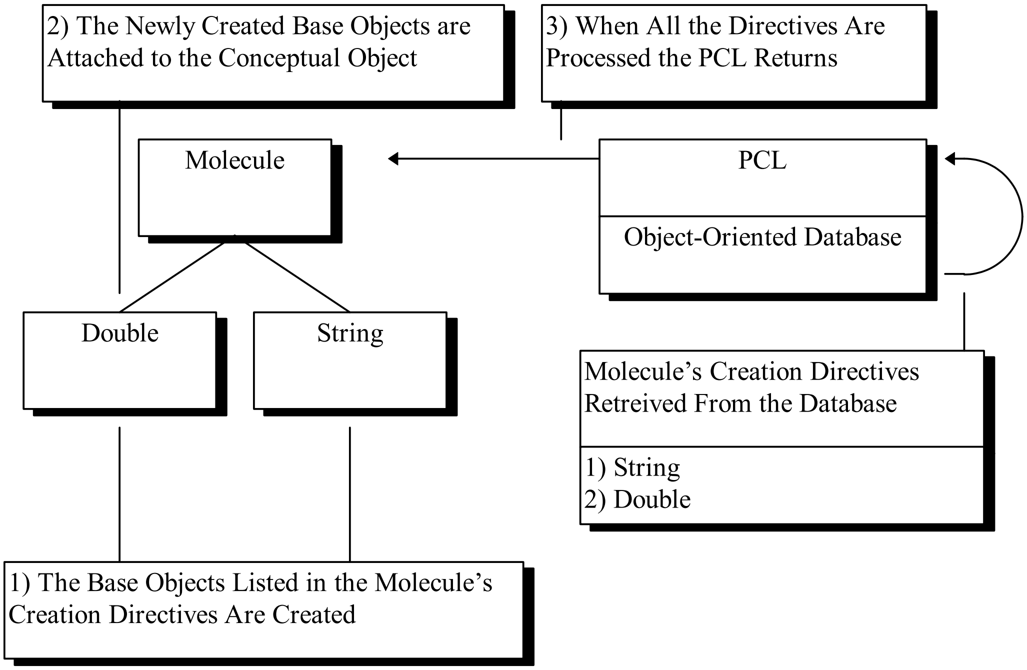

2) The Newly Created Base Objects are Attached to the Conceptual Object
3) When All the Directives Are Processed the PCL Returns
Molecule
PCL
Object-Oriented Database
Double
String
Molecule's Creation Directives Retreived From the Database
1) String
2) Double
1) The Base Objects Listed in the Molecule's Creation Directives Are Created

Figure 4-9 Creation Directive Processing For A Molecule

4.3.2Operation Of The PCL Parsing Directives

The second step in the loading process involves the location of data to be stored in the base objects. To accomplish this task the molecule needs to locate the data to be loaded into each base object. The molecule does not have the data needed to make this determination and defers to the PCL by invoking the PCL look-up-parsing-directive function and passing it the conceptual object and the base object. Figure 4-10 shows the process of locating the parsing directives for the base object molecule.

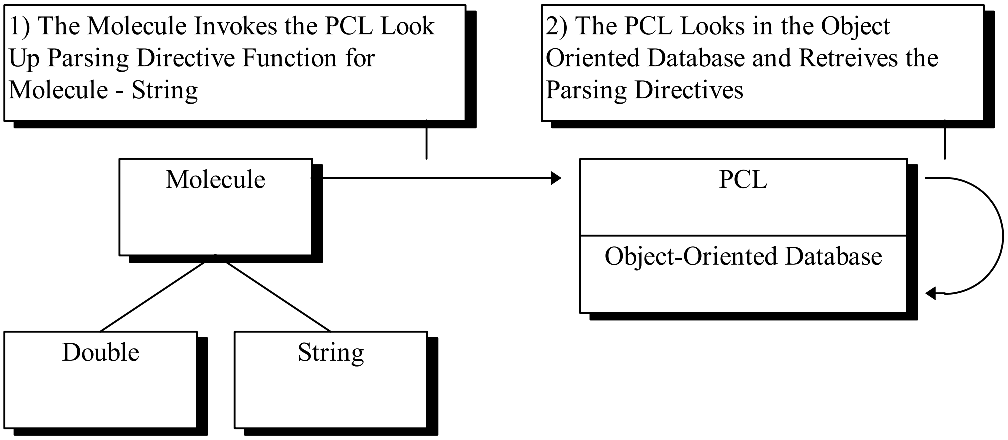

1) The Molecule Invokes the PCL Look Up Parsing Directive Function for Molecule - String
2) The PCL Looks in the Object Oriented Database and Retreives the Parsing Directives
Molecule
PCL
Object-Oriented Database
Double
String

Figure 4-10 Parsing Directive Look Up For The String Attribute Of Molecule

The PCL is instructed to look up the parsing directives that describe how to locate the data for a base object. It retrieves a list of parsing directives. Each parsing directive is processed by executing the interpreter's function with the supplied parameters. The PCL maintains the current location within the textual results with a parsing cursor. The execution of the parsing directives can cause the movement of the parsing cursor and the reformatting of complex text. When all the parsing directives have been processed, the PCL function returns. Figure 4-11 shows this procedure.

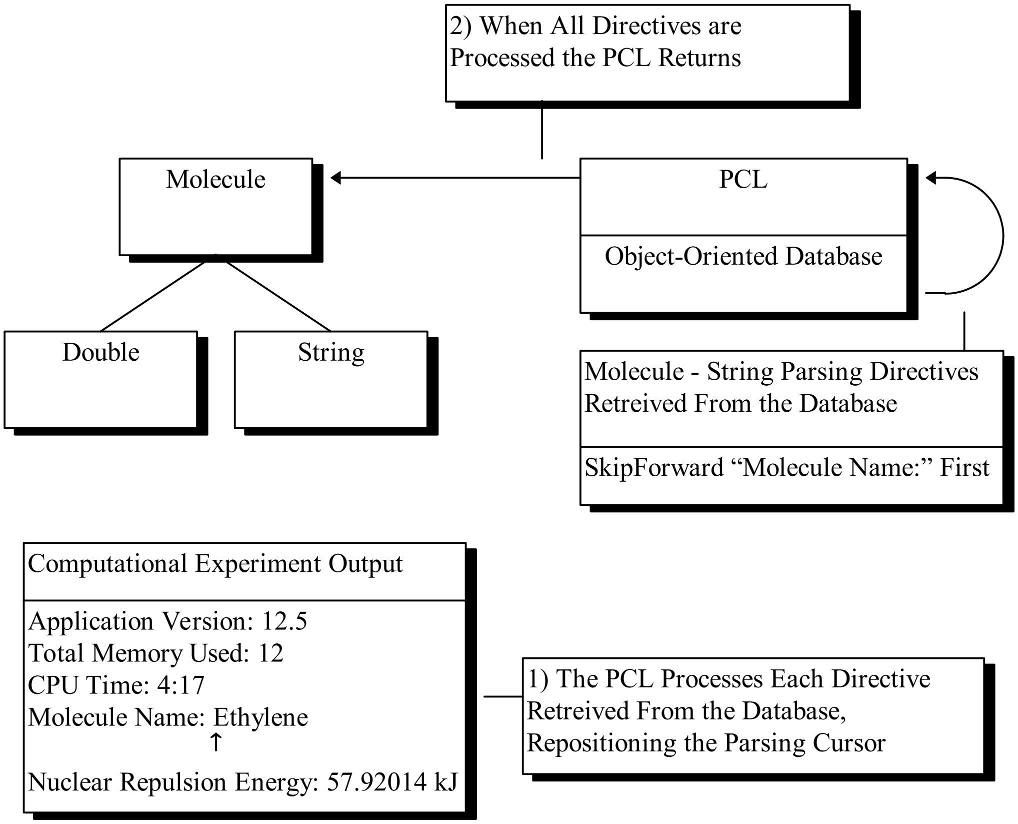

2) When All Directives are Processed the PCL Returns
Molecule
PCL
Object-Oriented Database
Double
String
Molecule - String Parsing Directives Retreived From the Database
SkipForward "Molecule Name:" First
Computational Experiment Output
Application Version: 12.5
Total Memory Used: 12
CPU Time: 4:17
Molecule Name: Ethylene
Nuclear Repulsion Energy: 57.92014 kJ
1) The PCL Processes Each Directive Retreived From the Database, Repositioning the Parsing Cursor

Figure 4-11 Parsing Directive Processing For String Attribute Of Molecule

Upon return from the PCL, the molecule's load function can safely assume that the PCL parsing cursor is properly positioned to read in the base class. The reading of the textual data is step three. The molecule then invokes the read function for the base class whose parsing directives were just processed. The base class then instructs the PCL to read in the text and passes the PCL its base class type. When reading the text the PCL knows what format the text should be stored in because it knows the type of the base class. The PCL uses its parsing cursor as the starting point from which to read the text. When the

PCL has read the data into the base class it updates the parsing cursor and returns to the base class read function. The complete process can be seen in Figure 4-12.

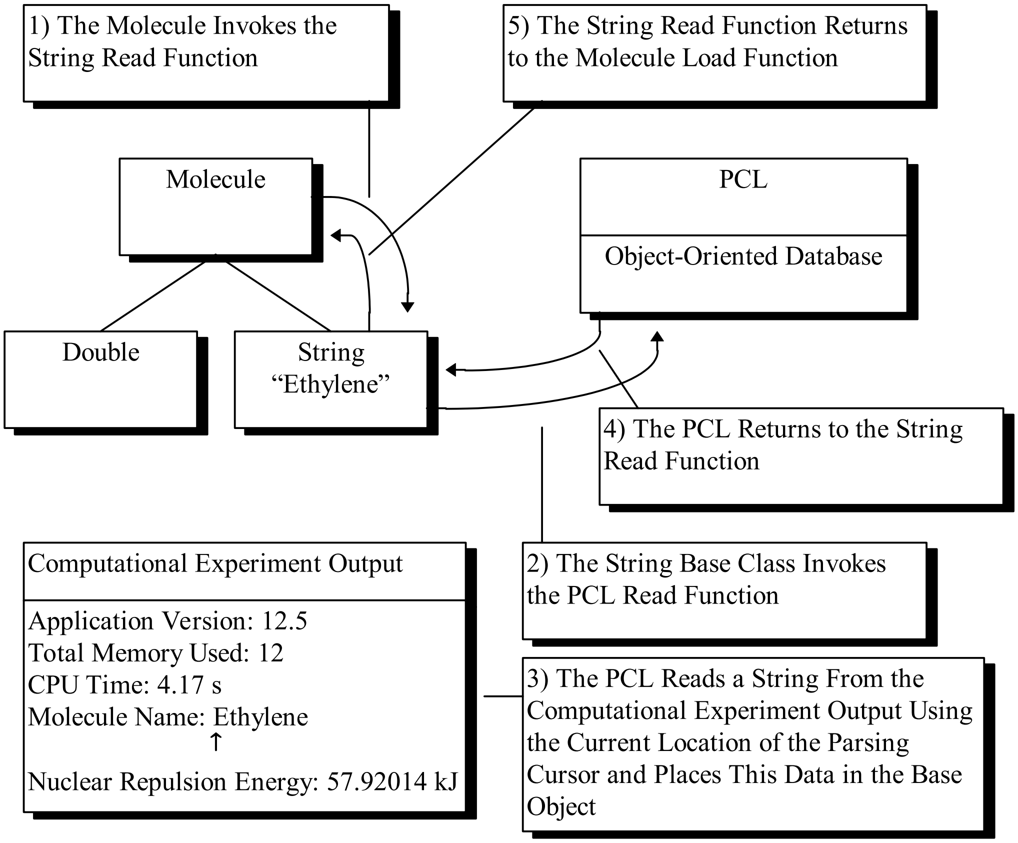

1) The Molecule Invokes the String Read Function
5) The String Read Function Returns to the Molecule Load Function
Molecule
PCL
Object-Oriented Database
Double
String
"Ethylene"
4) The PCL Returns to the String Read Function
Computational Experiment Output
Application Version: 12.5
Total Memory Used: 12
CPU Time: 4.17 s
Molecule Name: Ethylene
↑
Nuclear Repulsion Energy: 57.92014 kJ
2) The String Base Class Invokes the PCL Read Function
3) The PCL Reads a String From the Computational Experiment Output Using the Current Location of the Parsing Cursor and Places This Data in the Base Object

Figure 4-12 Reading Value For String Attribute Of Molecule Using The PCL

The processing of the parsing directive for the double base object occurs in a similar manner as the string base object. We will continue the example with the processing of the conversion directives.

4.3.3Operation Of The PCL Conversion Directives

The fourth step of the loading process involves converting data at each conceptual model level into the generic representation in the database. The PCL is instructed to look up the conversion directives that describe how to convert the data contained in the base objects into the generic representation. The PCL retrieves a list of conversion directives for each base object. This process is shown in Figure 4-13.

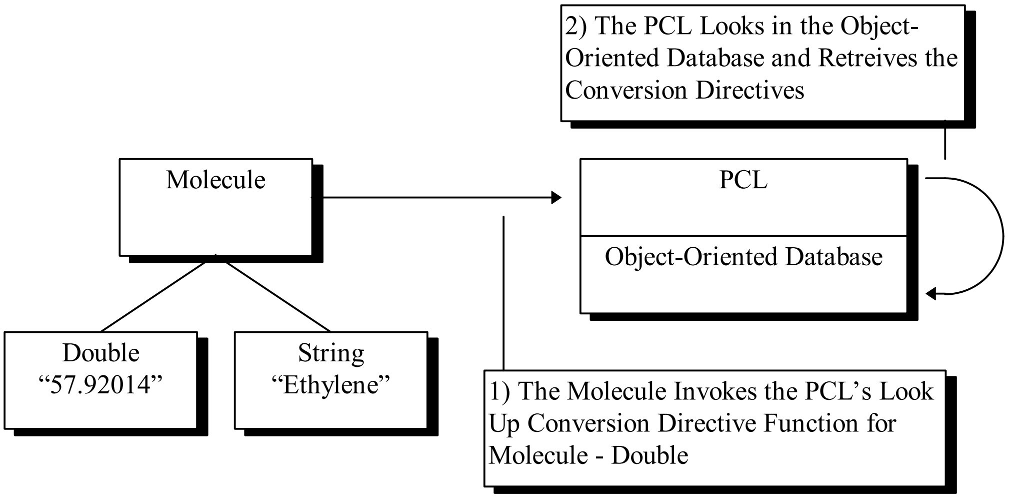

2) The PCL Looks in the Object-Oriented Database and Retreives the Conversion Directives
Molecule
PCL
Object-Oriented Database
Double
"57.92014"
String
"Ethylene"
1) The Molecule Invokes the PCL's Look Up Conversion Directive Function for Molecule - Double

Figure 4-13 Conversion Directive Look Up For Double Attribute Of Molecule

The PCL processes the list of conversion directives retrieved. During the conversion process the basic objects associated with the conceptual object are replaced by the base objects for the generic representation. These base objects are permanent and are allocated in the database. In Figure 4-14 the application representation of the Molecule's energy is converted from kiloJoules to Joules. In our implementation the registrar can make errors of accuracy like converting real to integer; this should be flagged in a production system.

