2008

# Development of high performance scientific components for interoperability of computing packages

Teena Pratap Gulabani
*Iowa State University*

# Development of high performance scientific components for interoperability of computing packages

by

Teena Pratap Gulabani

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Masha Sosonkina, Co-Major Professor
Les Miller, Co-Major Professor
Theresa L Windus
Simanta Mitra

Iowa State University

Ames, Iowa

2008

# DEDICATION

With gratitude for their unconditional love and support, I dedicate this thesis to my loving parents Neelam and Pratap Gulabani and my dear sister, Daya Gulabani.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# ABSTRACT

Three major high performance quantum chemistry computational packages, NWChem, GAMESS and MPQC have been developed by different research efforts following different design patterns. The goal is to achieve interoperability among these packages by overcoming the challenges caused by the different communication patterns and software design of each of these packages. A chemistry algorithm is hard to develop as well as being a time consuming process; integration of large quantum chemistry packages will allow resource sharing and thus avoid reinvention of the wheel. Creating connections between these incompatible packages is the major motivation of the proposed work. This interoperability is achieved by bringing the benefits of Component Based Software Engineering through a plug-and-play component framework called Common Component Architecture (CCA). In this thesis, I present a strategy and process used for interfacing two widely used and important computational chemistry methodologies: Quantum Mechanics and Molecular Mechanics. To show the feasibility of the proposed approach the Tuning and Analysis Utility (TAU) has been coupled with NWChem code and its CCA components Results show that the overhead is negligible when compared to the ease and potential of organizing and coping with large-scale software applications.

# CHAPTER 1. INTRODUCTION

Software reuse is the process of building systems using existing software rather than building the systems from scratch. Quantum Chemistry packages like NWChem [1,2], GAMESS [3] and MPQC [4] are developed to perform high-performance scientific simulations. These packages are developed and maintained by different research scientists. It is difficult for a single research group to effectively develop solutions for all of the methods one would desire. Also, since each research group needs some similar capability, it leads to duplication of efforts. When a new capability is to be added to a package, it is nice to have a gateway via which packages can reuse existing tested capabilities. Additionally, research groups often optimize their software for a particular hardware. Thus, the different algorithms for the same functionality may run better on different platforms. Being able to use the best algorithm for a particular platform increases the overall throughput of the science.

This leads to the potential usage of component software engineering. Commercial component based software engineering practices such as COM [5], EJB [6] and CORBA [7] exists in the market, which present an approach for managing the increasing complexity of business packages. In this research work, I use a similar framework known as the Common Component Architecture (CCA) [8], which is targeted at high-performance scientific development. CCA allows research scientists to create components, which can be used by different research groups through well-defined interfaces. As a result of this synergy, rapid development is possible while avoiding redundant efforts.

The remainder of this thesis is structured as follows: chapter 2 discusses the three computational packages, NWChem, GAMESS and MPQC. Also included in chapter 2 is a brief introduction of Quantum Mechanics and Molecular Mechanics (QM/MM) and CCA – one of the pillars of this research work. Chapter 3 describes the design and implementation details of the QM/MM component model. Also the potential use of interoperability and reusability are discussed in this chapter. Chapter 4 gives a review of performance analysis of the QM/MM component model using TAU [9]. Finally, chapter 5 states a summary of the proposed work and current research issues.

# CHAPTER 2. BACKGROUND

In this chapter, a broad overview of the quantum computational packages involved in this project is provided. Then I will briefly describe the important concepts required for understanding the work done as part of this research.

## 2.1 Overview of Computational Packages

Three of the world's most significant and widely used computational chemistry software codes developed under the US Department of Energy are: NWChem, MPQC and GAMESS. To provide interoperability between these three packages is the ultimate research computer science goal. The focus is to collaborate with another package to provide needed functionality instead of creating it "in-house". The collaboration among these packages is possible by bringing the best practices of Software Engineering into the field of quantum chemistry.

Pacific Northwest NATIONAL LABORATORY
NWChem
high performance computational chemistry software
ISU
GAMESS
DEPARTMENT OF ENERGY UNITED STATES OF AMERICA
Interoperability???
MPQC
Sandia National Laboratories

**Figure 1. US Department of Energy's three computational quantum packages**

As part of my thesis work, I will majorly focus on using the NWChem scientific package to create the underlying components for a very important functionality called combined Quantum Mechanics and Molecular Mechanics (QM/MM). It is important to study the overall architecture of NWChem before actual creation of components. It is equally important to know about the communication mechanics used in other packages involved in the collaboration since the goal is to provide interoperability between all three packages.

### 2.1.1 NWChem

**NWChem (NorthWest Chemistry)** developed and supported by the Pacific Northwest National Laboratory (PNNL) is designed to run on high-performance parallel supercomputers as well as standard HPC clusters. The application is built on object-oriented principles and implemented using Fortran, C and Python. The code is used to compute the properties of molecular and periodic systems using standard quantum mechanical descriptions. The code also allows molecular dynamics and free energy simulations, Gaussian Density Functional Theory (DFT), planewave based DFT and many body perturbation theory. Classical and quantum approaches are used together to perform hybrid quantum mechanics and molecular mechanics simulations.

NWChem is Non-Uniform Memory Access (NUMA) aware in order to scale to massively parallel computer architectures in all dimensions including CPU, memory and disk. Memory access is possible by using the message-passing capability provided by the portable Aggregate Remote Memory Copy Interface (ARMCI) [10]. ARMCI is a standalone system that is used to support user-level libraries and applications in a message-passing model. NWChem has a five-tiered modular architecture as shown in Figure 2. The five tiers are as follows:

1. **Generic task interface:**

   This is an interface between the user and the chemistry modules where NWChem processes the input, sets up the parallel environment and performs initializations needed for the desired calculations. This interface also determines the task to be accomplished and transfers control to the different modules in the second tier.

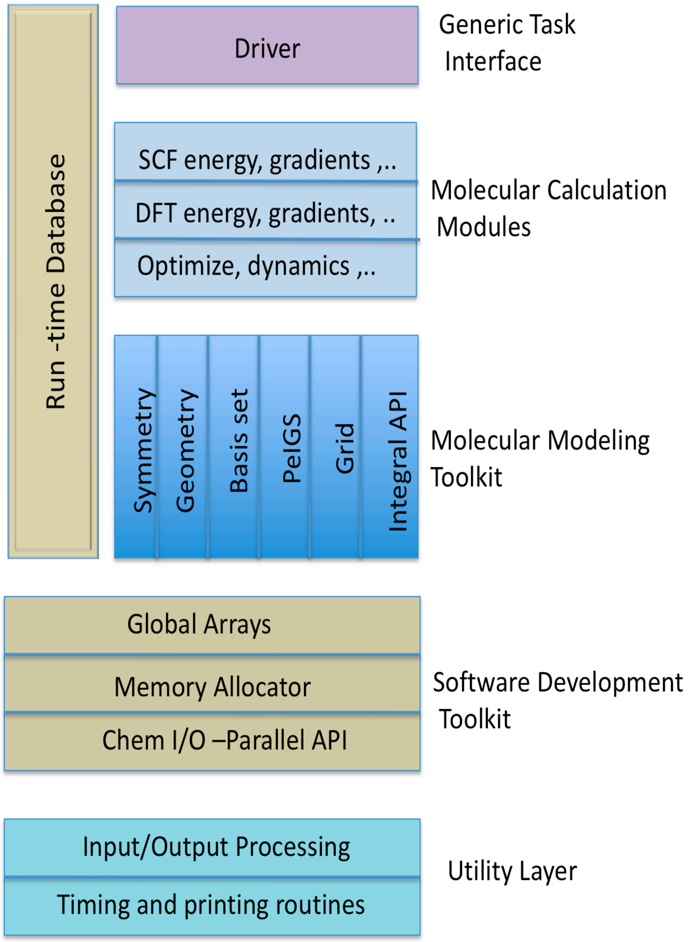


**Figure 2. Five-tiered NWChem Architecture**

2. **Molecular calculation modules:**

   These high-level programming modules accomplish computational tasks based on the input theories specified by the user and uses toolkits and routines that reside in the lower levels of the architecture and are theory specific. Various modules at this level communicate with each other via an information repository called the run time database or through well-defined disk files.

3. **Molecular modeling toolkit:**

   This tier contains tools that provide basic chemical functionality such as symmetry, basis sets, grids, geometry and integrals that are used by many of the molecular calculation modules.

4. **Software development toolkit:**

   This level forms the foundation of the architecture and allows development of object-oriented code mainly in Fortran 77. This tier consists of five tools and each of these tools provides an interface between the chemistry specific part of the program and the hardware.

   - *Run Time DataBase (RTDB) :*

     The RTDB is an information repository, which is used by various modules to share data.

   - *Memory Allocator (MA):*

     The MA tool allows the programmer to allocate memory that is local to the calling process - data not being shared with other processes. MA provides both heap as well as stack memory management disciplines. Since Fortran 77 doesn't support dynamic memory allocation, MA plays an important role in the NWChem code.

- *Global Arrays (GA) :*

  GA [11, 12, 13] toolkit provides an efficient and portable "shared memory" programming interface for distributed memory computers. Using this toolkit, a programmer can take advantages of both shared memory and message passing paradigms in the same program.

- *ChemIO*

  ChemIO is a high performance parallel I/O API and is used to create files that are either local to the process, distributed among the file systems or on shared disk spaces. This allows the developer to perform parallel I/O in the most efficient way for a particular algorithm or for a particular hardware.

5. **Utility routines:**

   This tier provides various functionalities such as input processing, output processing, timing and printing that most of the higher tiers require.

### 2.1.2 MPQC

**MPQC (Massively parallel Quantum Chemistry)** developed and maintained at Sandia National Laboratory uses an object oriented design paradigm and is implemented using C++ (with a few C functions). The MPQC package provides parallel implementation of Hartree-Fock, Density Functional, second order many body perturbation theory, MP2 and optimization methods. MPQC uses MPI as the message-passing layer to support parallel execution and is able to run on a wide range of architectures such as individual workstations, symmetric multiprocessors and massively parallel computers.