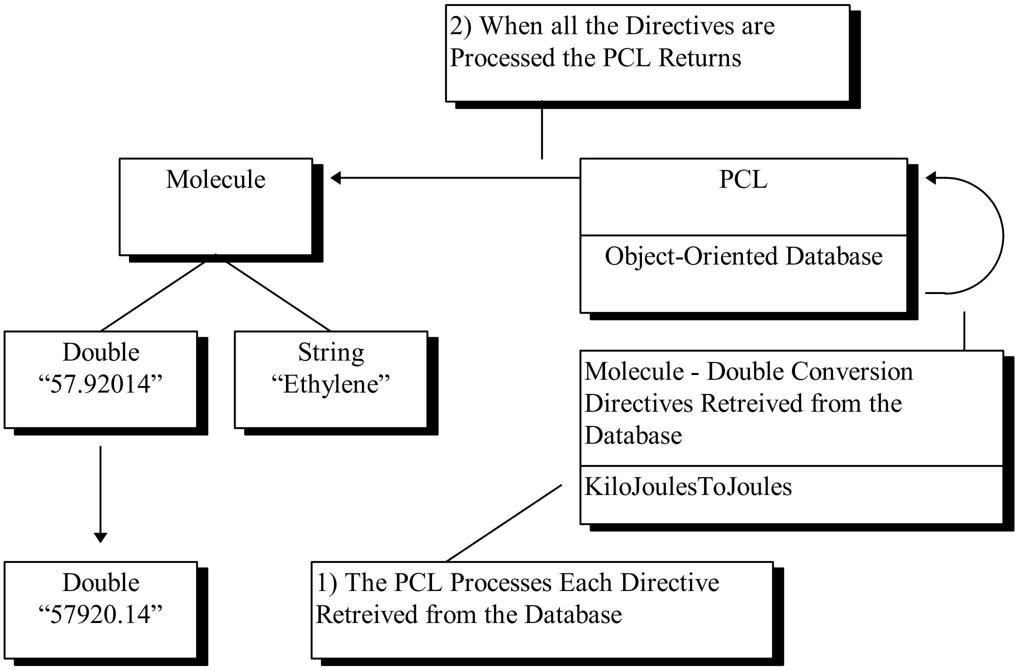

2) When all the Directives are Processed the PCL Returns
Molecule
PCL
Object-Oriented Database
Double "57.92014"
String "Ethylene"
Molecule - Double Conversion Directives Retreived from the Database
KiloJoulesToJoules
Double "57920.14"
1) The PCL Processes Each Directive Retreived from the Database

Figure 4-14 Conversion Directive Processing For The Double Attribute Of Molecule

The fifth and final step in loading the experiment data into the database is placing the newly formed generic conceptual hierarchy into the database. This step merely requires the root object to be loaded into the database. Once this has been performed, all the objects that make up the object hierarchy can be reached by traversing the hierarchy.

4.3.4Operation Of The PCL With A Complex Conceptual Hierarchy

Now that we have explained how the PCL operates with a single conceptual object we need to discuss how the operation proceeds when there are several levels of conceptual objects. A design tenet has been to allow an object to create, parse, and convert only that data that is directly available to that object. Using the conceptual object hierarchy shown in Figure 4-15, the molecule object can create whatever attributes are required to model a molecule for the particular computational application and experiment type. However, the molecule object cannot create, parse, or convert data in the atom object.

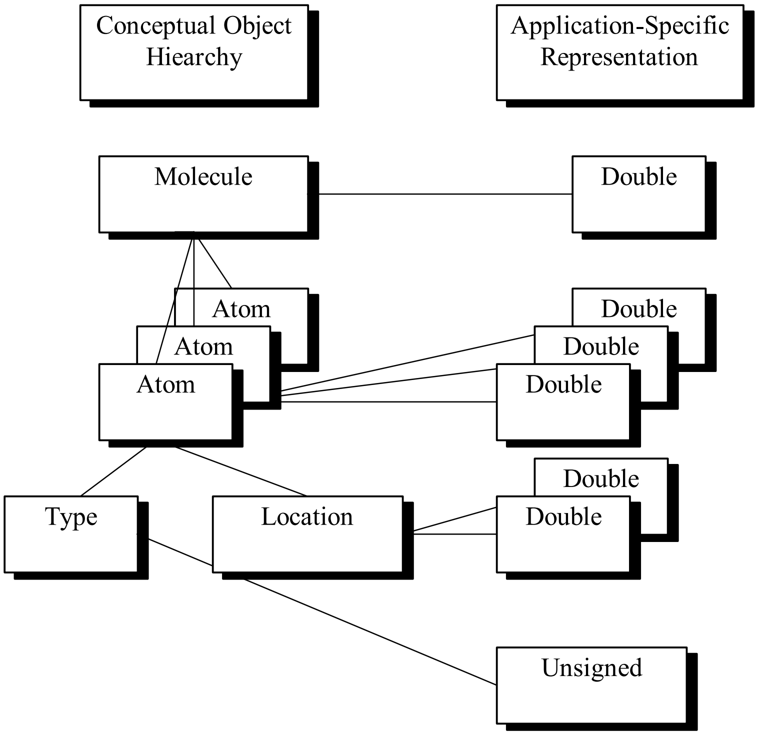


Figure 4-15 Conceptual Object Hierarchy For Molecule With An Application-Specific Model Representation

This design decision causes the processing of the PCL to percolate down the conceptual object hierarchy as directive requests are processed at each level. Figure 4-16 shows the processing that occurs when a conceptual hierarchy is loaded. In step one, the PCL invokes the molecule's load function to begin the processing of the hierarchy. In the second step the molecule process its application-specific representation as described earlier. When this processing is complete we reach step three and the molecule invokes the load function for each of its attributes. In our example hierarchy this consists of a single atom. The atom object then processes its application-specific representation and step four is complete. The atom object now needs to invoke the load function for each of its attributes. In our example the atom type's load function is invoked in step five. The application-specific processing begins in step six. Once complete the atom type's load function returns as there are no additional conceptual objects for which the load function can be invoked. The atom load function now can invoke the load function for the atom location, as is shown in step seven in our figure. When the atom location processing finishes in step eight, it returns to the atom object. Neither the atom object nor the molecule object has additional conceptual objects to which the load message should be forwarded so their load functions return a level. The PCL's original load function call returns at this point. The conceptual hierarchy now has a generic database representation of the experiment data associated with it. The PCL can insert the root of the conceptual hierarchy into the database thus completing the processing.

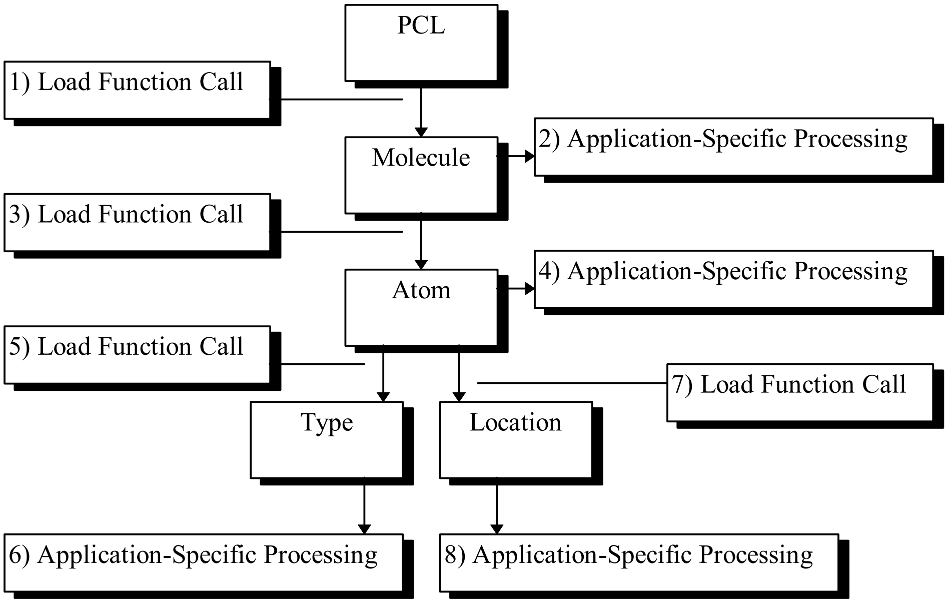

PCL
1) Load Function Call
Molecule
2) Application-Specific Processing
3) Load Function Call
Atom
4) Application-Specific Processing
5) Load Function Call
7) Load Function Call
Type
Location
6) Application-Specific Processing
8) Application-Specific Processing

Figure 4-16 Processing Steps For The Loading Of A Conceptual Object Hierarchy

We will now explain the design of each type of directive used in the PCL system.

4.4Creation Directives

Creation directives are instructions used by the PCL to create an application-specific representation of a conceptual object. Creation directives allow the creation of this representation to be different for each computational application. Creation directives are stored in the database for each computational application and experiment type. The registrar enters these directives into the database when support for the application is being added. The current creation directives are:

- Double
- Unsigned Short
- Signed Short
- Unsigned Long
- Signed Long
- String

These directives refer to data types in the ObjectStore database and thus are machine independent.

Each conceptual object has a list of directives that define what base objects are used by the application to model it. Creation directives are processed when a conceptual object invokes the PCL's create application-specific representation function. This processing occurs before the conceptual object is ready to be parsed from the experimental results. A simple example of this process is the creation of the nuclear repulsion energy of a molecule. As shown in Figure 4-17, the application represents nuclear repulsion energy as a double. The only creation directive for the nuclear repulsion energy of a molecule is double. A more complex example would be an atom, also shown in Figure 4-17. An atom is conceptually made up of an atom-location and an atom-type. The atom-location and atom-type are conceptual objects that have application-specific representations. The application represents the atom-location as two doubles. These two doubles represent the polar coordinates of the atom. The atom type is made up of an unsigned integer representing the atomic number of the atom.

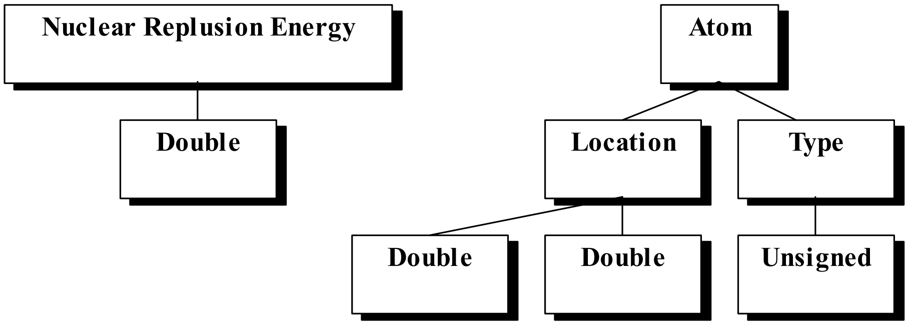


Figure 4-17 Conceptual Objects With Application-Specific Representation

In general the conceptual object hierarchy forms an acyclic graph, which has base objects at the leafs and complex objects at the root and interior nodes. The conceptual object hierarchy has base objects bound to it by the PCL when it processes the creation directives. The creation directives are an important portion of the extensibility available in the PCL system.

4.5Parsing Directives

The parsing directives are instructions used to communicate how to parse the data in the computational experiment's results. There are two types of parsing directives, positioning directives and reformatting directives. We will discuss each type below.

4.5.1Positional Parsing Directives

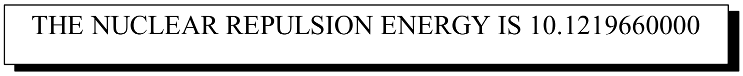
The PCL maintains a current location in the output file of the computational experiment. This data is maintained in a parsing cursor. The parsing cursor marks the place from which the PCL will read its next token. The parsing positioning directives are used to reposition the parsing cursor so that different value can be read. The parsing positioning directives are:

- Skip After (String, Occurrence)
- Skip Before (String, Occurrence)
- Next Line ( )
- Previous Line ( )
- Yield ( )

The Skip After directive allows the parsing cursor to be moved forward. There are two parameters required string and occurrence. The string is the text for which to look. The occurrence is the occurrence for which to look. Occurrence can be the first or last occurrence of the text. The Skip Before directive provides the same function as Skip After except that the processing proceeds toward the beginning of the file. The Next Line and Previous Line directives move the parsing cursor to the next and previous line respectively. The yield directive is used to instruct the PCL to stop processing parsing directives.

4.5.2Positional Parsing Directives Example

We will work through an example using the positional parsing directives by specifying the directives required to parse the conceptual object energy from computational experiment output in Figure 4-18. To simplify the example, we will assume that the PCL has not executed any other positional parsing directives. The location of the parsing cursor is crucial to this process. Initially the parsing cursor is at the start of the file. It moves sequentially, and its position is changed by the positional parsing directives and when a created object is loaded. The first instruction would be to skip to the beginning of the energy number in the text file. This would be specified with Skip To parsing directive, with the String "NUCLEAR REPULSION ENERGY IS" and Occurrence as First. After this parsing directive has been processed the parsing cursor would be located after the last character in the search string. The next directive would be Yield. The yield directive would signify to the PCL that the positioning and reformatting required for this object is complete, and that the energy value could now be read.



```
THE NUCLEAR REPULSION ENERGY IS 10.1219660000
```

Figure 4-18 Textual Representation Of Energy For GAMESS Computational Application

4.5.3Reformatting Parsing Directives

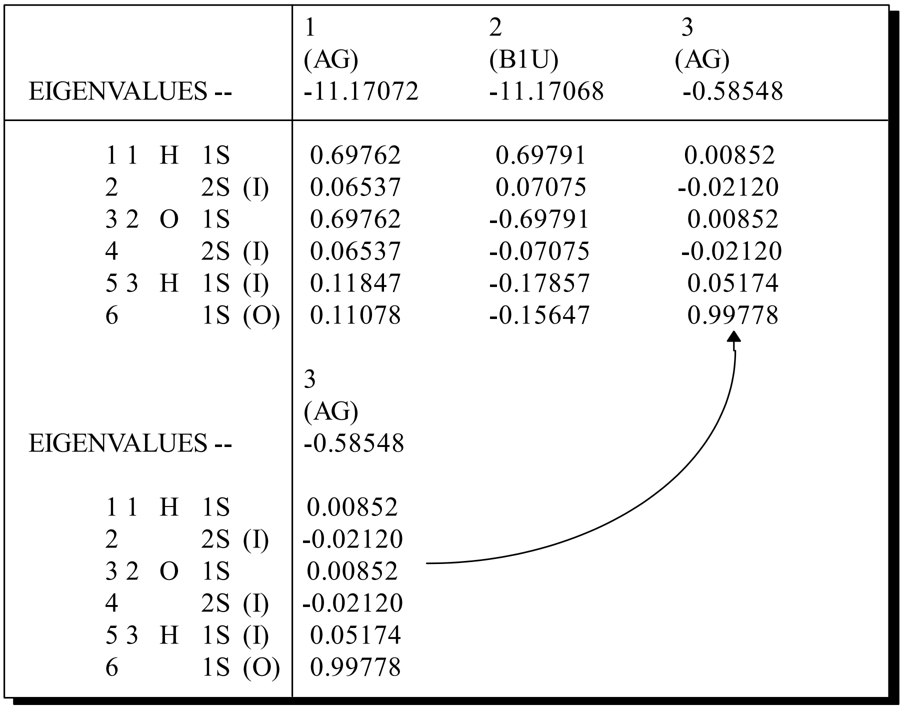
In most cases the computational experiment output has been formatted by the computational application to be viewed by a scientist rather than to be parsed by another program. This brings us to the next type of parsing directives, the reformatting parsing directives. These directives are used to define how the output is to be reformatted before being parsed. The reformatting is performed in order to facilitate the description of how to parse complex data contained in the matrix. The parsing formatting directives are:

- Unfold Matrix (Folded Pages, Rows In Matrix Header, Rows In Matrix Body, Columns In Matrix Row Header)
- Denormalize Matrix (Copy Length, From Relative Line, From Offset, To Relative Line, To Offset, CopyIf Blank, CopyIf Line, CopyIf Offset, CopyIf Length, MatrixStart, MatrixEnd, Increment)

The Unfold Matrix and Denormalize Matrix directives are complex. These two directives will be explained in the context of an example.