### 2.1.3 GAMESS

**GAMESS (General Atomic and Molecular Electronic Structure System)** developed and maintained at Iowa State University, is mainly implemented using Fortran 77. It performs various calculations including Hartree-Fock, Density Functional Theory, many body perturbation theory (MP2 and different coupled cluster methods), Generalized Valence Bond, and Multi-configurational self-consistent field. GAMESS uses the Data Distributed Interface (DDI) to obtain high parallel efficiency, which relies either on TCP/IP sockets or MPI.

All these packages have their strengths and weaknesses. The goal of this research is to provide an easy way to utilize the best of all the packages without imposing significant performance penalty

## 2.2 Quantum Mechanics and Molecular Mechanics

The rapid increase in computer speed and hardware technology advancement has made quantum chemistry a practical tool for chemists in different branches of chemistry, such as organic, inorganic, analytical, and physical. However, while Quantum Mechanical (QM) methods can treat chemical reactions accurately since they treat the atoms and the electrons using ab initio electronic structure, they are quite expensive for very large molecules. On the other hand, Molecular Mechanics (MM) methods can treat very large molecules since they are based on ignoring electrons and treating the atoms classically but are not well suited for chemical reactions. To deal with the chemical reactions in very large systems, a combined QM/MM method is used where a quantum mechanics calculation is embedded in a molecular mechanics model. In this, quantum mechanics can be used to treat the part of the system affected by the reaction, and molecular mechanics to treat the rest of the chemical environment [14]. For example, for a reaction in solution, one treats the reacting solute molecules and the first solvent shell using QM and the surrounding solvent molecules using MM.

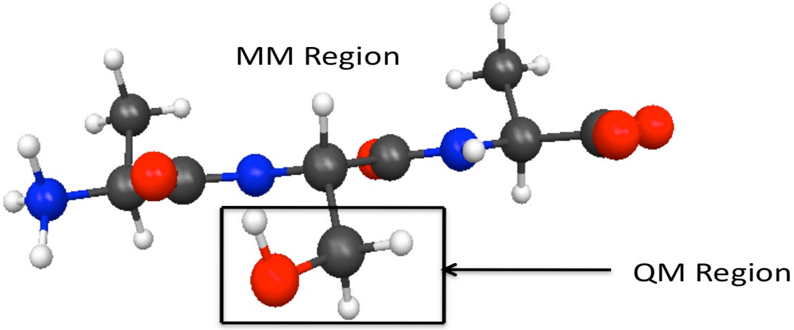


**Figure 3. Regions involved in QM/MM calculation of tripeptide alanine-serine-alanine**

The challenge is to find an appropriate boundary region between the QM and MM regions and to accurately describe the physics of the boundary region. This is critical because a selection of an inappropriate boundary will produce poor end results. There are many choices for the physics of the boundary conditions.  This work will focus on two models.

The first is a model where molecules are weakly interacting with one another (i.e. not bonded to one another) and only the charges or electrostatics of the MM region need to be taken into account in the QM calculation. In the second model, the border contains atoms that are bonded to one another. Figure 3 shows both QM and MM regions of the tripeptide alanine-serine-alanine. In this model, the quantum region will need to substitute the rest of the MM region by hydrogen or fluorine like caps as well as the electrostatic charges. In both models, the electrostatic interactions are included in the QM computation so that the QM wave function feels the effects of the environment of the MM atoms allowing the reaction to be modified based on the environment. Overall, the QM/MM approach combines the strength of both QM (accuracy) and MM (speed) packages.

## 2.3 Common Component Architecture

The Common Component Architecture specification helps in managing software complexity by bringing the benefits of a lightweight component system to high performance science. CCA is built on object-oriented principles such as encapsulation, abstraction, data hiding, interfaces and modularization and adds the flexibility of building applications in a plug-and-play fashion.

CCA enables communication between software packages that are written in different languages by using a tool called BABEL [15]. In order to support this multi-language interoperability, BABEL relies on the specifications of interfaces in Scientific Interface Definition Language (SIDL). A CCA component is a SIDL class that implements the gov.cca.Component interface and other user-defined interfaces. CCA employs the notions of ports that define the interaction between different components. CCA supports two types of ports: provide and use, to specify functionality provided by a component and to access functionality of other components (that provide the matching port type). Additionally, *Parameter Ports* are designed as provides ports for components that need run time settings for their computations.

A CCA framework is responsible for component management including dynamic loading, component instantiation, connection and disconnection of the matching ports. A range of CCA frameworks are available such as XCAT-C++ [16], XCAT-Java [17], DCA [18], DECAFE [19], CCAIN [20], Ccafffiene [21]. In the proposed work, I have used the Ccaffiene framework [13] that supports both Single Program Multiple Data (SPMD) and Multiple Program Multiple Data (MPMD) parallel models. It is possible to use components that rely on different communication systems under the same framework. This is a very useful feature that will be harnessed to provide interoperability between three quantum chemistry packages of this work: NWChem, MPQC and GAMESS.

In chapter 3, an overall description of how the CCA framework is used to create components from a legacy package is given. Also, I will focus on the crux of componentization approach i.e. providing interoperability between incompatible packages.

## 2.4 Literature Review

CBSE emerged from the failure of object-oriented development to support reuse. In object-oriented development, reuse is supported in the form of inheritance. Inheritance is a poor form of reuse because it requires familiarity with the internal details of the base class implementation. Component-based development allows reuse of existing components without knowing its internal details as long as the component complies with the set of interfaces [22]. To support CBSE, various component architectures such as COM, CORBA, and EJB came into existence and employed by millions of users. Unfortunately, these technologies focus exclusively on local and distributed computing and do not support the concept of parallelism. These technologies are often platform or language-specific. These issues have limited adoption in the HPC community and this led to the formation of the CCA forum in 1998. Collaboration among chemists and computational scientists to solve computational chemistry problems became possible using CCA. CCA project encouraged collaboration among three major research groups, NWChem, GAMESS and MPQC under USDOE.

In 2004, components were developed for MPQC and NWChem to perform geometry optimizations by integrating Toolkit for Advanced Optimization (TAO) [23]. TAO provides optimization software for the solution of scientific applications on high performance architectures. Efficient backtracking line search algorithm was implemented in TAO and was made available to quantum packages through the adoption of the CCA. This laid the groundwork for further integration between multiple codes. This optimization work encouraged GAMESS to have the same high level interfaces.

In 2005, for the very first time CCA along with GA tool were deployed to exploit variable concurrency in the field of computational chemistry in the form of multi-level parallelism on a high performance computer [24]. This was found to give substantial improvements in parallel scalability. Results showed that the numerical hessian calculation using three levels of parallelism outperformed the original NWChem code based on single level parallelism by a factor of 90% on 256 processors.

In 2006, low-level components were developed that enable the one and two electron integrals computed by MPQC package to be used by another package. MPQC implements the two-electron integrals needed for explicitly correlated methods which NWChem and GAMESS lack [25].

Both GAMESS and NWChem parallelize the integral computations instead of parallelizing the integral computation itself. In 2007, GAMESS components were developed with load balancing mechanism that allows the integrals to be distributed among processes evenly [26, 27].

These packages usually provide more than one method for a computation to be used under different circumstances. For Computational Quality of Service (CQoS), a performance database is created which stores the computational time, cache utilization, communication latency, etc. Chemists can use this data to select the component that meets his/her performance needs [28].

A remarkable new development is that these projects are transitioning to employ BOCCA [29], a comprehensive suite of CCA tools that provide project management and an environment for creating and managing applications made of CCA components.

# CHAPTER 3. COMPONENTIZING NWCHEM

In this chapter, I will describe the overall process of componentizing a legacy package starting with designing interfaces for developing components to form a complete interoperable application. At the end of this chapter, I will discuss reusability and interoperability of these components in detail.

## 3.1 QM/MM component design

In this section, I will describe the process of wrapping legacy NWChem code as CCA components, i.e. componentizing NWChem. The focus will be on componentizing the QM/MM code without modifying the legacy code, but the general technique can be applied to produce components for other legacy codes. Componentizing QM/MM (or for that matter any other component model) is an iterative and over-lapping four-step process:

1. Deciding the flow and the information passed between different pieces of the QM/MM code.
2. Designing generic interfaces using SIDL specifications.
3. Implementing interfaces in the form of classes or components and writing wrappers to the native library functions.
4. Writing a driver component that uses all of the other components and classes in the repository to accomplish the calculation of interest, providing and using ports inside the components so as to enable the communication between components.

These steps will be discussed in further details in the following sections.

### 3.1.1 Information flow in the legacy QM/MM code

The QM/MM code in NWChem is written using a combination of Fortran and C languages. In this section, the details of the communication between different modules and control of the QM/MM execution are laid out. The flow of control proceeds in the following steps (with more details for each step provided below):

1. Initialize the parallel environment
2. Open the input NWChem file and scan it for memory directives
3. Complete the parallel and memory initialization process.
4. Process start-up directives
5. Summarize start-up directives and write to an output file
6. Open the runtime database with the appropriate mode.
7. Set the quantum related information such as basis set and theory type in the rtdb.
8. Process and execute the prepare block. This step basically reads a PDB [30] file and generates topology and reset files specifying the quantum piece, if any, in the topology file.
9. Process and execute the MD block, which mainly sets up molecular mechanics parameters and calculates classical energy.
10. Process and execute QM/MM block, which is mainly responsible for defining the interface parameters between the QM and MM regions and for calculation of QM/MM energy of a system.

11. Finally closing the database, cleaning up GA and killing the parallel processes terminate the execution process.

In the first step, the parallel environment is set up by calling the ARMCI wrapper routine pbeginf(). This creates the parallel processes and provides basic message passing capabilities. In the second step, the input file is opened by process 0 to obtain user specified memory parameters, which are passed on to all other nodes. In the third step, the GA library and local memory allocator are initialized. At the end of this step, the parallel environment is fully initialized.