4.5.4Reformatting Parsing Directives Example

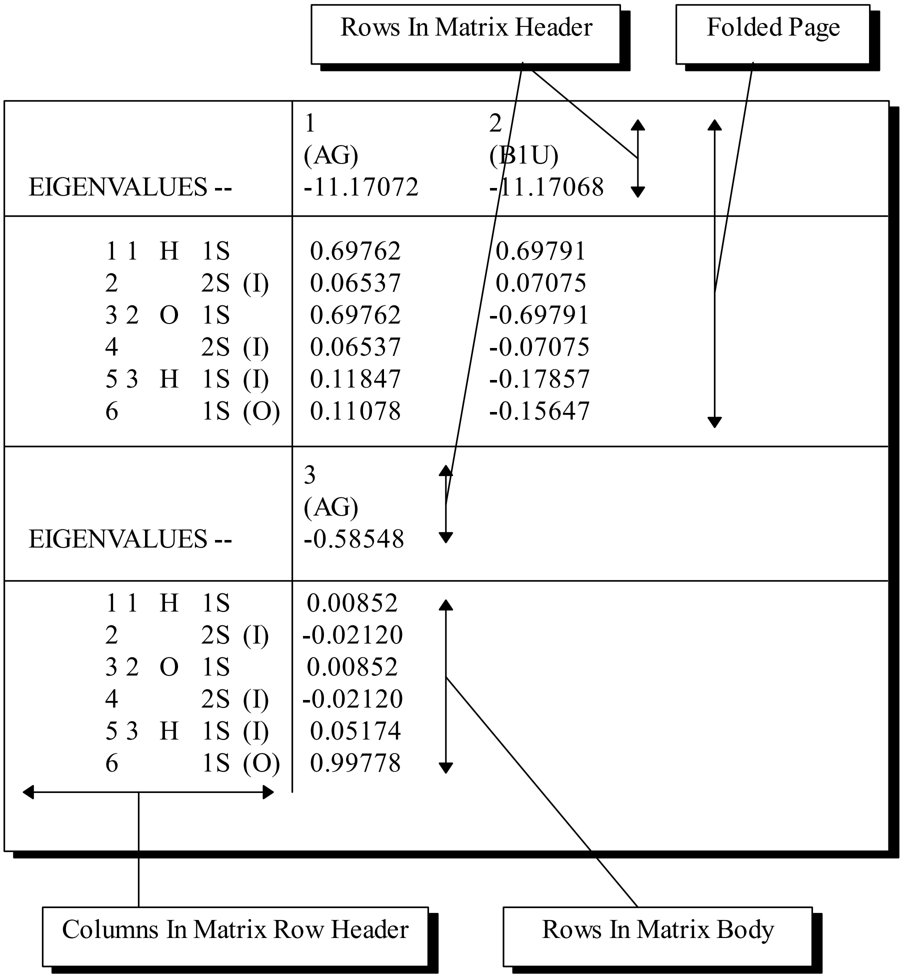
The Unfold Matrix directive is responsible for reformatting a matrix that has been folded across several pages. Figure 4-19 shows an example of a folded matrix. In this example, note that the column and row headers have been duplicated on each page of the folded matrix.



```
                     1             2             3
                     (AG)          (B1U)         (AG)
EIGENVALUES --       -11.17072     -11.17068      -0.58548

       1 1  H  1S     0.69762       0.69791       0.00852
       2       2S (I) 0.06537       0.07075      -0.02120
       3 2  O  1S     0.69762      -0.69791       0.00852
       4       2S (I) 0.06537      -0.07075      -0.02120
       5 3  H  1S (I) 0.11847      -0.17857       0.05174
       6       1S (O) 0.11078      -0.15647       0.99778

                     3
                     (AG)
EIGENVALUES --       -0.58548

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

Figure 4-19 A Folded Matrix Before And After Transformation

It is difficult to define how to parse the matrix in Figure 4-19 using the positional parsing directives listed earlier. The Unfold Matrix directive is used reformat the folded matrix into a single large unfolded matrix. The parsing of a single large matrix is much easier to describe using the positional parsing directives. The Unfold Matrix makes this transformation by locating the body of the matrix on each folded page after the first, and appending it to the first matrix page. This movement is graphically shown in Figure 4-19.

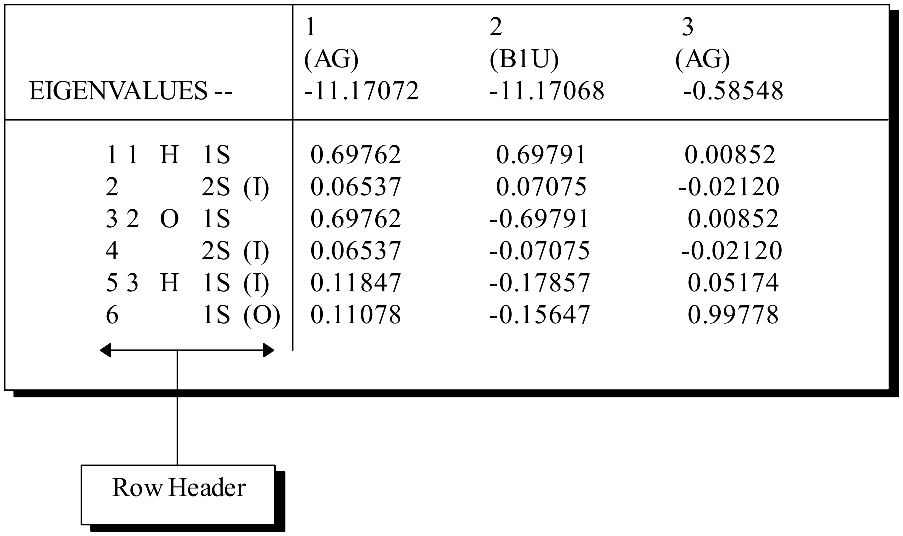


```
                         1              2
                         (AG)           (B1U)
  EIGENVALUES --         -11.17072      -11.17068

       1 1  H  1S         0.69762        0.69791
       2       2S (I)     0.06537        0.07075
       3 2  O  1S         0.69762       -0.69791
       4       2S (I)     0.06537       -0.07075
       5 3  H  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  O  1S         0.00852
       4       2S (I)    -0.02120
       5 3  H  1S (I)     0.05174
       6       1S (O)     0.99778
```

Figure 4-20 Parameters Used In The Unfold Matrix Directive

There are several parameters required in the Unfold Matrix directive. The Folded Pages parameter represents the number of pages in the folded matrix. In Figure 4-20 this value would be two. The first page holds columns one and two, the second page holds column three. The Rows In Matrix Header parameter is the number of rows in the matrix header. In the example this value is three. The Row In Matrix Body parameter is used to describe how many rows there are in the matrix body. This value is six. The Columns In

Matrix Row Header parameter is the number of columns in the matrix row header. This value is 31 in the example. When the Unfold Matrix directive is processed the matrix is reformatted using the parameters passed to the PCL. Figure 4-21 shows the resulting matrix.



```
                    1             2             3
                    (AG)          (B1U)         (AG)
EIGENVALUES --      -11.17072     -11.17068      -0.58548

   1 1  H  1S        0.69762       0.69791       0.00852
   2       2S (I)    0.06537       0.07075      -0.02120
   3 2  O  1S        0.69762      -0.69791       0.00852
   4       2S (I)    0.06537      -0.07075      -0.02120
   5 3  H  1S (I)    0.11847      -0.17857       0.05174
   6       1S (O)    0.11078      -0.15647       0.99778
```

Figure 4-21 Matrix Representation After The Unfold Matrix Directive Processing

Once a matrix is in an unfolded form, as seen in Figure 4-21, it is easier to describe how to parse. Describing how to parse the row headers, however, still remains difficult.

The first problem is that duplicate data has been removed from successive row headers. This has been done to help scientists read the experiment results. Specifically, on line one of Figure 4-21 there are two ‘1’’s, the letter ‘H’, and the string “1S”. Line number two does not have the number one or the letter ‘H’. These fields are the same as the previous line and have been eliminated in an effort to visually denote that this line’s data is related the previous line.

The second problem is visible on line two of the unfolded matrix. The second line has an additional string present, “(I)”, that was not present on line one. This string is actually part of the “2S” string just before it, however, there is white space between the two strings. The parsing of the first string will stop when the space character is read. In order to avoid this, the second string needs to be moved next to the first string. This will allow the two related strings to be retrieved as a single string rather than as two separate strings.

Both of the problems we have just describe are addressed by the Denormalize Matrix directive. This directive is responsible for moving data in the computational experiment output. The Denormalize Matrix directive is powerful and has numerous settings. Its parameters can be divided into four types, Move From data, Move To data, Move When, and a Move How Long. Each will be briefly discussed.

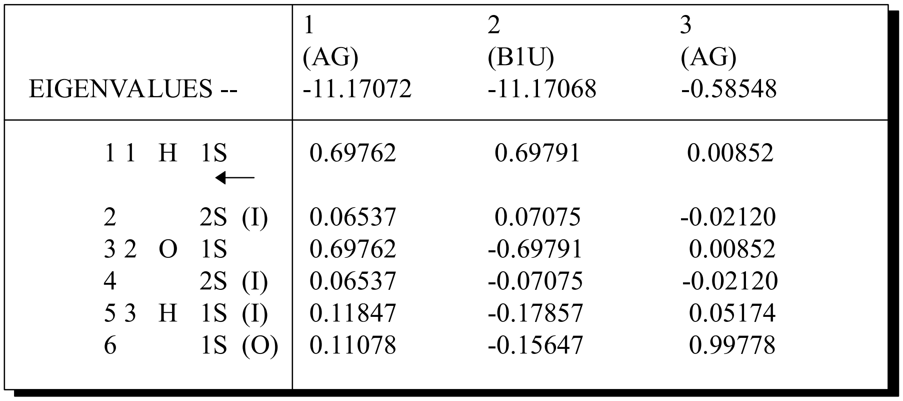
Move From data has three components that control how data will be moved between lines. It consists of three parts: Length, Line, and Offset. Move Length denotes the amount of data that will be moved. From Relative Line is the relative line number from which to move data. This number is relative to the current line number. From Offset is the offset from which to begin moving data. To Relative Line is the relative line number to which data will be moved. To Offset is the offset to which data will be moved.

Move When data has a single setting that controls when a move is performed. There are two options for this setting: move if blank and move if not blank. The move will be performed if the test is true.

The Move When data has three components (like the Move From components above). Move If Relative Line and Move If Offset options are the same as the To and From line and offset variables given above. Move if Length is used to control the amount of data tested.

The final type is the Move How Long and contains three options: Start Relative Line, End Relative Line and Increment. Start Line denotes on which relative line to begin processing. End line denotes on which relative line to stop processing. These settings are relative to the current line number. The Increment setting controls how many lines to increment after checking a line to be moved.

We will now explain how this directive can be used to eliminate the two final problems we have with the unfolded matrix. The processing of the Denormalize Matrix directive will create a final matrix that we call “well-formed”. The well-formed matrix allows for an easily described parsing process.