In the fourth step, the startup directories given in the input file are scanned and processed. In the fifth step, this information is summarized to an output file. In the sixth step, process 0 opens the runtime database with the appropriate mode (empty for startup, old for restart or continue). At the end of this step, all the startup directives have been processed.

After this, each input line is processed, and data is inserted into the rtdb for later usage. Only process zero is involved in the input module: reading input or putting data into the rtdb. For this purpose, the database is switched into sequential mode at the beginning of the input module, and back to parallel at the end.

Once a task directive is processed and entered into the rtdb, control is returned to the main program so that the task can be carried out. The main program initiates the execution of the task by calling the routine task(). On successful completion of the task, the main program invokes the input module once again.

Parameters defined for the QM region in the input file such as basis and theory, which are input into the database in the seventh step and used further in the QM/MM calculation.

In a QM/MM calculation, mainly three blocks (and corresponding three task directives) are encountered in the input file, which are covered in the steps 8-10. The prepare module, step 8 specifies the location of the input Protein Data Bank (PDB) file (described in detail in Appendix C), the force field to be used (AMBER95 or CHARMM22) and the location of the force field database files. The quantum region in the QM/MM calculation is specified by the 'modify atom' directive in the prepare block. Prepare is mainly responsible for generation of three files:

- **Sequence (.seq)**

  This file is generated after analyzing the supplied coordinates from a PDB-formatted file. It contains the description of the system in terms of basic building blocks found as fragment (.frg) or segment (.sgm) in the database directories for the force field used. A fragment contains the list of atoms with their force field dependent atom types, partial atomic charges calculated from a Hartree Fock calculation for the fragment, followed by a restrained electrostatic potential fit, and a connectivity list. Based on the information in this fragment file, the lists of all bonded interactions are generated, and the complete lists are written to a segment file.
- **Topology (.top)**

Based on the generated sequence file and the force field specific segment database files, the compiled lists of atoms, bonded interactions, excluded pairs, and substituted force field parameters are stored in the topology file. The quantum region information in the QM/MM calculation as specified by the ‘modify atom’ directive is also stored in the topology file.

- **Restart (.rst)**

  Based on the coordinates specified in the PDB file and the topology file for the chemical system, a restart file is generated for the system with coordinates, velocities and other dynamic information. It may also include solvation of the chemical system and periodic boundary conditions. The restart and topology files contain information about the classical force field as well as the coordinates of the QM and MM regions. This information is used in a MM simulation as well as in a QM/MM simulation.

In the ninth step, the MD block is executed. The molecular mechanics parameters are specified in the MD input block, which is required for the QM/MM simulations. It specifies the restart and topology file (points to the restart and topology files created in the prepare step) that will be used in the calculation. It also contains information relevant to the calculation of the classical region (e.g. cutoff distances, constraints, optimization and dynamics parameters) in the system. It is possible to set fixed atom constraints on both classical and quantum atoms inside this MD block.

In the tenth step, the interaction parameters between the QM and MM regions are specified inside the QM/MM block. The following are a few directives that can be specified in the QM/MM block:

- **eatoms**: This specifies the zero of energy for the QM module in QM/MM calculation. This is necessary because quantum calculations include atomic energies that are not included in force fields. The zero of energy for the MM system is the separated atom energy. The zero of energy for QM systems is the vacuum. This imbalance can be rectified by well-defined eatoms value.
- **cutoff** : This directive defines the radius of the zone around the quantum region where classical atoms will be allowed to interact with the quantum region.
- **mm_charges**: This directive controls the treatment of classical point (MM) charges (Bq charges) that are interacting with QM region.

Thus the overall QM/MM calculation can be divided into three major parts

- specification of the molecular mechanics parameters for the classical region
- specification of the quantum mechanical method for the quantum region
- specification of the interaction between quantum and classical methods

The third specification is the most critical part in the QM/MM calculation. Interfaces and flow of data should be well defined so as to allow interaction between the QM and MM regions.

In the final step, step 11, the execution is terminated by closing the database, cleaning GA and gracefully killing parallel processes.

### 3.1.2 Defining generic interfaces

The second step in componentizing is writing interfaces in accordance with the flow and pieces involved in a QM/MM calculation. The interface definitions are written in the cca-chem-generic package. Defining interfaces in this package will allow each chemistry package (e.g. NWChem, MPQC and GAMESS) to implement the interfaces and create chemistry components and classes. Interfaces will be implemented either by a class or component. The agreed upon standard in writing interfaces for the chemistry community is that factories will be used to provide a general component capability. Thus the naming convention is that the interface for a component ends with 'FactoryInterface' whereas for a class, it ends with 'Interface', both having a common prefix in their names. The key distinction between a class and a component is that a class cannot provide or use ports whereas a component can. So keeping in mind which functionalities are to be exposed to components and which functionalities are needed only inside components, the following interfaces have been written:

1. **Chemistry.MoleculeInterface and Chemistry.MoleculeFactoryInterface**

   The Chemistry.MoleculeInterface declares a variety of functions related to a molecule in the chemical system, such as setting/getting Cartesian coordinates, atoms and units. As discussed earlier, the QM molecule now needs to know about the external charges from the MM region and so functions are declared for setting/getting the point charge values and coordinates. The Chemistry.MoleculeFactoryInterface declares functions to instantiate and get a copy of a Molecule class.

2. **Chemistry.ModelInterface**

   Chemistry.ModelInterface is the basic interface that can be extended by quantum, classical as well as combined QM/MM ModelInterfaces. It provides functionalities such as calculation of molecular energy, gradient and Hessian.

3. **Chemistry.QC.ModelInterface and Chemistry.QC.ModelFactoryInterface**

   These interfaces are meant for pure quantum calculation as well for quantum zone calculations in QM/MM simulation. Chemistry.QC.ModelInterface extends the Chemistry.ModelInterface and provides additional functionalities related to setting up of a model along with evaluation of quantum energy, gradient and Hessian of a molecule. The Chemistry.QC.ModelFactoryInterface is used to instantiate and get a copy of a QC Model class. It is also used for setting up the quantum information such as theory and basis type.

4. **Chemistry.MM.ModelInterface and Chemistry.MM.ModelFactoryInterface**

   These interfaces are meant for pure molecular mechanics simulation as well as for MM zone calculations in combined QM/MM calculation. The Chemistry.MM.ModelInterface extends the Chemistry.ModelInterface and provides additional functionalities for setting up of the MM zone including creation of restart and topology files and defining the QM zone in QM/MM calculations, along with evaluation of molecular mechanics energy, gradient and hessian of a molecule. The

Chemistry.MM.ModelFactoryInterface is used to instantiate and get a copy of a MM Model class.

5. **Chemistry.QMMM.ModelInterface and Chemistry.QMMM.ModelFactoryInterface**

   These interfaces are used to provide communication between the QM and MM regions in combined QM/MM calculations. Chemistry.QMMM.ModelInterface extends the Chemistry.ModelInterface and provides additional functionalities for setting up the interaction between quantum and classical regions in QM/MM calculations along with evaluation of QM/MM energy, gradient and Hessian of a system. The Chemistry.QMMM.ModelFactoryInterface is used to instantiate and get a copy of a QMMM Model class. It also sets up references of QM and MM model instances inside a QMMM model.

6. **Database.ModelInterface and Database.ModelFactoryInterface**

   Database interfaces are used to provide a storage place for quantum zone information during QM and MM zone interactions. It also stores the QM and the MM energies. Database.ModelInterface defines functionalities related to database like connecting and disconnecting a database, opening and closing a database, insertion and retrieval of quantum coordinates, point charges, etc of a molecule. The Database.ModelFactory declares functions to instantiate and get a copy of Database Model class.

The first two steps are critical in the design phase. Writing interfaces involves a lot of assessment since the goal is to make them as generic as possible so that all packages are able to implement the same set of interfaces irrespective of their unique implementation details. This entails not only deciding on the important components, but it also means determining the important data (and the data layout) that must be available to the different components. Information is passed between different components and classes to achieve the correct flow for the chemistry calculations. For example, the molecule should be set before its energy is calculated, which means the molecule object must be passed to the energy calculation model. This is also the point where classes and components are defined. The key distinguishing feature between class and component is that a class cannot provide or use ports whereas a component can. So the developer needs to decide which functionalities are to be exposed to components and which functionalities are needed only inside components. In this work, I have designed components that interact with

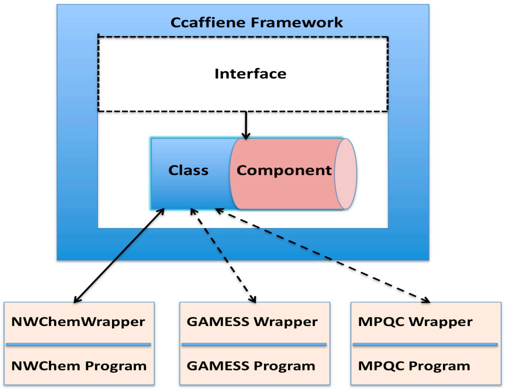


**Figure 4. Ccaffiene framework and the wrappers of the underlying computational packages**

classes and other components (sometimes with wrappers functions) whereas classes interact only with wrapper functions (Figure 4).

### 3.1.3 Implementation of interfaces

In this step, wrapper functions for the components/classes are developed. In the NWChem-CCA project, a wrapper function acts a bridge between the CCA components and the underlying computational package. This is not a strict requirement if the components are written in the native language but they are necessary if the components are written in different language or if a more generic interface to the package is required for non-CCA purposes. In this case, wrappers act as a middle layer for the function calls in the component language and native NWChem language. Wrappers are useful in making the CCA interfaces simple and in writing high-level routines in the native code that can be used for other purposes. For QM/MM calculations, wrapper functions are written in accordance with the flow described in the "Information flow in QM/MM" section. In accordance with the flow discussed in section 3.1.1, I wrote the corresponding wrappers and the one-to-one mapping for the same is shown in Figure 5. Wrapper functions provides the following advantages:

- It is possible to write a stub that can call wrapper functions to test the implementation without involving CCA components.
- When functionalities are added/edited in the native code, we can update the wrappers accordingly without modifying the CCA components.