```
                    1             2             3
                    (AG)          (B1U)         (AG)
 EIGENVALUES --     -11.17072     -11.17068     -0.58548

      1 1  H  1S     0.69762       0.69791       0.00852
               ←
      2       2S (I) 0.06537       0.07075      -0.02120
      3 2  O  1S     0.69762      -0.69791       0.00852
      4       2S (I) 0.06537      -0.07075      -0.02120
      5 3  H  1S (I) 0.11847      -0.17857       0.05174
      6       1S (O) 0.11078      -0.15647       0.99778
```

Figure 4-22 Elimination Of White Space Between Two Strings

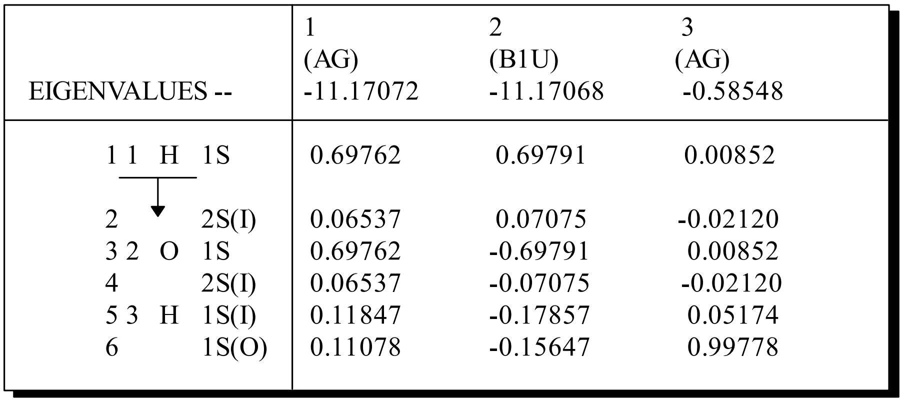
Our first goal is to specify how to get the optional second string next to the first string. First assume that before this directive was executed the current line was set to the beginning the matrix. This example has three rows in the matrix header and six rows in the matrix body. We will want to process each line in the matrix body. So we begin processing at relative line zero and end on relative line five. We should process each line, thus, the increment is one. Now we only need to specify when, to where, and from where to move.

We can look at where each second string begins on each line. If the line is blank we do not have a second string, and we do not need to move it. If there is a string we should move it back two spaces. Converting this data we have a move length of five characters.

Two characters of these five represent the space between the first and second strings and the next three represent the maximum length of the second string. The relative line number is three because we want to start processing line three past the current parsing location. Recall that we assumed this is where we began reformatting the matrix. The offset of the second string is 28 characters.

The final directive is:

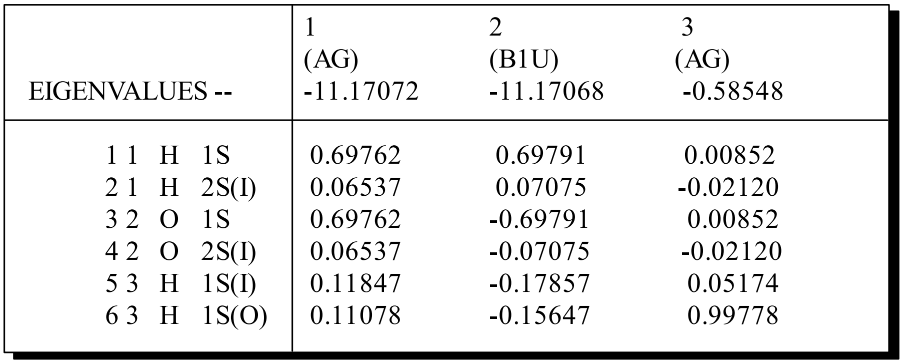
Denormalize Matrix (Copy Length 5, From Relative Line 3, From Offset 28, To Relative Line 3, To Offset 26, CopyIf Blank, CopyIf Line 3, CopyIf Offset 26, Copy If Length 2, Matrix Start 0, Matrix End 5, Increment 1)



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

Figure 4-23 Denormalization Of Data In The Row Header

Our second goal is to duplicate the atom number and abbreviation on any successive line that does not contain this data. The determination of the parameters for this directive proceeds in a similar manner to the previous example. Figure 4-24 shows the final well-formed matrix, after this final directive has been processed.



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

Figure 4-24 Final Well Formed Matrix

Through the use of the reformatting directive, complex transformation can be performed on the experiment output. These transformations ease the complexity of describing how text is located and parsed in computational experiment files. The positional and reformatting parsing directives form a powerful combination that allow complex file formats to be parsed and thus aid the extensibility in the of the PCL system.

4.6Conversion Directives

We have not implemented generic conversion directives in the PCL interpreter. However, we have designed this portion of the system to provide flexibility. This subsection contains some ideas about such future work.

The conversion directives are used to communicate to the PCL what conversion functions need to be applied to a conceptual object represented in an application-specific representation. Invoking the conversion functions on the conceptual object converts the application-specific representation into the database's representation. Figure 4-25 shows this graphically.

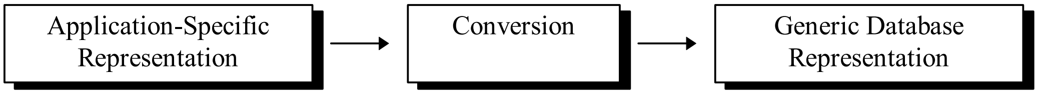

Application-Specific Representation
Conversion
Generic Database Representation

Figure 4-25 Conversion Of Application-Specific Representation Into Generic Database Representation

For example, the GAMESS application might represent the conceptual object atom type as the atomic number as seen in Figure 4-26. The conceptual object atom will have an application-specific representation as an integer. The database may represent the conceptual object atom type as the atomic weight of the atom. The atom in the generic database representation would have a representation of a float. The conversion directive is responsible for stating what functions must be applied to convert the integer representing the atomic number to the float representing the atomic weight.

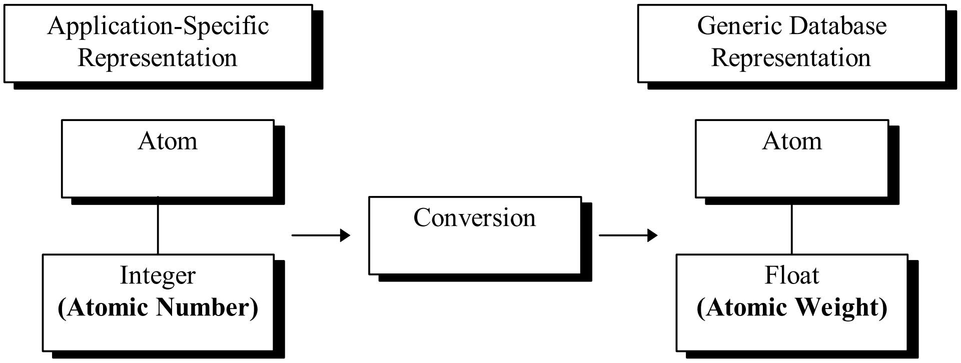

Application-Specific Representation
Atom
Integer
(Atomic Number)
Conversion
Generic Database Representation
Atom
Float
(Atomic Weight)

Figure 4-26 Conversion Of Application-Specific Representation Onto Generic Database Representation

There can be several conversion directives associated with converting a conceptual object from an application-specific representation into a generic database representation. An example would be converting a unit of measure from an application-specific representation of kilograms to a generic database representation of ounces. Assume that we have two conversion directives, one conversion directive for eliminating the kilo unit prefix and a second conversion directive that converts grams to ounces. To make the needed conversion we first apply the kilograms to grams conversion directive. Then we apply the grams to ounces conversion.

The reader might observe that in simple cases, syntactic conversions could be automatically applied. An example would be converting an unsigned integer into an unsigned long. This type of conversion is possible, but would be of limited benefit. The problem that arises is that some semantic data for the base object is not available. This problem can be demonstrated by looking at a promising case. If the application's representation of the conceptual object “Nuclear Repulsion Energy” were a double and the database’s representation a float, a conversion could be automatically applied. The problem is that there may be a conversion needed to change the units of measure on the double. This problem can occur even when the two objects are of the same base object type. For this reason we do not automatically promote base objects in the conversion process.

5The PCL Implementation

This chapter will explain the implementation of the PCL system. We will specifically discuss the object-oriented programming concepts used to implement the design outlined in Chapter Four. The benefit that object-oriented programming provided will be discussed next, followed by a discussion of the language and database systems chosen for implementation. We will also explain how conceptual and base objects were implemented. Parsing directive implementation will be considered, as will several aspects of directive processing. The PCL message-forwarding process will be described in the final section.

5.1Object-Oriented Programming

Object-oriented programming is a method of programming where messages are sent to objects. Objects are abstractions of items being modeled. The abstraction includes the messages to which the objects respond. Figure 5-1 shows an example of a molecule object.

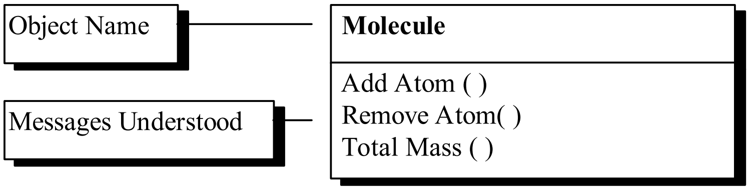
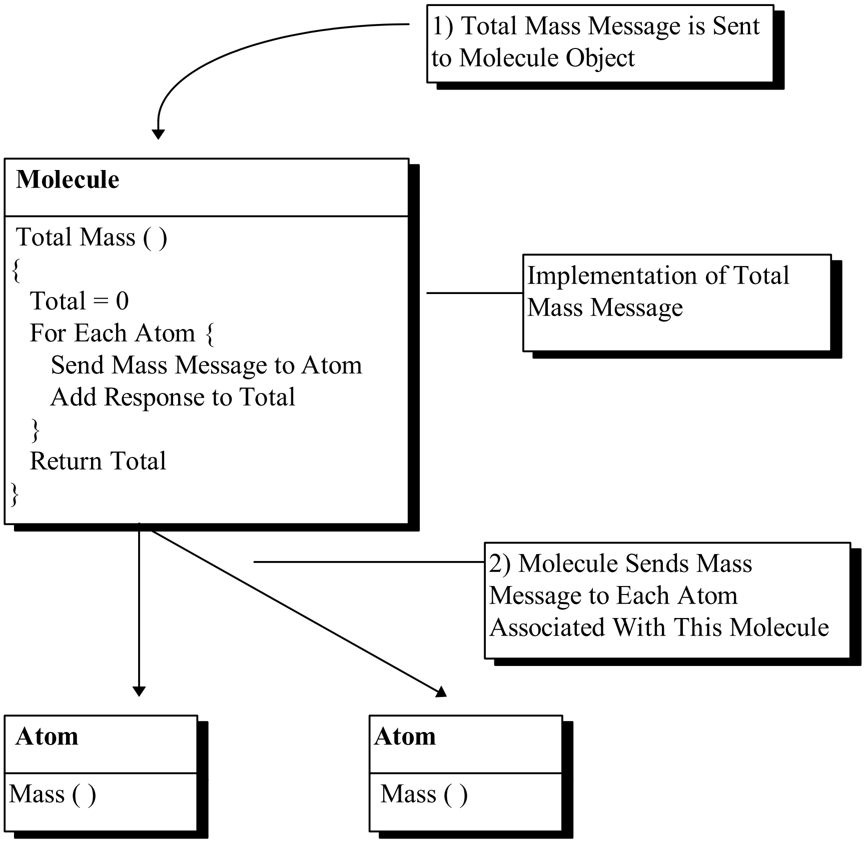


Figure 5-1 Interface For The Molecule Object

The object understands the Add Atom, Remove Atom, and Total Mass messages. These messages form an external interface that other objects can invoke. Notice that the mass units and number of atoms in the molecule are not included as part of the external interface. An object's abstraction need only capture the data necessary to model the entity to other objects. For the example we assume that this abstraction is sufficient.

Messages sent to objects constitute requests for data about that object; objects respond to messages. In Figure 5-1 the Molecule object can respond to the Add Atom, Remove Atom, and Total Mass messages. When a message is sent to an object, the object processes the message and takes appropriate action. This action may involve changing the object's internal state or sending messages to other objects. Figure 5-2 shows an example of the Molecule object responding to the Total Mass message.



1) Total Mass Message is Sent to Molecule Object
Molecule
Implementation of Total Mass Message
2) Molecule Sends Mass Message to Each Atom Associated With This Molecule
Atom
Mass ( )
Atom
Mass ( )