- Code looks cleaner and follows the modularity principle. Code becomes easier to manage with the introduction of a wrapper layer because we don't embed significant amounts of NWChem code in the CCA implementation files.

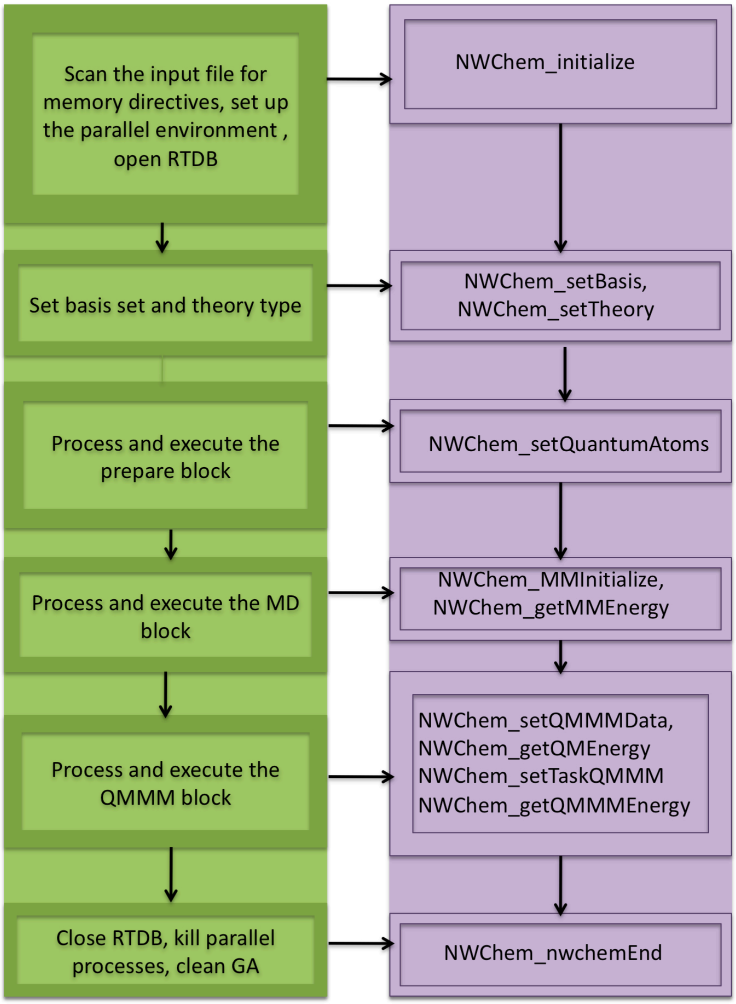

Scan the input file for memory directives, set up the parallel environment , open RTDB
Set basis set and theory type
Process and execute the prepare block
Process and execute the MD block
Process and execute the QMMM block
Close RTDB, kill parallel processes, clean GA
NWChem_initialize
NWChem_setBasis, NWChem_setTheory
NWChem_setQuantumAtoms
NWChem_MMInitialize, NWChem_getMMEnergy
NWChem_setQMMMData, NWChem_getQMEnergy NWChem_setTaskQMMM NWChem_getQMMMEnergy
NWChem_nwchemEnd

**Figure 5. Mapping from legacy QM/MM code to underlying wrappers**

Once the interfaces and wrapper functions are written, the next step is to generate the glue code through Babel in the form of implementation files. The following are the classes/components involved in QM/MM calculation:

1. **Chemistry.Molecule and Chemistry.MoleculeFactory**

   Chemistry.Molecule is a class that implements the Chemistry.MoleculeInterface and Chemistry.MoleculeFactory is a component that implements the Chemistry.MoleculeFactoryInterface and is able to return an instance of a Molecule class. Both are implemented in the cca-chem-generic package.

2. **NWChem.QM.Model and NWChem.QM.ModelFactory**

   NWChem.QM.Model is a class that implements the Chemistry.QC.ModelInterface and NWChem.QM.ModelFactory is a component that implements the Chemistry.QC.ModelFactoryInterface and is able to return an instance of NWChem.QM.Model class. NWChem.QM.Model class interacts with the wrapper functions to set up the theory and basis set, and calculates quantum energy, gradient, hessian of a molecule.

3. **NWChem.MM.Model and NWChem.MM.ModelFactory**

   NWChem.MM.Model is a class that implements Chemistry.MM.ModelInterface and NWChem.MM.ModelFactory is a component that implements the Chemistry.MM.ModelFactoryInterface and is able to return an instance of the NWChem.MM.Model class. NWChem.MM.Model class interacts with the wrapper functions to set up the molecular mechanics parameters and generate topology and restart files, extract quantum information and store it in a database component (described later) if it is a QM/MM calculation, and finally calculate classical energy, gradient, Hessian of a molecule.

### 4. NWChem.QMMM.Model and NWChem.QMMM.ModelFactory

NWChem.QMMM.Model is a class that implements the Chemistry.QMMM.ModelInterface and NWChem.QMMM.ModelFactory is a component that implements the Chemistry.QMMM.ModelFactoryInterface and is able to return an instance of the NWChem.QMMM.Model class. NWChem.QMMM.Model class interacts with the wrapper functions to set up the interaction parameters between quantum and classical zones in a QM/MM calculation and calculate the combined QM/MM energy of a system.

### 5. NWChem.Database.Model and NWChem.Database.ModelFactory

NWChem.Database.Model is a class that implements the Database.ModelInterface and Database.ModelFactory is a component that implements the NWChem.Database.ModelFactoryInterface and is able to return an instance of Database.Model class. Database.Model class is used to set up a database and allows retrieval and storage of quantum information passed between the QM and MM regions.

Note that a component is also a class that must implement the gov.cca.Component interface and also gov.cca.Port interface. In addition, if components wish to take inputs from the user via parameter ports then it should additionally implement the gov.cca.ports.BasicParameterPort interface.

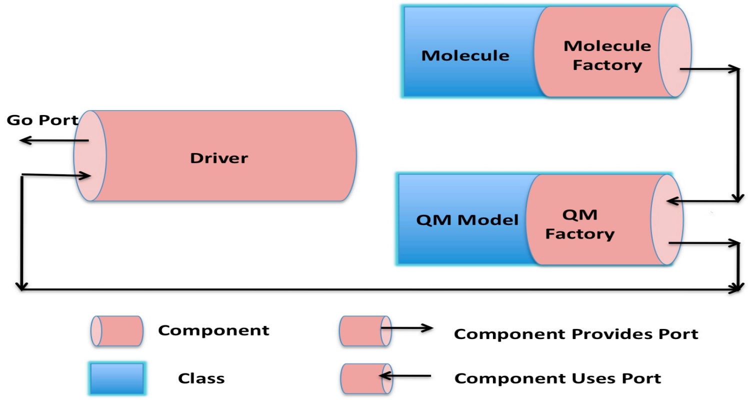


**Figure 6. Quantum Chemistry component model**

For the QM/MM model, we initially need a QM model, which should be able to calculate a molecular energy based on the molecule definition, theory level, and basis set provided by the user. The design of the QM model has been discussed in [10] and is shown in Figure 6. This was implemented using the same process as discussed above. However, the need to calculate the point charges from the environment was something that the original design did not take into account. Therefore, this has been added to the previous implementation. The interfaces are designed in such a way that this model can be used independently to calculate a purely quantum energy as well as in combination with MM to calculate the energy of the quantum region. A generic interface is the key to reusability.

The next step is to have a MM component model, which will calculate the MM energy based on the inputs given by the user such as the force field, cut-off distances and other constraints, related to the classical region. The input for the molecule is read in through a PDB file in this implementation, the MM input also identifies the MM and QM molecular zones. This is quite common in QM/MM implementations since the MM region is usually quite large compared to the QM region. To enable the interaction between the MM and QM regions, I introduce another component called the *Database* to store and retrieve the intermediate quantum zone information and energies. We also introduce a QM/MM *Model* and *Model Factory* to handle the flow and interaction between the QM and MM regions. The QM component gets the quantum information from the database component inserted previously by the MM component (after identifying the quantum zone in QM/MM).

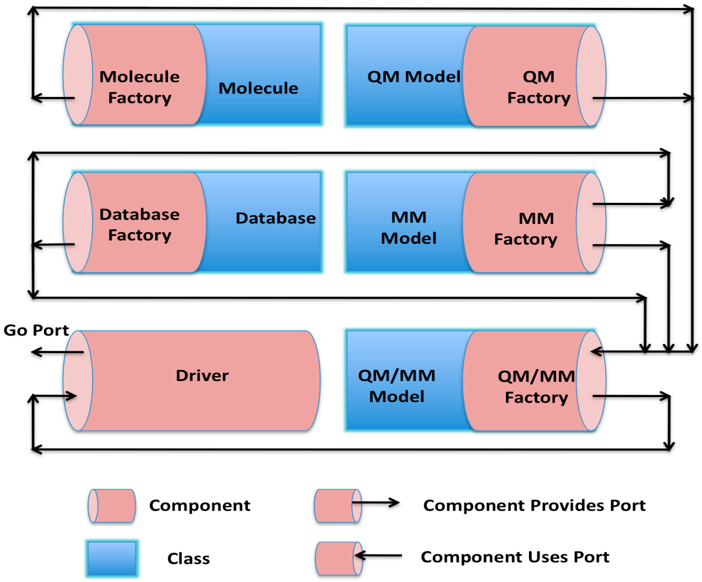


**Figure 7. QM/MM component model**

I have meticulously developed a thin interface, which is required between the QM and MM regions in such a way that the moment the relevant information about the quantum zone is set by the MM component, the QM component uses the information to contribute its part towards the QM/MM energy. Defining the thin interface was a challenging task because final energy calculations are mostly dependent on the interface between QM and MM regions. The design layout is shown in Figure 7.