```
Total Mass ( )
{
  Total = 0
  For Each Atom {
    Send Mass Message to Atom
    Add Response to Total
  }
  Return Total
}
```

Figure 5-2 Molecule Object's Processing Of The Total Mass Message

In this example the Molecule object sends the Mass message to two Atom objects associated with the Molecule at an earlier time and aggregates their mass.

We used four object-oriented concepts during the implementation of the PCL system: abstraction, encapsulation, inheritance, and polymorphism. We will define each of these terms and briefly discuss their importance in the development of the PCL system.

Discussion of their benefits will be deferred until specific portions of the system are discussed.

Abstraction is the set of messages to which an object responds. Abstraction was used to define what the object was intended to model and what operations could be performed on the object.

Encapsulation is concealing how an object is internally modeled. Encapsulation and abstraction were used to partition implementation details from their abstraction or external interfaces. This allowed different internal representations of objects to be examined without requiring modifications to other object types.

Inheritance is the ability to derive an object’s interface and implementation from the interface of another object. The object that is derived from is called the parent object; the object that is derived is the child object. Inheritance also allows a child object to selectively processes messages differently that the parent object; in this case the child is considered a specialized type of the parent. The child object can accept the default processing available from the parent object or can override the parent’s implementation. Inheritance is the main idea that differentiates object-oriented programming from structured programming concepts.

Polymorphism is the ability of a message to be processed differently by different objects. Polymorphism allows objects specialized through inheritance to respond to the same messages as the parent object, but process the message differently.

5.2Object-Oriented Solutions To Development Problems

Development problems occur during the creation of any large computer system. In this section we will discuss two problems we encountered and how we used object-oriented solutions to solve them. The first problem we will consider is the duplication of common methods and the other is the lack of support for lists of heterogeneous objects. We will explain each of these below. How these two solutions were used in the development of the PCL system is discussed in sections 5.4-5.8.

Sometimes in a system two functional areas perform similar processing, often duplicating the methods that perform this processing is inefficient for several reasons. First, code maintenance must be performed in several locations. Second, the size of the program is needlessly increased.

We used inheritance to avoid placing duplicate methods in several locations. Our approach involves factoring out the common methods from each object. We call this technique method factoring. The factored methods are placed into a parent object.

Objects that need to use the common methods are derived from the parent object, thus sharing the implementation. Figure 5-3 shows the method factoring for the Link method.

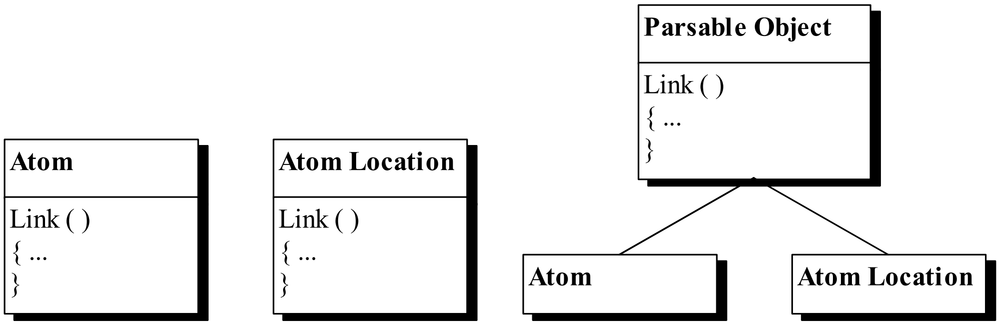


Figure 5-3 The Process Of Factoring A Method To A Parent Class

In the example above, the Atom and Atom Location objects have the same Link method. In order to share this method, we create a new parent object called Parsable Object. The code for the shared method is added to the parent object. Deriving the Atom and Atom Location objects from the Parsable Object allows the sharing of the Link implementation.

There are several benefits associated with factoring similar methods into a parent object. First, the maintenance of the system is simplified because there is only one location to make changes to the shared method. Second, the code is smaller because the method is not duplicated in several locations.

The second problem we will address is the lack of support for lists of heterogeneous objects. During the development of a system it is common to maintain a list of objects. This could be a list of integers, doubles, or structures. Most languages require a list of objects to all be of the same basic type. When several different types of objects must be maintained a heterogeneous list is useful.

Inheritance was used to allow support for lists of heterogeneous objects. Our approach has two parts. The first part involves deriving all the objects that could be placed in the list from a single parent object. The second part consists of having each object derived from the parent override a function that returns the object's type. We call this technique parent factoring. Figure 5-4 shows the parent factoring for the Integer, Double, and Long objects. Each of the three objects have been derived from the Parsable Object parent object. Each has also overridden the Type method.

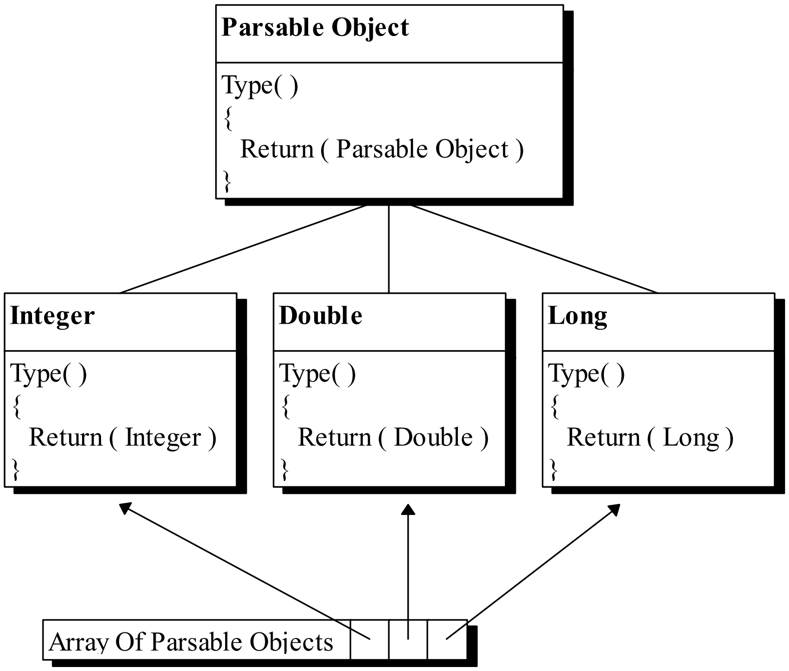


Figure 5-4 The Process Of Parent Factoring

The bottom of Figure 5-4 shows an array of three Parsable Objects. The Integer, Double, and Long objects can all be placed into any array of type Parsable Object. This is because they have all been specialized from this object type. This means that they inherit all the methods of the parent type and they can respond to the same messages.

When an object is retrieved from the list it considered a Parsable Object, not the actual type of the object. This is because the type of the array is Parsable Object. We need the ability to infer the actual type of an object placed in the list. In step two, each object derived from the Parsable Object was required to override a function that returned the object's type. This function allows us determine the type of an object and then cast it back to the correct specialized type. This is necessary because the objects being read in reside in storage and are not able to inform the parser of their types.

In this section we have discussed two of the problems we encountered during the development of the PCL system. These two problems appeared several times during the development of the system. We explained how we used object-oriented solutions to solve them.

5.3Language And Database Selection

In order to be able to leverage the benefits of abstraction, encapsulation, inheritance, and polymorphism we needed to select an object-oriented programming language and database for system development. We selected the C++ language [8] to implement the PCL. This decision was primarily because the Chemists we worked with were already working with C++ and required us to use C++ in this project.

In an effort to select a database, a feasibility study was conducted. The study consisted of evaluating the GemStone and ObjectStore object-oriented databases. Either product could have been used to develop the system. The study demonstrated to us that, at that time, ObjectStore had a better C++ database interface. ObjectStore was selected as the database for the project for this reason.

5.4Structure Of Conceptual and Base Objects

As described in Chapter Four there are two types of objects used in the PCL: conceptual objects and base objects. Conceptual objects model data in the discipline's conceptual schema. Conceptual objects do not specify a physical representation. Base objects are used to create application-specific representations of the conceptual objects. Base objects are attached to a conceptual object to give it a physical representation.

Conceptual objects need the ability to associate base objects with them at run time. This association allows the conceptual object to be modeled in different ways by computational applications. We implemented this by deriving the conceptual and base objects from a parent object called the parsable object. Figure 5-5 show this graphically. We will first discuss why conceptual objects were derived in this manner and then consider the reasons for deriving base objects.

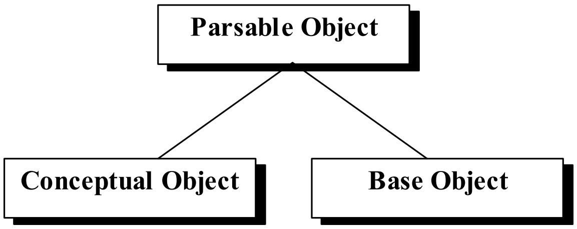


Figure 5-5 Parsable Object With Derived Conceptual And Base Objects

Deriving the conceptual object from a parent class simplifies the linking of base objects with conceptual objects. This is because we can use method factoring to implement the linking in the parent object rather than in each conceptual object.

Deriving the conceptual objects in this manner simplifies the maintenance of the system because there is only one location to make changes to the object linking code. It also makes the code smaller because the link management is not duplicated in several locations. An additional benefit of inheriting the conceptual object from the parsable object is the clear delineation of what functions needed to be implemented for additional conceptual object types. The clear distinction of the interface helps with the maintenance of the object hierarchy as changes are made to the system

Base objects are derived from parsable objects for two reasons. The first reason involves the implementation of the links between conceptual objects and base objects. We used parent factoring to implement the links a list of pointers to parsable objects.

The second reason stems from the implementation of the parsing directives and will be discussed in section 5.6.

5.5Structure Of The PCL Directives

Now that we have discussed the implementation of the conceptual and base objects we turn our attention to the PCL directives. The PCL uses three types of directives to control the loading of experiment data. At different times during the loading process the PCL is instructed to look into the database and retrieve a list of directives to process. This look up of directives is performed using the conceptual object type and possibly the base object type. After this list has been retrieved the PCL processes each directive and returns. Figure 5-6 demonstrates this general procedure.

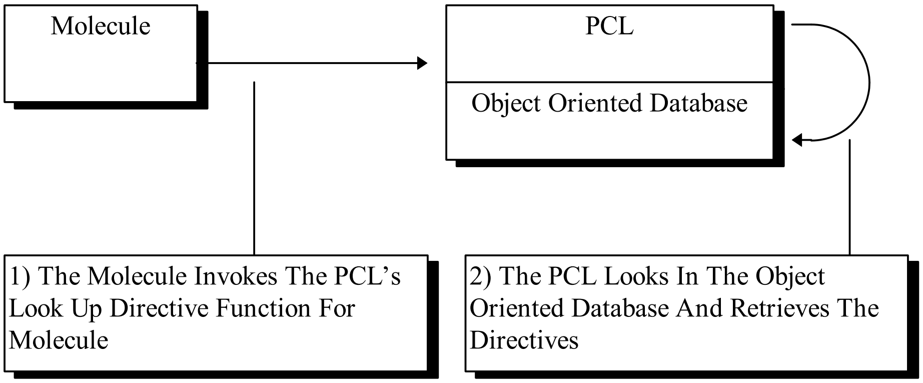

Molecule
PCL
Object Oriented Database
1) The Molecule Invokes The PCL's Look Up Directive Function For Molecule
2) The PCL Looks In The Object Oriented Database And Retrieves The Directives

Figure 5-6 Generic Directive Look Up Procedure

When implementing the parsing directives, we used parent factoring and derived all the directives from a single parent object. This parent object is called the parsing directive base. Parent factoring allowed the list of parsing directives to be stored as a single list of type parsing directive base and simplified the storing and retrieving of parsing directives.

Once the list of type parsing directive base is retrieved it can be iterated through by the PCL. Before processing each directive in the list the PCL first determines the actual type of the directive. This is accomplished by sending the directive a message that has been overridden by each child object. This method returns the type of specialized directive. Parent factoring allows a generic list of parsing directives to be maintained, while allowing each directive to retain it's specialized directive data.

5.6Processing Of Creation Directives

Recall from Chapter Four that the PCL processing begins by invoking the load function for the root of the conceptual object hierarchy. Ultimately the determination of the application's representation of this conceptual object is deferred to the PCL. This is accomplished by the conceptual object invoking the PCL's look-up-creation-directive function and passing the conceptual object whose representation should be determined. This process is shown in Figure 5-7.

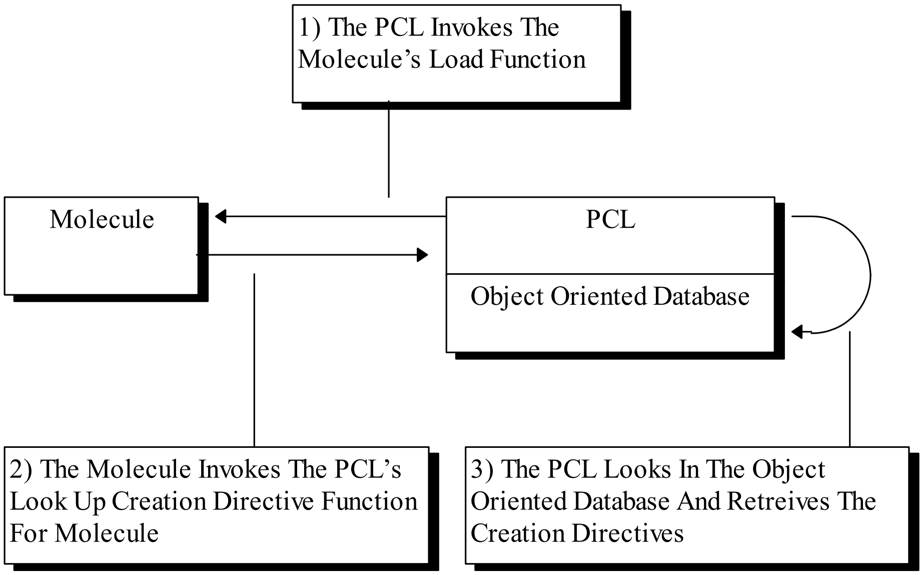

1) The PCL Invokes The Molecule's Load Function
Molecule
PCL
Object Oriented Database
2) The Molecule Invokes The PCL's Look Up Creation Directive Function For Molecule
3) The PCL Looks In The Object Oriented Database And Retreives The Creation Directives

Figure 5-7 Creation Directive Look Up For A Molecule

In order to accomplish this, we needed the ability to pass a conceptual object to the PCL and be able to determine the type of the object passed. This is accomplished by declaring the PCL's look up method to take a parsable object type. This will allow any conceptual object to be passed to this function. This is because the conceptual object type is a specialized from of a parsable object type. Each conceptual object has a method that returns the conceptual object actual type. This allows the PCL to determine the conceptual object's type and look up the proper directives.

5.7Processing Of Parsing Directives

Later in the processing of the conceptual object the PCL is invoked and required to look up the parsing directives for each attribute. Again the location of the parsing directives is deferred to the PCL. This occurs by having the conceptual object invoking the PCL's look-up-parsing-directive function. When this function is invoked it passes the conceptual object and the base object whose parsing directives are to be located. Figure 5-8 graphically represents this processing.

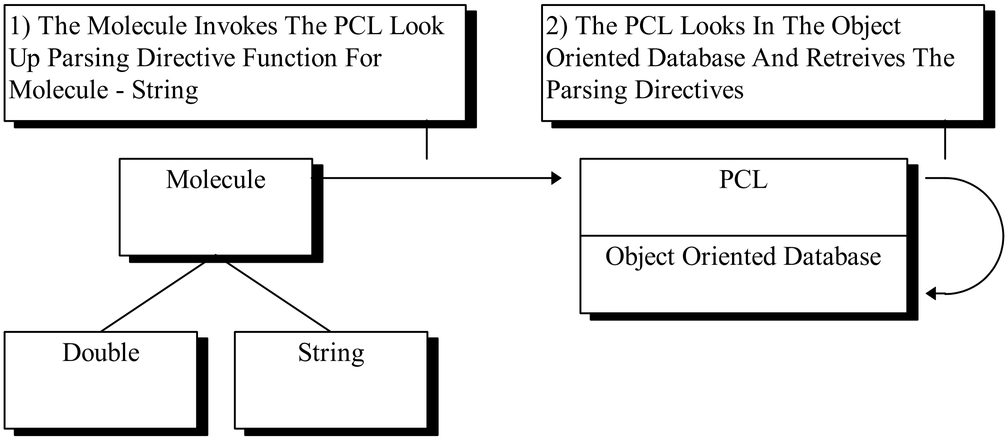

1) The Molecule Invokes The PCL Look Up Parsing Directive Function For Molecule - String
2) The PCL Looks In The Object Oriented Database And Retreives The Parsing Directives
Molecule
PCL
Object Oriented Database
Double
String

Figure 5-8 Parsing Directive Look Up For The String Attribute Of Molecule

In order to accomplish this, we needed the ability to pass a conceptual object and a base object to the PCL. Once this data has been passed to the PCL we need a method of determining the type of each object passed. This problem is similar to the problem noted in the implementation of the creation directives. The only difference in this case is that we are passing two objects to the PCL. We solve this problem by declaring the PCL's look up method to take two parsable object types. This will allow any conceptual object and base object to be passed to this function. This is because the conceptual object and base object types have been specialized from of a parsable object type. Each conceptual and base object has a method that returns the actual type of the object. This allows the PCL to determine the conceptual and base object's type and look up the proper directives.

5.8Conversion Directives

Conversion directives have not been implemented in the current version of the PCL. Their implementation would be very similar to that used in the parsing directives. An implementation of this type would be straight forward extension to the PCL.

5.9Operation Of The PCL

Now that we have discussed the implementation of the different objects that make up the PCL we need to discuss how they work together to load a experiment. The primary implementation tenet was that the PCL directive messages were to be forwarded down the conceptual object hierarchy and be handled at each level. The method we used to send this cascading message was the C++ input operator >>. Each conceptual object is required to understand the input operator message. This message is responsible for invoking the procedures that create the application representation of the conceptual object, parse the attached base objects, and forward the message to the conceptual object’s sub-components. An example of the input operator for an atom is listed in Figure 5-9.