### 3.1.4 Implementation of Driver and provision/usage of ports

Driver component is different from other components as it provides a starting point in the entire application run (similar to main() in C programming) by means of a *GoPort*. The driver component binds the other components in the framework and drives the entire calculation. The driver component also implements gov.cca.Component and gov.cca.ports.GoPort interfaces.

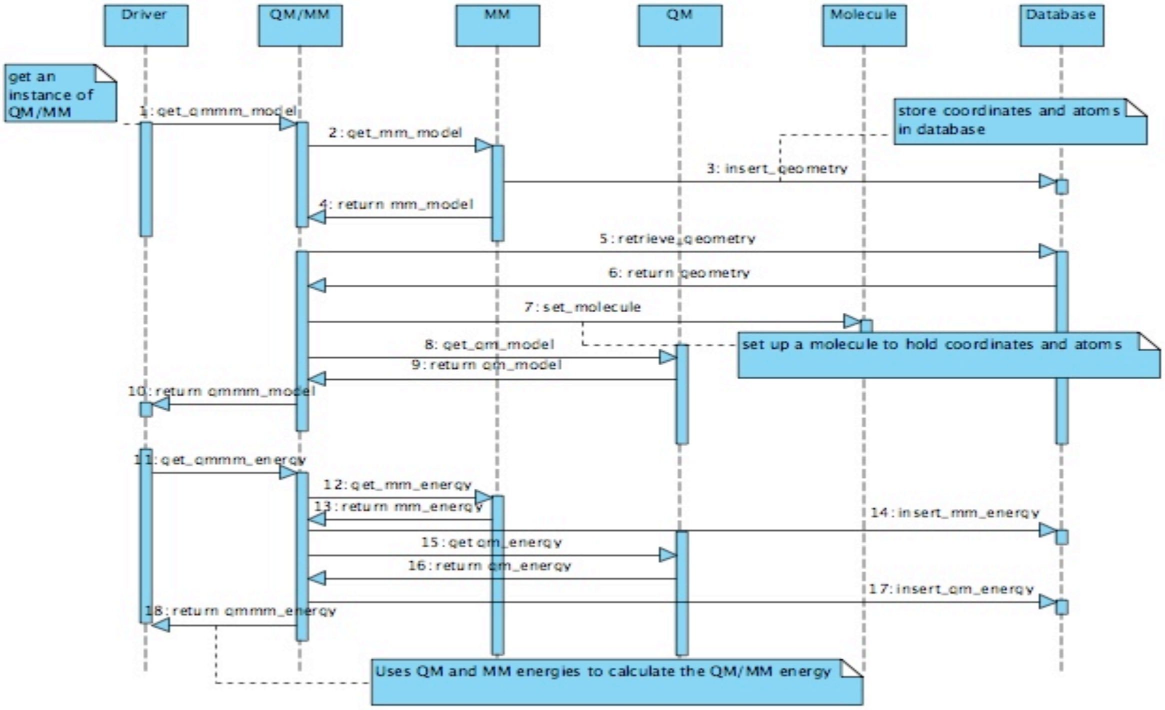


**Figure 8. UML sequence diagram for QM/MM calculation depicting the interactions between different components over time.**

A QMMMDriver component is written that orchestrates the different components defined previously to perform combined QM/MM calculation. It gets the reference of the QMMM Model class (through QMMM ModelFactory) and then it uses this reference to calculate the QMMM energy.

The sequencing diagram in Figure 8 depicts the sequence of data flow between different components and the following connections can be observed:

1. The Driver component communicates with the QM/MM component, i.e. driver uses functionalities of QM/MM Model and ModelFactory.
2. The QM/MM component communicates with the Molecule, QM, MM and Database components i.e. i.e. driver uses functionalities of Molecule (to set the molecule information such as coordinates and point charges), QM (to get the quantum energy), MM (to get the classical energy and obtain quantum information), and Database (for retrieval of intermediate quantum information) Models and ModelFactories.
3. The MM component communicates with the database component to store intermediate quantum information extracted from the topology file.

At this point, I know which components interact with each other. Accordingly, I made connections between them via CCA ports as shown in Table 1. As is true for much of software development, it is often true that the design needs to be revisited as the implementation is taking place since complicating factors often only surface during the implementation. So the design and the implementation often evolve together, as was the case for the QM/MM design.

| **Component** | ***uses* port** | ***provides* port** |
|---|---|---|
| Driver | QMMMFactoryInterface | Go |
| Molecule | - | MoleculeFactoryInterface |
| QM | - | QMFactoryInterface |
| MM | DatabaseFactoryInterface | MMFactoryInterface |
| QMMM | MoleculeFactoryInterface,<br>QMFactoryInterface,<br>MMFactoryInterface,<br>DatabaseFactoryInterface, | QMMMFactoryInterface |
| Database | - | DatabaseFactoryInterface |

**Table 1. CCA compliant components involved in QM/MM calculation and the corresponding ports they provide and/or use.**

## 3.2 Reusability and Interoperability

Interoperability and reusability call for high-quality interfaces and components, classifications and retrieval mechanisms, sufficient and proper documentation of components, a flexible means of combining components and a means of adapting components to specific needs. In this section, I will formally about how the components developed are reusable and then I will show how NWChem components are able to interoperate with other quantum packages.

### 3.2.1 Reusability

Software design is a challenging task due to the numerous complexities involved in the process. These days, this complexity is increasing to levels in which reuse of previous software designs are needed to reduce the development time and efforts. Reusability is the process of adapting a generalized component to various contexts of use. The main intention of this research is to encourage reusability and I have focused on three flavors of reusability in this work [31]:

1. **Black-Box Reuse:**

   Using a component as a black box means using it without seeing, knowing or modifying any of its internals. In this type of reuse, one can see the SIDL interfaces and not the implementation of the component. The interface contains the function definitions and documentation explaining the requirements and restrictions of the interface. The previous implementation of the interface can be changed without any effects on new implementations. In the proposed QM/MM component Model, the 'Chemistry.QC.ModelInterface' is implemented by all three quantum computational packages, NWChem, GAMESS and MPQC without seeing each other's implementations.

2. **Glass-Box Reuse: -**

   Using a component as a glass box is similar to that of a black box with the addition that the internals can be seen from the outside. This type of reuse gives information about the working of a component but it doesn't provide ability to change the internals of the component. In QM/MM component model, I have used the previously implemented quantum components and developed few more components to provide the complete QM/MM CCA application. Also, the new components developed in the process can be reused or plug and played by MPQC and GAMESS (discussed in detail in the next section).

3. **White-Box Reuse:**

   Using a component as a white box means it is possible to see and change the inside of the box (component) as well as the interface. The new component can choose to retain or change its previous functionality. During the process of creation of QM/MM component model, I reused the existing component 'Chemistry.MoleculeFactory' and added the functionality of creation of molecules via user specified input parameters in the interface and implemented the functionality in the component (missing in the previous implementations).

The proposed componentization approach is flexible enough to support all the above types of reusability.

### 3.2.2 Interoperability

The term component interoperability can be defined as the ability of components to provide services and to accept services from other components and to use the services exchanged to enable an effective operation together. The ultimate goal of this research is to provide interoperability between components written in different languages to be connected together.

Section 3.1 described the components developed for the NWChem Package. The design was then tested using the QM and MM component models developed within the underlying NWChem package. It should be ideally possible for other packages having quantum chemistry components to use the NWChem-based molecular mechanics and combined QMMM components for calculation of QM/MM energies. The next obvious test was to check if it is possible to use QM/MM component model with other quantum packages like MPQC.  Theoretically speaking, end users shouldn't need to understand the underlying design and communication patterns. This is a challenging task if the components are developed from a legacy code. The proposed model achieves this interoperability by means of well-defined generic interfaces and high-quality component implementations. As shown in Figure 9, we need to load the QM components of the respective packages and other component in the QM/MM model of the NWChem package in the Ccaffiene framework. Proposed model flawlessly calculates the QM/MM energy by hiding the internal details from the end users and overcoming language barriers, input formats and underlying message-passing libraries used by each package.

Following are the steps to show interoperability between MPQC and NWChem Components in the Ccaffiene framework:

1> Instantiate all the components: NWChem.QMMMDriver and NWChem.QMMM.ModelFactory, NWChem.MM.ModelFactory, NWChem.DatabaseFactory and Chemistry.MoleculeFactory MPQC.QM.ModelFactory in the framework.

2> QMMM.ModelFactory asks for an instance of NWChem.MM.Model class through NWChem.MM.ModelFactory.

3> NWChem.MM.ModelFactory reads the PDB file and molecular mechanics input parameters, initializes NWChem, generates topology and restart files, gets an instance of the NWChem.DatabaseModel and store quantum information in the database.

4> NWChem.QMMM.ModelFactory then retrieves the quantum zone information from NWChem.Database Model class and sets up a molecule via Chemistry.MoleculeFactory and gets an instance of the MPQC.Model class via MPQC.ModelFactory.

5> MPQC.ModelFactory reads the quantum input parameters and associates Chemistry.Molecule class instance with MPQC.Model class instance.