```
// Create Application Representation For Atom - Section One
PCL.CreateApplicationRepresentation ( *this );

// For Each Base Object Attached To The Atom - Section Two
//     Process The Parser Directives For The Base Object
//     Send The Base Object The Input Message

unsigned short Index=0;

for ( Index = 0 ; Index < NumberOfApplicationObjects( ) ; Index++ ) {
  PCL.ProcessDirective( *this, GetApplicationObject ( Index ) );
  GetApplicationObject ( Index ).operator>>( PCL );
}

// Process Parser Directives For Concaputal Object - Element - Section Three
// Send The Element The Input Message
PCL.ProcessDirective ( *this, Element );
Element.operator>> ( PCL );

// Process Parser Directives For The Conceptual Object - AtomLocation
// Send The AtomLocation The InputMessage
PCL.ProcessDirective ( *this , AtomLocation );
AtomLocation.operator>> ( PCL );
```

Figure 5-9 C++ Input Operator For Conceptual Object Atom

The input operator shown has three main sections. The beginning of each section is labeled in a comment. The first section shows the creation application-specific representation of the conceptual object. The second section is the parsing of the base objects that have been associated with the conceptual object. The third section is the forwarding of the input operator to the next conceptual level.

Describing the input operator method is instructive in demonstrating how the directives are processed. For this discussion we assume that an atom conceptually composed of an element and an atom location. When the conceptual object atom is sent the input operator message it must create an application-specific representation of itself. This involves creating and linking base objects to itself. This processing is show in the first section of Figure 5-9 above.

In section two each base object created and linked to the conceptual object in section one is parsed. The parsing involves first positioning the PCL parsing cursor and then instructing the base object to read in a value using the PCL. When a base object receives the input operator message it retrieves data from the PCL at the parsing cursor's location. The base object cannot forward the message to any other objects because the base objects are not composed of additional levels.

The final step in the processing is section three. The input message is sent to the next deeper level in the conceptual hierarchy. At that level the processing of section one through three continues as described above.

Since the PCL is a sub-component of the CCDB project that is not directly used by a computational chemist, Judy Cushing and David Maier reviewed the PCL's implementation. The PCL system's implementation was validated by loading a molecule orbital for the GAMESS application. The molecule orbital is a complex conceptual object comprised two additional conceptual levels and involves the reformatting of complex matrix data. The loading of this conceptual object required the PCL to process all the parsing directives explained in Chapter Four.

5.10Timing Of The PCL

In order to demonstrate the ability of the PCL directives to control the parsing of experiment results we used samples from two different computational chemistry applications. We then created the PCL directives necessary to parse the most complex object contained in the output, namely the molecular orbitals. A production system would require all the information in the optimized molecular configuration to be loaded into the database. The directives required to load the simpler objects were not included because they do not demonstrate any additional functionality.

We selected Gammes and Gaussian as the computational chemistry applications for our tests. This selection was made because experimental runs for these two applications were readily available and are used by our collaborators.

The timings were gathered running on a 80 MHz Intel i486, running Windows NT Server 3.5. The system has 32 Megabytes of memory and contains a Samsung 559 Megabyte drive with a FAT file-system. The PCL system was compiled using Borland C++ 3.1 in large model using 386 instructions, but, no optimizations.

Gammes	3.76 seconds
Gaussian	3.75 seconds

Figure 5-10 Time Required To Process Molecular Orbital Creation And Parsing Directives

These two timings include the initialization of the parser's output file data structure in addition to process the two sets of directives. The time required to process directives for other objects should be similar. If there were ten additional objects to be loaded we would expect thirty seconds to be required to process the creation and parsing directives.

The PCL directives used to parse the outputs are listed in appendix. Included are the input parameters required to produce the optimized molecular configuration.

6Evaluation and Conclusions

In this section we will review the PCL system and summarize what we have learned from this research. We will specifically discuss the results achieved by our research and the effectiveness of the concepts used in the creation of the system.

6.1Confirmation Of Concept

The result achieved by our research was a confirmation of our concept that application-specific model data for the computational sciences can be reused. This reuse can be achieved by transforming application-specific data formats into a generic format. This generic format can then be placed into a database of stored experiment data for later transformation and reuse. We have designed the PCL to be extensible and efficient, although only future testing will verify this.

The common conceptual model has been instrumental to us in this development. The basis of a common conceptual model was used to design processing of creation, parsing, and conversion directives. Thus we have confirmed that a common conceptual model can be useful in developing application which convert information from several different formats, this was predicted by Maier [4].

6.2Conceptual System Structure

The central concept in the PCL system is that of a table-driven interpreter. This interpreter is responsible for the creation of application representations of conceptual objects, the parsing of those objects, and the conversion of the application-specific objects into generic semantically equivalent forms. These three main portions of the interpreter are controlled by tables of instructions. Additions and modifications can be made to these tables without requiring changes to the PCL system. In this manner, the system can support additional computational applications easily.

The concepts and implementation of the creation, parsing, and the conversion directives are similar. This similarity helps make the design and implementation of the system easier to understand, maintain, and extend.

7Analysis and Retrospective

In this section we will analyze the PCL system and provide a retrospective of the project including the pitfalls encountered during implementation.

7.1Innovative Design And Implementation

We have implemented a computational infrastructure that facilitates data management and reuse in the computational sciences. This reuse is centered on a common conceptual model, and a "computational proxy". Reuse is provided by converting application experiment data into a common format that is stored in an object-oriented database. The transformation process is controlled by the PCL. The PCL is an interpreter that uses tables of instructions to construct conceptual data in application-specific format. These application-specific formats are then parsed and converted into a generic form that is placed in the database. The data in this generic format can then be reused by retrieving and converting it into the form required by a specific application. The reuse of data while leaving legacy application file formats unaffected is a unique approach. This approach will be of interest to computational scientists who have large amount of legacy data in application-specific formats and desire to use this data.

7.2Design And Implementation Trade Offs

The implementation of the PCL includes several design trade offs. We implemented the PCL an interpreter in an effort to allow the system to be easily modified and not tied to a single hardware platform. The speed of data conversion and loading is acceptable using this approach.

There are numerous base classes used to implement the PCL. The need for these base classes would be eliminated in a language like Smalltalk, as all objects are automatically derived from a universal type. It might be easier to implement the PCL in such a language.

The generic algorithm used in the parsing directive search engine works well, but, is not efficient. The time required to perform a search become noticeable for large files. The performance could be improved by the use of algorithms in [6], such as the Boyer-Moore algorithm.

8Future Work

In this section we consider future work based on the PCL system. Our work has addressed the problem of data reuse in the computational chemistry field. There are several interesting extensions to our work that could be pursued. The extensions are focused in four areas: system extensions, object hierarchy, directive specification, and directive processing.

8.1Computational Discipline Extensions

One of the most important extensions of our work would be to incorporate it into a production system. This would clearly demonstrate the benefits and advantages and flaws of the system by allowing computational chemists to be more effective with their time. Once the PCL system is incorporated into a production system, support for additional computational programs will become important. There are several additional programs that will need to be incorporated, in addition to GAMESS and Gaussian, including HONDO and MELDF. We conjectured that a generic conversion application saves development time and cost over a customized approach. The adaptation of the PCL to support more modeling programs will also allow the testing of this hypothesis.

We are hopeful that the PCL work will be extended into additional computational science disciplines, specifically Biochemistry and the Earth Sciences. As noted in the introduction our work holds potential benefits for all the computational sciences. The adoption of the PCL work would be accelerated with a successful production system.

The last system extension would be looking into the feasibility of creating a version of the PCL that would process the Computational Chemistry Output Language (CCOL) and the Computational Chemistry Input Language (CCIL). The CCIL is a language that describes how experiment data in the database is converted into a form used by a computational application. It performs the opposite operation of CCOL. There are numerous similarities in the processing of the CCIL and CCOL languages. Research into how these two languages can be implemented in a similar manner would help ease the maintenance of the system.

An innovative extension to the PCL system would be to research data interpolation and extrapolation. This research could be thought of as adding extrapolated or interpolated objects into the system. This work would allow the PCL to be used to aid the analysis of data from varying sources with different data granularities. For example, Earth-orbiting satellites may gather vegetation density data in five mile grids, but another application may desire this data in one mile grids. The new system would be responsible for interpolating a value for the missing grids. When the results based on this analysis became available an error value would be assigned to the results indicating the purity of the data used to arrive at this conclusion.

8.2Object Hierarchy Extensions

An additional extension to the object hierarchy would be a way to group attributes of a conceptual object. Currently the attributes of a conceptual object are determined by the order of the objects in the application-specific representation. This scheme has several limitations, one being that it is error-prone. A way to link conceptual attributes and application-specific representations of those attributes would make the object hierarchies more understandable.

The directives available in the PCL need to be extended. The extension should include additional support for types and conversions. This change would allow applications to represent experiment data in additional formats. Addressers would include new basic and complex object types, such as unsigned character, signed character, and vectors.

8.3Directive Specification Extensions

Currently, the PCL creation, parsing, and conversion directives rarely need to be changed. Their creation is not an easy task and requires precise work and verification by the registrar. An important extension would be to ease the work required to create and specify these directives. The addition of an intermediate non-procedural language for the specification of directives would aid system managers. The language could be textual or graphical. The graphical language would allow the manager to highlight portions of sample output and specify the operations that need to occur during the transformation. From this graphical description the PCL directives could be created and loaded into the database. A simple but powerful extension would be to add support for regular expression searches in the parsing directives.

8.4Directive Processing Extensions

The interpreter currently transforms the computational chemistry experiment data in a reasonable amount of time. When adapting the PCL to additional scientific disciplines the amount of data being converted may increase several fold. If this amount of additional of data does increase, the speed of the interpreter may become a bottleneck. This will especially be true if the source of the data can produce it more quickly that the PCL can consume. In this case some of the PCL design trade-offs will need to be reconsidered. Specifically, the PCL may need to be changed to compile transformation plans into executable programs and update these programs when the PCL directives are changed. In addition to this, the speed of processing conversions in parallel may prove helpful.

9References

[1] P. Bennighoff. Interoperating with DIF Data. Oregon Graduate Institute of Science & Technology, Portland, OR. 1995.

[2] J. Cushing. Computational Proxies: An Object-based Infrastructure for Computational Science. Ph.D. thesis, Department of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, Portland, OR, 1995.

[3] H. Korth and A. Silberschatz. *Database System Concepts.* McGraw-Hill, 1991.

[4] 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 Data Systems.* STP 1214, American Society for Testing and Materials (ASTM), 1994.

[5] 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, 1995.

[6] R. Sedgewick. *Algorithms*, pages 286-289. Addison-Wesley, 1988.

[7] 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.

[8] B. Soustroup. *The C++ Programming Language*. Addison-Wesley, 1991.

[9] J. Ullman. *Principles of Database and Knowledge-base Systems: Volume 1: Classical Database Systems.* Computer Science Press, 1988.

[10] L. Wall and R. L. Schwartz. *Programming PERL*. O'Reilly & Associates, 1990.

10Appendix

10.1Gaussian Creation Directives

# Note: The numbers in the directives can be derived from the experiment information

# or are constant for a version of the computational chemistry application

# Create the application representation of the Molecular Orbital

<u>Molecular Orbital</u>

    6 Atoms

# Create the application representation of the Atom

<u>Atom</u>

    26 Doubles

10.2Gaussian Parsing Directives

<u>Molecular Orbitals</u>

    # Unfold the molecular orbitals

    Skip After First Occurrence of ‘Orbital’

    Next Line

```
Next Line

Unfold Matrix 6 19 3 26


# Copy the atom abbreviation and number

Skip Before First Occurrence of ‘Orbital’

Next Line

Next Line

Denormalize Matrix 6 3 5 4 5 Blank 4 5 6 0 25 1


# Copy the orbital

Skip Before First Occurrence of ‘Orbital’

Next Line

Next Line

Denormalize Matrix 5 3 16 3 14 Blank 3 0 1 0 25 1


# Reposition so an Atom can be read, repeat for each Atom

Skip Before First Occurrence of ‘Orbital’

Yield
```

Atom

```
Skip After First Occurrence of ‘EIGENVALUES’

Skip After First Occurrence of ‘--’
```

```
    # Reposition so Double can be read, repeated for each Double
    Next Line
    Line Offset
    Yield
```

10.3Gaussian Output

```
0 1
C
C 1 RCC
H 2 RCH 1 ANG1
H 2 RCH 1 ANG1 3 180.
H 1 RCH 2 ANG1 3 0.0
H 1 RCH 2 ANG1 3 180.0

RCC=1.334
RCH=1.0802
ANG1=121.646

                Z-Matrix orientation:
 ---------------------------------------------------------
 Center     Atomic              Coordinates (Angstroms)
```

```
Number     Number             X          Y          Z
-----------------------------------------------------
   1         6          0.000000    0.000000    0.000000
   2         6          0.000000    0.000000    1.334000
   3         1          0.919581    0.000000    1.900748
   4         1         -0.919581    0.000000    1.900748
   5         1          0.919581    0.000000   -0.566748
   6         1         -0.919581    0.000000   -0.566748

ORBITAL SYMMETRIES.
     OCCUPIED  (AG) (B1U) (AG) (B1U) (B2U) (AG) (B3G) (B3U)
     VIRTUAL   (B2G) (AG) (B2U) (B1U) (B3G) (B1U) (AG) (B2U)
               (B3U) (B2G) (B1U) (AG) (B3G) (B2U) (B1U) (B3G)
               (AG) (B1U)
 THE ELECTRONIC STATE IS 1-AG.
Alpha eigenvalues --  -11.17072 -11.17068  -1.03155  -0.78772  -0.64316
Alpha eigenvalues --   -0.58548  -0.50058  -0.37542   0.18182   0.29618
Alpha eigenvalues --    0.31209   0.33981   0.43644   0.53790   0.88167
Alpha eigenvalues --    0.92681   0.99297   1.07672   1.10187   1.12548
Alpha eigenvalues --    1.31809   1.35476   1.39767   1.64159   1.66056
Alpha eigenvalues --    1.96291


   Molecular Orbital Coefficients
```

```
                    1        2        3        4        5
                  (AG)     (B1U)    (AG)     (B1U)    (B2U)
  EIGENVALUES --   -11.17072 -11.17068  -1.03155  -0.78772  -0.64316

1 1   C  1S          0.69762   0.69791  -0.16583  -0.12814   0.00000
2        2S  (I)    0.06537   0.07075   0.18160   0.13176   0.00000
3        2PX (I)     0.00000   0.00000   0.00000   0.00000   0.00000
4        2PY (I)     0.00000   0.00000   0.00000   0.00000   0.27881
5        2PZ (I)     0.00158  -0.00186  -0.10650   0.14173   0.00000
6        2S  (O)    -0.03133  -0.06594   0.37110   0.41853   0.00000
7        2PX (O)     0.00000   0.00000   0.00000   0.00000   0.00000
8        2PY (O)     0.00000   0.00000   0.00000   0.00000   0.19373
9        2PZ (O)    -0.00431   0.01506  -0.01624   0.06345   0.00000
10 2   C  1S          0.69762  -0.69791  -0.16583   0.12814   0.00000
11        2S  (I)    0.06537  -0.07075   0.18160  -0.13176   0.00000
12        2PX (I)     0.00000   0.00000   0.00000   0.00000   0.00000
13        2PY (I)     0.00000   0.00000   0.00000   0.00000   0.27881
14        2PZ (I)    -0.00158  -0.00186   0.10650   0.14173   0.00000
15        2S  (O)    -0.03133   0.06594   0.37110  -0.41853   0.00000
16        2PX (O)     0.00000   0.00000   0.00000   0.00000   0.00000
17        2PY (O)     0.00000   0.00000   0.00000   0.00000   0.19373
18        2PZ (O)     0.00431   0.01506   0.01624   0.06345   0.00000
```

```
19 3   H  1S  (I)    -0.00176   0.00026   0.07802  -0.13740   0.14677

20        1S  (O)     0.00958  -0.00797   0.00528  -0.06737   0.10969

21 4   H  1S  (I)    -0.00176   0.00026   0.07802  -0.13740  -0.14677

22        1S  (O)     0.00958  -0.00797   0.00528  -0.06737  -0.10969

23 5   H  1S  (I)    -0.00176  -0.00026   0.07802   0.13740   0.14677

24        1S  (O)     0.00958   0.00797   0.00528   0.06737   0.10969

25 6   H  1S  (I)    -0.00176  -0.00026   0.07802   0.13740  -0.14677

26        1S  (O)     0.00958   0.00797   0.00528   0.06737  -0.10969

                        6         7         8         9        10

                     (AG)      (B3G)     (B3U)     (B2G)     (AG)

  EIGENVALUES --     -0.58548  -0.50058  -0.37542   0.18182   0.29618

 1 1   C  1S          0.00852   0.00000   0.00000   0.00000  -0.09105

 2        2S  (I)    -0.02120   0.00000   0.00000   0.00000   0.03129

 3        2PX (I)     0.00000   0.00000   0.32018   0.30382   0.00000

 4        2PY (I)     0.00000   0.26045   0.00000   0.00000   0.00000

 5        2PZ (I)     0.36314   0.00000   0.00000   0.00000   0.13028

 6        2S  (O)     0.02475   0.00000   0.00000   0.00000   1.37625

 7        2PX (O)     0.00000   0.00000   0.37551   0.75082   0.00000

 8        2PY (O)     0.00000   0.27526   0.00000   0.00000   0.00000

 9        2PZ (O)     0.22453   0.00000   0.00000   0.00000   0.62103

10 2   C  1S          0.00852   0.00000   0.00000   0.00000  -0.09105

11        2S  (I)    -0.02120   0.00000   0.00000   0.00000   0.03129

12        2PX (I)     0.00000   0.00000   0.32018  -0.30382   0.00000
```

```
13        2PY (I)     0.00000  -0.26045   0.00000   0.00000   0.00000

14        2PZ (I)    -0.36314   0.00000   0.00000   0.00000  -0.13028

15        2S  (O)     0.02475   0.00000   0.00000   0.00000   1.37625

16        2PX (O)     0.00000   0.00000   0.37551  -0.75082   0.00000

17        2PY (O)     0.00000  -0.27526   0.00000   0.00000   0.00000

18        2PZ (O)    -0.22453   0.00000   0.00000   0.00000  -0.62103

19 3   H  1S  (I)     0.11847  -0.17857   0.00000   0.00000  -0.01761

20        1S  (O)     0.11078  -0.15647   0.00000   0.00000  -0.95260

21 4   H  1S  (I)     0.11847   0.17857   0.00000   0.00000  -0.01761

22        1S  (O)     0.11078   0.15647   0.00000   0.00000  -0.95260

23 5   H  1S  (I)     0.11847   0.17857   0.00000   0.00000  -0.01761

24        1S  (O)     0.11078   0.15647   0.00000   0.00000  -0.95260

25 6   H  1S  (I)     0.11847  -0.17857   0.00000   0.00000  -0.01761

26        1S  (O)     0.11078  -0.15647   0.00000   0.00000  -0.95260

                      11         12         13         14         15

                    (B2U)      (B1U)      (B3G)      (B1U)      (AG)

   EIGENVALUES --     0.31209    0.33981    0.43644    0.53790    0.88167

 1 1   C  1S          0.00000  -0.12205   0.00000   0.09363   0.01653

 2        2S  (I)     0.00000   0.04686   0.00000   0.00410   0.10140

 3        2PX (I)     0.00000   0.00000   0.00000   0.00000   0.00000

 4        2PY (I)    -0.21783   0.00000   0.24038   0.00000   0.00000

 5        2PZ (I)     0.00000   0.08606   0.00000   0.15187  -0.65071

 6        2S  (O)     0.00000   1.60267   0.00000  -2.54368   0.42451
 7        2PX (O)     0.00000   0.00000   0.00000   0.00000   0.00000

 8        2PY (O)    -0.80291   0.00000   1.63487   0.00000   0.00000

 9        2PZ (O)     0.00000   0.29286   0.00000   2.56435   1.03012

10 2   C  1S          0.00000   0.12205   0.00000  -0.09363   0.01653

11         2S  (I)    0.00000  -0.04686   0.00000  -0.00410   0.10140

12         2PX (I)    0.00000   0.00000   0.00000   0.00000   0.00000

13         2PY (I)   -0.21783   0.00000  -0.24038   0.00000   0.00000

14         2PZ (I)    0.00000   0.08606   0.00000   0.15187   0.65071

15         2S  (O)    0.00000  -1.60267   0.00000   2.54368   0.42451

16         2PX (O)    0.00000   0.00000   0.00000   0.00000   0.00000

17         2PY (O)   -0.80291   0.00000  -1.63487   0.00000   0.00000

18         2PZ (O)    0.00000   0.29286   0.00000   2.56435  -1.03012

19 3   H  1S  (I)     0.05174   0.02451  -0.03947   0.06882  -0.13648

20         1S  (O)    0.99778   0.98923   1.38452   0.42093  -0.12698

21 4   H  1S  (I)    -0.05174   0.02451   0.03947   0.06882  -0.13648

22         1S  (O)   -0.99778   0.98923  -1.38452   0.42093  -0.12698

23 5   H  1S  (I)     0.05174  -0.02451   0.03947  -0.06882  -0.13648

24         1S  (O)    0.99778  -0.98923  -1.38452  -0.42093  -0.12698

25 6   H  1S  (I)    -0.05174  -0.02451  -0.03947  -0.06882  -0.13648

26         1S  (O)   -0.99778  -0.98923   1.38452  -0.42093  -0.12698

                       16        17        18        19        20

                     (B2U)     (B3U)     (B2G)     (B1U)     (AG)

   EIGENVALUES --     0.92681   0.99297   1.07672   1.10187   1.12548
```

```
1 1   C  1S          0.00000   0.00000   0.00000   0.09045   0.03667
2        2S  (I)     0.00000   0.00000   0.00000  -0.00294   0.36011
3        2PX (I)     0.00000   0.76482  -0.79488   0.00000   0.00000
4        2PY (I)    -0.43116   0.00000   0.00000   0.00000   0.00000
5        2PZ (I)     0.00000   0.00000   0.00000   0.57779  -0.20554
6        2S  (O)     0.00000   0.00000   0.00000  -0.27814  -0.44815
7        2PX (O)     0.00000  -0.58017   0.94532   0.00000   0.00000
8        2PY (O)     0.70083   0.00000   0.00000   0.00000   0.00000
9        2PZ (O)     0.00000   0.00000   0.00000  -0.51447  -0.06188
10 2   C  1S         0.00000   0.00000   0.00000  -0.09045   0.03667
11       2S  (I)     0.00000   0.00000   0.00000   0.00294   0.36011
12       2PX (I)     0.00000   0.76482   0.79488   0.00000   0.00000
13       2PY (I)    -0.43116   0.00000   0.00000   0.00000   0.00000
14       2PZ (I)     0.00000   0.00000   0.00000   0.57779   0.20554
15       2S  (O)     0.00000   0.00000   0.00000   0.27814  -0.44815
16       2PX (O)     0.00000  -0.58017  -0.94532   0.00000   0.00000
17       2PY (O)     0.70083   0.00000   0.00000   0.00000   0.00000
18       2PZ (O)     0.00000   0.00000   0.00000  -0.51447   0.06188
19 3   H  1S  (I)   -0.44021   0.00000   0.00000  -0.45836   0.60306
20       1S  (O)     0.02782   0.00000   0.00000   0.09416  -0.23463
21 4   H  1S  (I)    0.44021   0.00000   0.00000  -0.45836   0.60306
22       1S  (O)    -0.02782   0.00000   0.00000   0.09416  -0.23463
23 5   H  1S  (I)   -0.44021   0.00000   0.00000   0.45836   0.60306
```

```
24        1S  (O)     0.02782   0.00000   0.00000  -0.09416  -0.23463

25 6   H  1S  (I)     0.44021   0.00000   0.00000   0.45836   0.60306

26        1S  (O)    -0.02782   0.00000   0.00000  -0.09416  -0.23463

                    21        22        23        24        25

                  (B3G)     (B2U)     (B1U)     (B3G)     (AG)

   EIGENVALUES --     1.31809   1.35476   1.39767   1.64159   1.66056

 1 1   C  1S          0.00000   0.00000   0.02496   0.00000   0.03378

 2        2S  (I)     0.00000   0.00000  -0.12310   0.00000  -1.20393

 3        2PX (I)     0.00000   0.00000   0.00000   0.00000   0.00000

 4        2PY (I)    -0.82019   0.68651   0.00000  -0.31427   0.00000

 5        2PZ (I)     0.00000   0.00000  -0.71567   0.00000  -0.16391

 6        2S  (O)     0.00000   0.00000  -0.36333   0.00000   1.65425

 7        2PX (O)     0.00000   0.00000   0.00000   0.00000   0.00000

 8        2PY (O)     1.82948  -0.84236   0.00000   2.48277   0.00000

 9        2PZ (O)     0.00000   0.00000   1.31171   0.00000   0.33917

10 2   C  1S          0.00000   0.00000  -0.02496   0.00000   0.03378

11        2S  (I)     0.00000   0.00000   0.12310   0.00000  -1.20393

12        2PX (I)     0.00000   0.00000   0.00000   0.00000   0.00000

13        2PY (I)     0.82019   0.68651   0.00000   0.31427   0.00000

14        2PZ (I)     0.00000   0.00000  -0.71567   0.00000   0.16391

15        2S  (O)     0.00000   0.00000   0.36333   0.00000   1.65425

16        2PX (O)     0.00000   0.00000   0.00000   0.00000   0.00000

17        2PY (O)    -1.82948  -0.84236   0.00000  -2.48277   0.00000
```

```
18        2PZ (O)    0.00000   0.00000   1.31171   0.00000  -0.33917

19 3   H  1S  (I)   -0.29255  -0.47854  -0.47437   0.68933   0.20134

20        1S  (O)    1.06586   0.88470   0.75739   0.59615  -0.66776

21 4   H  1S  (I)    0.29255   0.47854  -0.47437  -0.68933   0.20134

22        1S  (O)   -1.06586  -0.88470   0.75739  -0.59615  -0.66776

23 5   H  1S  (I)    0.29255  -0.47854   0.47437  -0.68933   0.20134

24        1S  (O)   -1.06586   0.88470  -0.75739  -0.59615  -0.66776

25 6   H  1S  (I)   -0.29255   0.47854   0.47437   0.68933   0.20134

26        1S  (O)    1.06586  -0.88470  -0.75739   0.59615  -0.66776

                    26    #

                   (B1U)

  EIGENVALUES --     1.96291

 1 1   C  1S        -0.00250

 2       2S  (I)   -1.41566

 3       2PX (I)     0.00000

 4       2PY (I)     0.00000

 5       2PZ (I)    0.08663

 6       2S  (O)    3.82153

 7       2PX (O)     0.00000

 8       2PY (O)     0.00000

 9       2PZ (O)    -1.14177

10 2   C  1S          0.00250

11        2S  (I)    1.41566
```

```
12        2PX (I)     0.00000

13        2PY (I)     0.00000

14        2PZ (I)     0.08663

15        2S  (O)    -3.82153

16        2PX (O)     0.00000

17        2PY (O)     0.00000

18        2PZ (O)    -1.14177

19 3   H  1S  (I)     0.11246

20        1S  (O)     0.35773

21 4   H  1S  (I)     0.11246

22        1S  (O)     0.35773

23 5   H  1S  (I)    -0.11246

24        1S  (O)    -0.35773

25 6   H  1S  (I)    -0.11246

26        1S  (O)    -0.35773


   DENSITY MATRIX.

----------------------------------------------------------------------

Total atomic charges:

           1

 1  C   -0.425338

 2  C   -0.425338

 3  H    0.212669
 4  H    0.212669

 5  H    0.212669

 6  H    0.212669


   nuclear repulsion energy   33.4010108717 Hartrees.


 26 basis functions      42 primitive gaussians


Dipole moment (Debye):

  X=     0.0000    Y=     0.0000    Z=    -1.2860  Tot=     1.2860

Quadrupole moment (Debye-Ang):

 XX=    -4.6255   YY=    -4.6255   ZZ=    -3.4983

 XY=     0.0000   XZ=     0.0000   YZ=     0.0000


Dipole moment (Debye):

  X=     0.0000    Y=     0.0000    Z=     0.0000  Tot=     0.0000

Quadrupole moment (Debye-Ang):

 XX=   -15.7191   YY=   -12.3174   ZZ=   -12.1050

 XY=     0.0000   XZ=     0.0000   YZ=     0.0000

Octapole moment (Debye-Ang**2):

 XXX=     0.0000  YYY=     0.0000  ZZZ=     0.0000  XYY=     0.0000

 XXY=     0.0000  XXZ=     0.0000  XZZ=     0.0000  YZZ=     0.0000

 YYZ=     0.0000  XYZ=     0.0000
```

```
Hexadecapole moment (Debye-Ang**3):

XXXX=   -16.4808 YYYY=   -24.9877 ZZZZ=   -65.5232 XXXY=     0.0000

XXXZ=     0.0000 YYYX=     0.0000 YYYZ=     0.0000 ZZZX=     0.0000

ZZZY=     0.0000 XXYY=    -7.5604 XXZZ=   -14.8582 YYZZ=   -12.3932

XXYZ=     0.0000 YYXZ=     0.0000 ZZXY=     0.0000


 -------------------------------------------------------------------


GradGradGradGradGradGradGradGradGradGradGradGradGradGradGradGradGradGrad

 -------------------------------------------------------------------

                 Standard orientation:

 ---------------------------------------------------------

Center     Atomic                   Forces (Hartrees/Bohr)

Number      Number              X              Y              Z


    1         6           0.000000000    0.000000000    0.020106520

    2         6           0.000000000    0.000000000   -0.020106520

    3         1          -0.004582715    0.000000000   -0.002473539

    4         1           0.004582715    0.000000000   -0.002473539

    5         1          -0.004582715    0.000000000    0.002473539

    6         1           0.004582715    0.000000000    0.002473539
```

10.4Gammes Creation Directives

# Note: The numbers in the directives can be derived from the experiment information

# or are constant for a version of the computational chemistry application

# Create the application representation of the Molecular Orbital

<u>Molecular Orbital</u>

    6 Atoms

# Create the application representation of the Atom

<u>Atom</u>

    38 Doubles

10.5Gammes Parsing Directives

<u>Molecular Orbitals</u>

    # Unfold the molecular orbitals

    Skip After First Occurrence of ‘MOLECULAR’

    Next Line

    Next Line

    Next Line