6> NWChem.QMMM.ModelFactory then reads the QM/MM interface parameters and

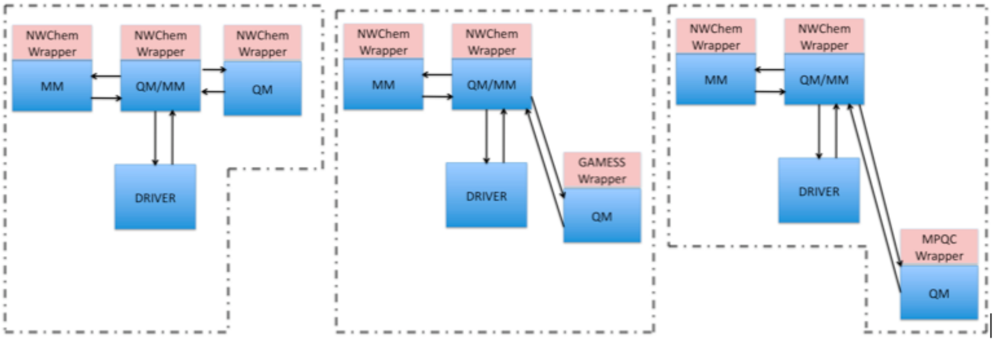


**Figure 9. Usage of QM/MM component model by three different packages. The first is all with NWChem, the second uses NWChem for the MM and QM/MM Factories and GAMESS for the QM Factory and the third uses NWChem for the MM and QM/MM Factories and MPQC for the QM Factory**

asks for quantum and classical energies using MPQC.Model class instance and NWChem.MM.Model class instance respectively.

7> NWChem.QMMM.ModelFactory then computes the QM/MM energy

8> Last step is to destroy all class instances and components.

Using the NWChem-based QM/MM CCA component model, MPQC (and GAMESS) need not write its own piece of computational code for Molecular Mechanics and the combined Quantum Mechanics and Molecular Mechanics calculations which otherwise would have called for large development efforts by various scientists. In addition, theories that are unique to MPQC or GAMESS can be used to model the reactive part of the chemical system.

# CHAPTER 4. PERFORMANCE MEASUREMENT AND ANALYSIS

In this chapter, I will discuss the overhead of using componentizing approach for a QM/MM calculation. Also comparison is done between the execution time needed to calculate QM/MM energy in a legacy code and time needed by using CCA components. To evaluate the execution time of the functions called in a legacy code as well as in CCA components, TAU is used. TAU allows capturing of information at the node/context/thread levels. Using TAU, I perform source level code instrumentation by using interface calls at function entry and exit points. I have integrated the CCA components with TAU Performance component (Figure 10). Performance component provides *MeasurementPort* interface via which we are able to use the measurement capabilities provided by TAU in Ccaffiene framework. Both serial and parallel executions are possible using Ccaffiene framework.

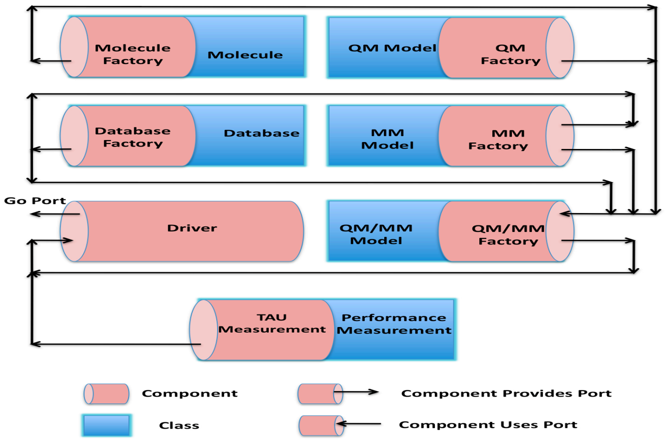


**Figure 10. TAU QM/MM component**

I have specifically used the timer capability provided by the *MeasurementPort* to calculate the execution time of a function. For testing, I chose two molecules: the tripeptide alanine-serine-alanine and ethane. Table 2 shows the execution time for calculation of the QM/MM energy using plain NWChem program.

| ***Molecule*** | ***Basis set*** | ***No of basis function*** | ***NWChem (seconds)*** | ***NWChem CCA Components (seconds)*** |
|---|---|---|---|---|
| tripeptide alanine-serine-alanine | sto-3g | 14 | 0.93 | 1.085 |
| tripeptide alanine-serine-alanine | aug-cc-pVTZ | 210 | 325 | 374 |
| ethane | sto-3g | 16 | 1.36 | 1.571 |
| ethane | aug-cc-pVTZ | 260 | 1015 | 1122 |

**Table 2. Execution time for calculating QM/MM energy using QM/MM component model**

Also, it shows the execution time taken by the component model of QM/MM for the same molecules and basis sets. These tests were performed on a SMP cluster of 4 nodes. Each node has a dual core Xeon 2 GHz CPU with 8GB of RAM with Red Hat Linux as the operating system.

ParaProf [32], a tool for analyzing the profiles generated by TAU is used to observe the profiling output. Thus, by using profiling, the best-known method for observing the performance of software, it can be seen that the overhead of componentization decreases as

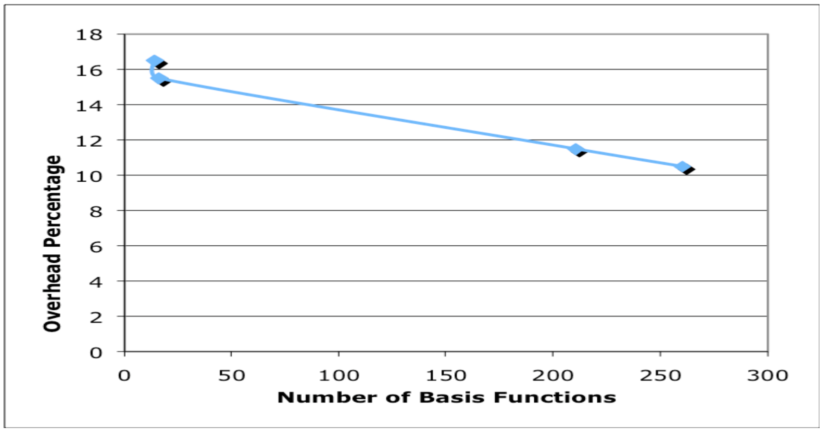


**Figure 11. Performance overhead of QM/MM component model**

larger molecules are simulated. For sto-3g basis set, overhead is 16% and for aug-cc-pVTZ, it is 11%. Thus, it can be observed from Figure 11 that the overhead reduces as the number of basis functions increases.

# CHAPTER 5. SUMMARY AND DISCUSSION

In this thesis, I outlined the process of creation of scientific components from legacy software using CCA specifications and discussed how to make these components truly interoperable with other components. I also discussed about the overhead of componentization and concluded that the overhead is negligible when compared to the overall calculation time. The main motive of this research work is to minimize the programming efforts of the scientists and at the same time not to impose a significant overhead as a result of componentization. The success of the proposed QM/MM component model will encourage further research and implementation of component engineering methods.

## 5.1 Lessons learned

Component development should not be monolithic thus allowing different component developers to incrementally develop the application. This allows development using different previously built and tested components without implementing the entire model at once. In this work, I have used the components previously developed for quantum calculations and modified them as per the requirements (while maintaining their original intent of usage) for combined quantum mechanics and molecular mechanics calculations.

Throughout the development process, object oriented principles should be wisely used so program development will not lead to creation of monolithic applications. I have used object-oriented principles such as writing generic interfaces and implementation of interfaces by classes thus avoiding the unpleasant effects of multiple inheritance. Data and methods are tied together in the form of a class and message passing is possible via objects in the CCA-compliant application, thus utilizing the advantages of encapsulation and data hiding.

A variety of tools, libraries, compilers and frameworks are involved in component development; it is worthwhile checking with different versions if errors pop up in the development process. It might be the case that a bug in a tool or compiler for instance traverses down to your application. This can be a very time consuming process, but is necessary part of debugging the component applications.

Application development should be split into different parts, which can be modified independently, and unit tested from the rest of the application.

Design is the most important part of the componentization approach. Team discussion is very necessary to discuss and resolve design issues and come up with generic flexible interfaces.

Componentizing a legacy code so that components are interoperable with different packages is critical since we do not want to revamp the original legacy code.  Special care should be taken while writing wrappers so that when components are loaded and interoperated in different environments and frameworks, they still perform their intended task.

Portability is a major issue in application development; the developer should not assume that an application would always run on the same operating system and use the same compiler (versions).

An application can always be optimized. Ideally, optimization should be done at the end of the development stage and the components should be developed in a way that optimized pieces of it are easily glued with the rest of the application.

## 5.2 Potential reusable results

Given the broad scope of the componentizing approach discussed in this thesis, it is worthwhile mentioning the results of this research that can be applied to other projects: Throughout the different stages of the component life cycle starting with requirements gathering, design, and development to testing, the main focus was on building components that can be used in a plug and play manner with other packages. This avoids functionality to be reimplemented by other packages. The issues faced throughout the process can be taken as a guiding tool by different application developers interested in componentizing their packages. In this thesis, I also discussed the overall process of enabling componentization in a legacy code. Different application scientists/ developers can use these guidelines to use the CCA tool with their legacy codes.

In specific, the Model and Model Factories generated and discussed for QM/MM are generic enough to be used in a "plug and play fashion" by different chemistry packages.

Performance is always an issue in software world. In this research, I have used lightweight tools to bring componentization into a legacy code and the results show that there is not a significant overhead. Significant efforts are needed during the componentization process; but in the end the result is worthwhile as there is a significant saving of programming effort and time by not reinventing the “same software” by different application scientists.

## 5.3 Open issues

Componentizing a large-scale high performance legacy package is a challenging task. A variety of tools are used in combination such as CCA, BABEL, Ccaffiene framework, TAU, CCA-Chem Generic, MPICH, C++ and Fortran compilers that are often being updated. To provide consistency, it is required to recompile the packages and test them to ensure stability. The rebuilding process is time consuming and if errors or bug pops up, it is very difficult to locate the source of the error because of the various packages involved and to conclude which particular combination of packages may conflict.

Application decomposition into components is equally critical. There are no hard-coded rules for decomposing a legacy code in components and for determining their level of granularity. It can be said that the smaller the granularity of the components, the more the application is flexible since selective components can be replaced. At the same time, the larger the number of components, the more invariants each component has to respect. In addition, the overall overhead will increase for applications. So taking into consideration the pros and cons, one should wisely develop scientific components.