```
Next Line
Unfold Matrix 2 16 4 38

# Copy the atom abbreviation and number
Skip Before First Occurrence of ‘MOLECULAR’
Next Line
Next Line
Next Line
Next Line
Next Line
Next Line
Denormalize Matrix 4 0 9 1 9 Blank 1 9 4 0 37 1

# Copy the orbital
Skip Before First Occurrence of ‘MOLECULAR’
Next Line
Next Line
Next Line
Next Line
Next Line
Next Line
Denormalize Matrix 3 0 14 0 13 Blank 0 13 1 0 37 1
```

```
# Reposition so an Atom can be read, repeat for each Atom
Skip Before First Occurrence of ‘MOLECULAR’
Yield
```

Atom

```
Skip Before First Occurrence of ‘MOLECULAR’
Next Line
Next Line
Next Line
Next Line
Next Line
Next Line

# Reposition so Double can be read, repeated for each Double
Next Line
Line Offset
Yield
```

10.6Gammes Output

```
 TOTAL NUMBER OF BASIS FUNCTIONS     =   74
 FINAL ENERGY IS      -78.0561311759 AFTER  12 ITERATIONS


          ---------------------
          ELECTROSTATIC MOMENTS
          ---------------------


 POINT   1           X           Y           Z (BOHR)    CHARGE
                0.000000    0.000000    0.000000        0.00 (A.U.)
         DX          DY          DZ         /D/  (DEBYE)
     0.000000    0.000000    0.000000    0.000000


          ---------------------
          ELECTROSTATIC MOMENTS
          ---------------------


 POINT   1           X           Y           Z (BOHR)    CHARGE
                0.000000    0.000000    0.087542        0.00 (A.U.)
         DX          DY          DZ         /D/  (DEBYE)
     0.000000    0.000000    1.285987    1.285987
  QXX        QYY        QZZ        QXY        QXZ        QYZ  (BUCKINGHAMS) -
0.622276   -
0.622276    1.244552    0.000000    0.000000    0.000000
 ...... END OF PROPERTY EVALUATION ......
```

```
      STEP CPU TIME =    1.84  TOTAL CPU TIME =   130.09   (   2.2 MIN) IS 94.96
PERCENT OF
REAL TIME OF    137.00
  174978 WORDS OF DYNAMIC MEMORY USED
 EXECUTION OF GAMESS TERMINATED NORMALLY Fri Aug  7 15:21:04 1992


        GRADIENT OF THE ENERGY
        ----------------------


    ATOM            E'X            E'Y            E'Z
   1 C          0.000194871     0.000000000     0.000000000
   2 C         -0.000194871     0.000000000     0.000000000
   3 H         -0.000012376    -0.000030270     0.000000000
   4 H          0.000012376    -0.000030270     0.000000000
   5 H         -0.000012376     0.000030270     0.000000000
   6 H          0.000012376     0.000030270     0.000000000

 ...... END OF 2-ELECTRON GRADIENT ......
```

```
      STEP CPU TIME =  228.12  TOTAL CPU TIME =   361.23   (   6.0 MIN) IS 98.70
PERCENT
OF REAL TIME OF    366.00


        MAXIMUM COMPONENT =   0.000194871
            RMS GRADIENT =   0.000066761
 ..... END OF SINGLE POINT GRADIENT .....


        MOLECULAR ORBITALS
        ------------------


                 1         2         3         4         5         6         7         8         9        10
              -11.1794   -11.1790    -1.0472    -0.7972    -0.6550    -0.5991    -0.5078    -
0.3844
0.0506     0.0629
                  A          A          A          A          A          A          A          A          A
A
1  H    1 S  -0.000885  -0.000008   0.076201   0.137654  -0.146789  -0.120192  -
0.180163
0.000000   0.011374   0.009358
```

```
   2  H      S   0.006193   0.006800   0.011909   0.067802  -0.101247  -0.098893  -
0.126968

0.000000  -0.024885  -0.049591

   3  H      S  -0.009440  -0.003157   0.033003   0.023562  -0.031374  -0.005345
0.036526

0.000000  -1.300060  -2.181759

   4  H    2 S  -0.000885  -0.000008   0.076201   0.137654   0.146789  -0.120192
0.180163

0.000000   0.011374   0.009358

   5  H      S   0.006193   0.006800   0.011909   0.067802   0.101247  -0.098893
0.126968

0.000000  -0.024885  -0.049591

   6  H      S  -0.009440  -0.003157   0.033003   0.023562   0.031374  -0.005345  -
0.036526

0.000000  -1.300060  -2.181759

   7  C    3 S   0.697865   0.697977  -0.167342  -0.128405   0.000000  -0.007556
0.000000

0.000000  -0.019737  -0.023338

   8  C      S   0.067681   0.071494   0.180421   0.131430   0.000000   0.020580
0.000000

0.000000   0.037107   0.029252

   9  C      X  -0.001270   0.001987   0.110436  -0.143379   0.000000   0.365347
0.000000
```

```
0.000000  -0.048198  -0.025495
  10  C      Y   0.000000   0.000000   0.000000   0.000000   0.281317   0.000000
0.261388
0.000000   0.000000   0.000000
  11  C      Z   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000
0.000000
0.319790   0.000000   0.000000
  12  C      S  -0.035730  -0.069076   0.379102   0.426661   0.000000  -0.034121
0.000000
0.000000   0.047718   0.168226
  13  C      X   0.003347  -0.016034   0.019167  -0.062108   0.000000   0.224264
0.000000
0.000000  -0.059446   0.066616
  14  C      Y   0.000000   0.000000   0.000000   0.000000   0.197203   0.000000
0.299992
0.000000   0.000000   0.000000
  15  C      Z   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000
0.000000
0.345201   0.000000   0.000000
  16  C      S   0.023040   0.094930  -0.074646  -0.325696   0.000000  -0.003667
0.000000
0.000001   2.109155   4.408334
```

```
  17  C      X  -0.001024   0.022965   0.002612  -0.070989   0.000000   0.000681
0.000000
0.000000  -0.432063  -0.213937
  18  C      Y   0.000000   0.000000   0.000000   0.000000  -0.012496   0.000000
0.101566
0.000000   0.000000   0.000000
  19  C      Z   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000
0.000000
0.051772  -0.000001   0.000000
  20  H    4 S  -0.000885   0.000008   0.076201  -0.137654  -0.146789  -0.120192
0.180163
0.000000   0.011374  -0.009358
  21  H      S   0.006193  -0.006800   0.011909  -0.067802  -0.101247  -0.098893
0.126968
0.000000  -0.024885   0.049591
  22  H      S  -0.009440   0.003157   0.033003  -0.023562  -0.031374  -0.005345  -
0.036526
0.000000  -1.300060   2.181759
  23  H    5 S  -0.000885   0.000008   0.076201  -0.137654   0.146789  -0.120192  -
0.180163
0.000000   0.011374  -0.009358
  24  H      S   0.006193  -0.006800   0.011909  -0.067802   0.101247  -0.098893  -
0.126968
```

```
0.000000  -0.024885   0.049591
  25  H      S  -0.009440   0.003157   0.033003  -0.023562   0.031374  -0.005345
0.036526
0.000000  -1.300060   2.181759
  26  C    6 S   0.697865  -0.697977  -0.167342   0.128405   0.000000  -0.007556
0.000000
0.000000  -0.019737   0.023338
  27  C      S   0.067681  -0.071494   0.180421  -0.131430   0.000000   0.020580
0.000000
0.000000   0.037107  -0.029252
  28  C      X   0.001270   0.001987  -0.110436  -0.143379   0.000000  -0.365347
0.000000
0.000000   0.048198  -0.025495
  29  C      Y   0.000000   0.000000   0.000000   0.000000   0.281317   0.000000  -
0.261388
0.000000   0.000000   0.000000
  30  C      Z   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000
0.000000
0.319790   0.000000   0.000000
  31  C      S  -0.035730   0.069076   0.379102  -0.426661   0.000000  -0.034121
0.000000
0.000000   0.047718  -0.168226
```

```
  32  C      X  -0.003347  -0.016034  -0.019167  -0.062108   0.000000  -0.224264
0.000000
0.000000   0.059446   0.066616
  33  C      Y   0.000000   0.000000   0.000000   0.000000   0.197203   0.000000  -
0.299992
0.000000   0.000000   0.000000
  34  C      Z   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000
0.000000
0.345201   0.000000   0.000000
  35  C      S   0.023040  -0.094930  -0.074646   0.325696   0.000000  -0.003667
0.000000  -
0.000001   2.109155  -4.408334
  36  C      X   0.001024   0.022965  -0.002612  -0.070989   0.000000  -0.000681
0.000000
0.000000   0.432063  -0.213937
  37  C      Y   0.000000   0.000000   0.000000   0.000000  -0.012496   0.000000  -
0.101566
0.000000   0.000000   0.000000
  38  C      Z   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000
0.000000
0.051772   0.000001   0.000000

                   11        12        13        14        15        16        17        18
```

```
                 0.0644    0.0855    0.1057    0.1416    0.1485    0.1794    0.2223
0.2271
                  A         A         A         A         A         A         A         A
   1  H    1 S  -0.006642  -0.019541   0.000000   0.000000   0.010285   0.010967  -
0.013302
0.001006
   2  H      S   0.033673   0.159217   0.000000   0.000000   0.116543   0.016273  -
0.374957
0.168627
   3  H      S   2.583125   6.204936   0.000000   0.000002   0.749773   3.392583  -
1.645794
3.523880
   4  H    2 S   0.006642   0.019541   0.000000   0.000000   0.010285   0.010967  -
0.013302  -
0.001006
   5  H      S  -0.033673  -0.159217   0.000000   0.000000   0.116543   0.016273  -
0.374957  -
0.168627
   6  H      S  -2.583125  -6.204936   0.000002   0.000002   0.749773   3.392583  -
1.645794  -
3.523880
   7  C    3 S   0.000000   0.000000   0.000000   0.000000   0.011124  -0.014849
0.053668
```

```
0.000000
   8  C      S   0.000000   0.000000   0.000000   0.000000  -0.001274   0.020942  -
0.013858
0.000000
   9  C      X   0.000000   0.000000   0.000000   0.000000  -0.049655   0.007771
0.033561
0.000000
  10  C      Y   0.063482   0.036137   0.000000   0.000000   0.000000   0.000000
0.000000  -
0.111203
  11  C      Z   0.000000   0.000000   0.174261  -0.104903   0.000000   0.000000
0.000000
0.000000
  12  C      S   0.000000   0.000000   0.000000   0.000000  -0.151453   0.043641  -
0.550773
0.000000
  13  C      X   0.000000   0.000000   0.000000   0.000000  -0.066363  -0.035358
0.003438
0.000000
  14  C      Y   0.041498   0.136666   0.000000   0.000000   0.000000   0.000000
0.000000  -
0.271277
```

```
  15  C      Z   0.000000   0.000000   0.268473  -0.283838   0.000000   0.000000
0.000001
0.000000
  16  C      S   0.000000   0.000000   0.000003   0.000028  -1.794034  34.339009
4.057977
0.000000
  17  C      X   0.000000   0.000000   0.000003   0.000010   1.923014  12.551861  -
0.637850
0.000000
  18  C      Y   1.113523   4.695565  -0.000001   0.000000   0.000000   0.000000
0.000000
2.553448
  19  C      Z   0.000000   0.000000   1.548186   0.597545  -0.000001   0.000000  -
0.000001
0.000000
  20  H    4 S  -0.006642   0.019541   0.000000   0.000000   0.010285  -0.010967  -
0.013302
0.001006
  21  H      S   0.033673  -0.159217   0.000000   0.000000   0.116543  -0.016273  -
0.374957
0.168627
  22  H      S   2.583125  -6.204936   0.000001  -0.000002   0.749773  -3.392583  -
1.645794
```

```
3.523880
  23  H    5 S   0.006642  -0.019541   0.000000   0.000000   0.010285  -0.010967  -
0.013302  -
0.001006
  24  H      S  -0.033673   0.159217   0.000000   0.000000   0.116543  -0.016273  -
0.374957  -
0.168627
  25  H      S  -2.583125   6.204936  -0.000001  -0.000002   0.749773  -3.392583  -
1.645794  -
3.523880
  26  C    6 S   0.000000   0.000000   0.000000   0.000000   0.011124   0.014849
0.053668
0.000000
  27  C      S   0.000000   0.000000   0.000000   0.000000  -0.001274  -0.020942  -
0.013858
0.000000
  28  C      X   0.000000   0.000000   0.000000   0.000000   0.049655   0.007771  -
0.033561
0.000000
  29  C      Y   0.063482  -0.036137   0.000000   0.000000   0.000000   0.000000
0.000000  -
0.111203
```

```
  30  C      Z   0.000000   0.000000  -0.174261  -0.104903   0.000000   0.000000
0.000000
0.000000
  31  C      S   0.000000   0.000000   0.000000   0.000000  -0.151453  -0.043641  -
0.550773
0.000000
  32  C      X   0.000000   0.000000   0.000000   0.000000   0.066363  -0.035358  -
0.003438
0.000000
  33  C      Y   0.041498  -0.136666   0.000000   0.000000   0.000000   0.000000
0.000000  -
0.271277
  34  C      Z   0.000000   0.000000  -0.268473  -0.283838   0.000000   0.000000  -
0.000001
0.000000
  35  C      S   0.000000   0.000000  -0.000006  -0.000028  -1.794034 -34.339009
4.057977
0.000000
  36  C      X   0.000000   0.000000   0.000000   0.000010  -1.923014  12.551861
0.637850
0.000000
  37  C      Y   1.113523  -4.695565   0.000001   0.000000   0.000000   0.000000
0.000000
```

```
2.553448

  38  C      Z   0.000000   0.000000  -1.548186   0.597545   0.000001   0.000000

0.000001

0.000000


          -----------------

          ENERGY COMPONENTS

          -----------------



 COORDINATES OF ALL ATOMS ARE (ANGS)

   ATOM   CHARGE       X              Y              Z

 ------------------------------------------------------------

 H           1.0  -1.2265061870  -0.9134808718   0.0000000369

 H           1.0  -1.2265061868   0.9134808717  -0.0000000630

 C           6.0  -0.6602791538   0.0000000000  -0.0000000383

 H           1.0   1.2265061871  -0.9134808718   0.0000000632

 H           1.0   1.2265061868   0.9134808718  -0.0000000368

 C           6.0   0.6602791538   0.0000000000   0.0000000307
```