Performance of the interoperable components will vary with different packages because of the underlying message passing libraries, memory access and coding schemes the different packages use. For example, MPQC uses MPI as the message-passing layer whereas NWChem uses ARMCI and GAMESS uses DDI or MPI.

## 5.4 Future work

Some directions for future work in this area would be to explore a few more important calculations of various packages and extend this work to develop high performance interoperable components for these important computations. Following are the potential possibilities for future work:

1> Develop components for Dynamical Nucleation Theory –Monte Carlo Method.

2> Develop one-body operator implementations, for example, potential, kinetic, overlap, and density.

3> Construction of a multi-level parallel component model on different processors.

4> Incremental progress in the design and implementation of CQoS, increasing the level of automation and making it more generally applicable to component applications.

This research is certainly very exciting as after the object oriented paradigm, component based software development in a high performance computational environment is the next big thing due to the numerous potentialities of component based applications. High performance component based development can skyrocket by demonstrating the benefits of component-based development to real life projects of significant size.

# APPENDIX A. INCORPORATING CCA IN NWCHEM

A variety of packages and tools are involved to bring componentization in a legacy package. For the NWChem computational code following are the prerequisites:

- Install C/C++ and Fortran compilers and MPICH library then configure your login environment to use them.
- Download and install CCA tools (Cccafiene and BABEL) and configure your login environment to use them.
- Download and install CCA-Chem-generic2.
- Download a copy of the NWChem Package.
- Download and install TAU for performance.

Figure 12 shows a variety of tools/packages involved in a component based chemistry project. GNU build system (autoconf and libtool) is used to incorporate CCA inside NWChem. It allows us to write cross-platform software for Unix-like systems. Since autoconf and libtool are important in the building and maintenance of the CCA components, desriptions of these tools and how they are used in NWChem are given next.

autoconf is a gnu tool used for producing shell scripts that automatically configure source code packages to many kinds of UNIX-like systems. A shell script wrapper around a macro language, m4, forms the basis of autoconf. m4 is a preprocessor which besides just macro expansion has built-in functions for:

- file inclusion

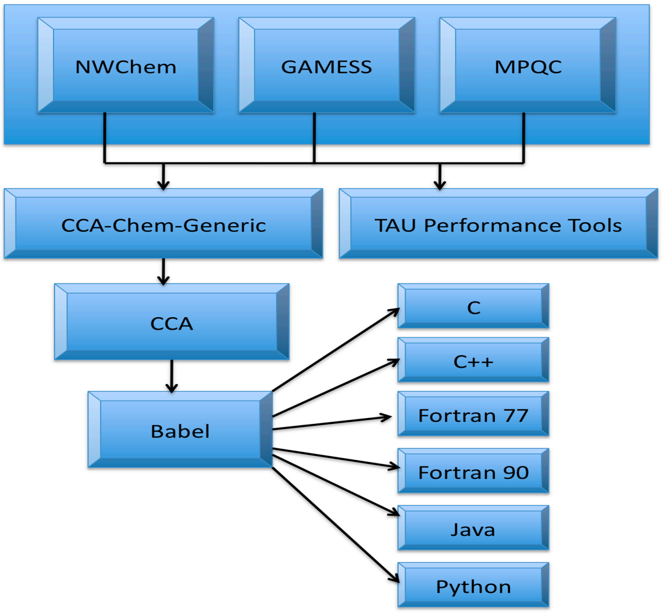


**Figure 12. Packages/tools involved in componentization a chemistry package**

- string manipulation
- conditional evaluation
- performing arithmetic expressions and recursion
- running shell commands

In the NWChem-CCA base directory, there is a file called ‘aclocal.m4’ which is a series of m4 commands. It is used to set up configuration variables and relies on some predefined variables. There is a function called ‘AC_CHECK_CCA’ which detects the CCA tools environment (checks for BABEL libtool, BABEL C/C++/Fortran compilers, mpi configuration, python configuration, cca-specs).

Another important file is ‘configure.in’ which is basically a series of m4 commands. Here are some useful macros that autoconf uses in the ‘configure.in’ file:

1. **AC_INIT** (package, version, [bug-report], [tarname])

   Processes any command-line arguments and some initialization.

   Sets the name of the package and version.

2. **AC_OUPTUT**

   Generates and runs config.status, which creates the makefiles and other files with the suffix .in in the directory structure.

   Called once at the end of the configure.in and is basically a list of all of the files that need to be generated by configure.

3. **AC_ARG_WITH** (package, help-string, [action-if-given], [action-if-not-given])

   This is how you pick up the variables defined in the configure with --with-package or --without-package. The variable withval holds the value given on the right side of the = when the package is configured.

4. **AC_ARG_ENABLE**(feature, help-strong,[action-if-given], [action-if-not-given])

   This is how you "turn on" optional features using the --enable-feature option in configure. For example, you can pick to compile the F90 components instead of just the C++ components. The idea here is that you should not use this feature to swap features, but to add them. The variable enableval holds the value given on the right side of the = when the package is configured. If no option is given on the --enable-feature line, then enableval is yes.

5. **AC_SUBST**(variable,[value])

   These are variables that will be substituted in the ".in" files when configure is run.

6. **AC_DEFUN**(macro-name, macro-body)

   Allow the developers to define their own macros. The NWChem aclocal.m4 has quite a few of these.

7. **AC_PROG_INSTALL**

   Finds the location of the install-sh script. In my case it is in the $NWCHEM_TOP/src/cca directory.

Next, a configure file is generated from the configure.in and aclocal.m4 files through the autoconf tool with no arguments. autoconf processes ‘configure.in’ to produce a configure shell script. 'configure’ is a portable shell script which examines the build environment to determine which libraries are available, which features the platform has, where libraries and headers are located, and so on. Based on this information, it modifies compiler flags, generates makefiles, and/or outputs the file ‘config.h’ with appropriate preprocessor symbols defined.

Typing ‘configure’ configures the package to adapt to different systems. The following are a few options that can be used with configure:

- --prefix=PREFIX        install architecture-independent files in PREFIX [/usr/local]

- --disable-FEATURE do not include FEATURE (same as --enable-FEATURE=no)
- --enable-FEATURE[=ARG] include FEATURE [ARG=yes]
- --with-PACKAGE[=ARG] use PACKAGE [ARG=yes]
- --without-PACKAGE do not use PACKAGE (same as --with-PACKAGE=no)

For example, to configure CCA within NWChem, use:

configure --prefix=$HOME/NWChem/src/cca/obj --enable-f90 --with-nwchem-dir=$HOME/NWChem --with-cca-chem-config=$HOME/CCA-Chem-Generic2-Install/bin/cca-chem-config --with-nwchem-target=LINUX64

Currently, this allows the user to configure the value variables using values of --with-nwchem-dir and --with-nwchem-target to set the environment variables NWCHEM_TOP and NWCHEM_TARGET needed by NWChem for compilation. These varibales get exported for use in configure by the AC_SUBST command so the variables can be used in the builds. It also sets the location of the cca-chem-config script that sets a group of other environment variables such as CCA_CHEM_INCLUDE, CCA_CHEM_LIB, CCA_CHEM_REPO, CCA_CHEM_BIN, CCA_CHEM_CONFIG, CCA_CHEM_PREFIX and CCAFE_CONFIG that are needed for compiling with CCA and incorporating CCA-Chem-generic. Configure also checks the CCA environment to make sure that everything is operational. Whenever, the behavior of the configure program needs to be changed, configure.in file should be modified and autoconf should be run to get a new configure script. When run, configure also creates several files, replacing configuration parameters in them with appropriate values. The files that configure creates are:

- Each subdirectory that contains something to be compiled or installed contains files such as ‘Makefile.in’. configure performs a simple variable substitution, replacing occurrences of `*@variable@*' in `Makefile.in' with the value that configure has determined for that variable and produces a ‘Makefile’. configure processes all files with a ‘.in’ extension replacing variables and producing a file without the ‘.in’ extension.
- a shell script called `config.status' that, when run, will recreate the files (i.e. reruns config with the options that you gave it).
- a file called `config.log' containing any messages produced by the compiler for debugging if configure failed.

There are several files residing in the $NWCHEM_TOP/src/cca/lib directory that get used by the other makefiles in the code.

- → ‘MakeVars.in’ which configure uses to generate ‘MakeVars’. This is essentially an expanded list of all of the variables that will be used in the makefiles. If a new variable is to be added, ‘MakeVars.in’ should be modified and then configure should be run again. The variable definitions are not all static, some of them are dependent on other variables that are in the Makefiles in the subdirectories. For example the value of CLIENT_OR_SERVER is used in the $NWCHEM_TOP/src/cca/Chemistry/server/cxx/Makefile has an effect on the variables that are used in the compile. Also, CLIENT_OBJDIRS and SERVER_OBJDIRS set in this 'MakeVars' is used by Makefile.libs in the same directory.

- →'Makefile.objs' holds all the compiling rules for the whole makefile structure. For example, it gives the rules for creating the .d files and for compiling the .c files to .lo files. Some of the variables in this file come from the MakeVars file mentioned above. This is also the Makefile that has the commands to run the Babel commands to generate the *Impl*, etc. files. It is only included in the $NWCHEM_TOP/src/cca/Chemistry/server/cxx/Makefile.
- → 'Makefile.libs' holds all of the information for linking the objects together to create libraries. Here, libtools is used through the LTLINK variable to create dynamically linked objects. This is included in the $NWCHEM_TOP/src/cca/lib/Makefile that does the linking of everything together.
- $NWCHEM_TOP/src/cca/lib/Makefile.in sets a few environment variables and does the linking step.
- Makefile.babel sets up the XML repository. It is used to make the BABEL process the .sidl files and update the XML repository.

- ’components.cca.in’ is used to generate the components.cca file by configure. It internally points to the actual libraries that comprise the component. Ccaffeine uses a “path” to determine where it should look for CCA components (specifically the .cca files) If you make a new component, you will need to update the components.cca.in file and run configure again.

The Makefile.in at the $NWCHEM_TOP/src/cca level (and many of the other Makefiles in the subdirectories), include lib/MakeVars to get the list of all of the variables. This is the makefile that makes sure that all of the subdirectories get built and linked. It does some of the BABEL preparation work of creating the repository of xml (.babel-stamp target) and then compiles the chemistry server code (.chemistry-stamp target) by calling the makefile in the $NWCHEM_TOP/src/cca/Chemistry/server/f90 directory. This is also the makefile that takes care of the cleaning and the installation.

The Makefile.in in the $NWCHEM_TOP/src/cca/Chemistry/server/f90 directory defines the source directory that we are working in, the BABEL_TARGETS (note that these should coincide with the classes in the $NWCHEM_TOP/src/cca/lib/components.cca file), and defines if we are using the client or the server code. It also includes the $NWCHEM_TOP/src/cca/lib/Makefile.objs file that gives the rules to generate the object files.

The overall GNU build is shown in Figure 13. The final important discussion is related to the ability of CCA components to interact with NWChem routines. This is possible by means of creating shared libraries i.e libraries which can be shared among programs. Creation of a shared library is a two-step process:

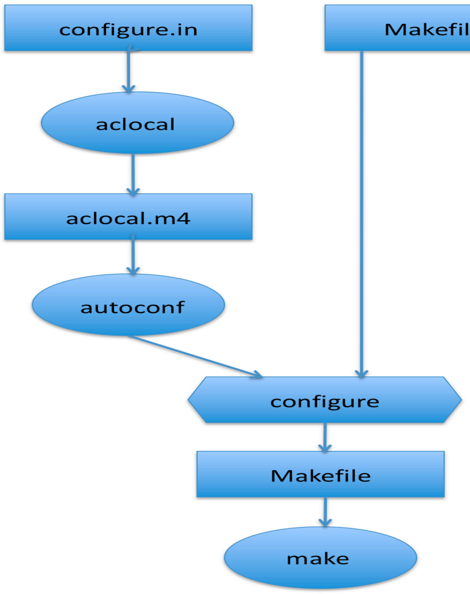


**Figure 13. GNU build system**

1. First, create the object files that will go into the shared library using the –fPIC option associated with the compiler (e.g. gcc, ifort). The -fPIC option is essential to produce *Position-Independent Code* (i.e. to generate code that can be loaded anywhere in the process space of a process). It is also very important for a shared object. By using this option, a number of relocations that have to be performed are cut down to the very minimum. On loading a shared object that is used by an executable, some space has to be allocated for it and the text and data sections have to be allocated some locations. If they

are not built in a position-independent way, then a fair amount of relocations have to be done by the program loading the shared object, thus impacting performance adversely.

2. The second step is to create a shared library using gcc with the –shared option. A shared object called ‘libnwchem-sumo.so’ is created which links the different object files and libraries like Global Arrays, Memory Allocator, ARMCI, etc. CCA then uses this single large shared object file to make NWChem components. A single shared object files is important since there are various dependencies at run-time and cannot be split into various shared object files.

Once, NWChem is compiled and shared object file (libnwchem-sumo.so) is generated, the next step is to write interface definitions and develop NWChem based CCA application by implementing those interfaces.

# APPENDIX B.  QM/MM INTERFACES

```
package Chemistry version 0.4.0 {
/** The Chemistry.Model interface provides energies and
      gradients for a Molecule.  This interface provides the
        primary functionality of a  chemistry code and can be
        extended  by QC, MM and combined QM/MM interfaces.
       */

   interface ModelInterface
  {

     /// Returns the energy.
        double get_energy();

        /** Sets the accuracy for subsequent energy calculations.
           @param acc The new accuracy. */
        void set_energy_accuracy(in double acc);

        /** Returns the accuracy to which the energy is already computed.
           The result is undefined if the energy has not already
           been computed.
           @return The energy accuracy. */
        double get_energy_accuracy();

        /** This allows a programmer to request that if any result
           is computed,
           then the energy is computed too.  This allows, say, for a request
           for a gradient to cause the energy to be computed.  This computed
           energy is cached and returned when the get_energy() member
           is called.
           @param doit Whether or not to compute the energy.
          */
        void set_do_energy(in bool doit);
/** Returns the Cartesian gradient.  */
        array<double,1> get_gradient();

        /** Sets the accuracy for subsequent gradient calculations
           @param acc The new accuracy for gradients. */
```

```
    void set_gradient_accuracy(in double acc);

    /** Returns the accuracy to which the gradient is already computed.
      The result is undefined if the gradient has not already
      been computed.
      @return The current gradient accuracy. */
    double get_gradient_accuracy();

    /** Returns the Cartesian Hessian. @return The Hessian. */
    array<double,2> get_hessian();

    /** Sets the accuracy for subsequent Hessian calculations.
      @param acc The new accuracy for Hessians. */
    void set_hessian_accuracy(in double acc);

    /** Returns the accuracy to which the Hessian is already computed.
      The result is undefined if the Hessian has not already
      been computed. */
    double get_hessian_accuracy();

    /** Returns a Cartesian guess Hessian. */
    array<double,2> get_guess_hessian();

    /** Sets the accuracy for subsequent guess Hessian calculations.
      @param acc The new accuracy for guess Hessians. */
    void set_guess_hessian_accuracy(in double acc);

    /** Returns the accuracy to which the guess Hessian is
already computed.  The result is undefined if the guess Hessian
      has not already been computed.
      @return The guess hessian accuracy.  */
    double get_guess_hessian_accuracy();

    /** This should be called when the object is no longer needed.
      No other members may be called after finalize. */
    int finalize();
 };


   interface ModelFactoryInterface extends gov.cca.Port {

   ModelInterface get_model();
```

```
};


 /** The Molecule interface provides a collection of atoms.

     Each atom is associated with a charge, an atomic number, a Cartesian
     coordinate, and an optional label.

     Ghost atoms (atoms do not contribute a charge) have the appropriate
     atomic number but a zero charge.  Point charges have an atomic number
     of zero, but have a nonzero charge.

   */
  interface MoleculeInterface {

     /** Initialize a molecule.
        @param natom The number of atoms.
        @param npcharge The number of point charges.
        @param unitname The name of the units ("angstroms" or "bohr").
*/
     void initialize(in long natom, in long npcharge, in string unitname);

     /** Returns a units object that corresponds to the units that are used
        by get_cart_coor or set_cart_coor.
        @return The units of the coordinates.
      */
     Physics.UnitsInterface get_units();

     /** Returns the number of atoms.
        @return The number of atoms.
       */
     long get_n_atom();

     /** Returns the number of point charges.
        @return The number of point charges.
       */
     long get_n_pcharge();

     /** Returns the atomic number of an atom
        @param atomnum The number of the atom.
        @return The atomic number.
       */
     int get_atomic_number(in long atomnum);

     /** Sets atomic number of an atom.
        @param atomnum The number of the atom.
        @param atomic_number The atom's new atomic number.
       */
     void set_atomic_number(in long atomnum, in int atomic_number);

     /** Returns net charge of the molecule.
        @return The net charge.
*/
     double get_net_charge();

     /** Sets the net charge of the molecule.
        @parm charge The new net charge.
       */
     void set_net_charge(in double charge);

     /** Returns charge at an atom.
        @param atomnum The number of the atom.
        @return The charge on the atom.
       */
     double get_charge(in long atomnum);

     /** Sets charge of an atom.
        set_net_charge must called as well to ensure internal consistency.
        @param atomnum The number of the atom.
      */
     void set_charge(in long atomnum, in double charge);

     /** Returns point charge value.
        @param atomnum The number of the point charge.
        @return The point charge value.
       */
     double get_point_charge(in long pchargenum);

     /** Sets point charge value
        @param atomnum The number of the point charge.
        @param charge  The point charge value.
      */
void set_point_charge(in long pchargenum, in double pcharge);
```

```
          /** Returns point charge array.
          @return An array with all the point charge values.
        */

        array<double,1> get_all_point_charge();

  /** Gets  Cartesian coordinates for point charges.
          @return The Cartesian coordinate array.
        */

       array<double,1> get_pcharge_coor();

      /** Returns the Cartesian coordinate array.
          @return An array with all Cartesian coordinates.
        */
      array<double,1> get_coor();

      /** Sets the Cartesian coordinates.
          @param x The new coordinates.
        */
      void set_coor(in array<double,1> x);

      /** Gets individual Cartesian coordinate.
@param atomnum The atom number.
          @param xyz Give 0 for x, 1 for y, and 2 for z.
          @return The Cartesian coordinate.
        */
      double get_cart_coor(in long atomnum, in int xyz);

      /** Sets individual Cartesian coordinate.
          @param atomnum The atom number.
          @param xyz Give 0 for x, 1 for y, and 2 for z.
          @param val The new Cartesian coordinate.
        */
      void set_cart_coor(in long atomnum, in int xyz, in double val);

      /** Gets individual Cartesian coordinate for point charge.
          @param pchargenum The point charge number.
          @param xyz Give 0 for x, 1 for y, and 2 for z.
          @return The Cartesian coordinate.
          */
         double get_pcharge_cart_coor(in long pchargenum, in int xyz);

         /** Sets individual Cartesian coordinate for point charge.
           @param pchargenum The point charge number.
           @param xyz Give 0 for x, 1 for y, and 2 for z.
           @param val The new Cartesian coordinate.
          */
         void set_pcharge_cart_coor(in long pchargenum, in int xyz, in double val);

         /** Returns label for atom @param atomnum */
         string get_atomic_label(in long atomnum);

         /** Sets label for atom @param atomnum. */
         void set_atomic_label(in long atomnum, in string label);

       };
      /** The MoleculeFactory is used to create Molecule objects.
         A filename is given to MoleculeFactory which contains
         the data that is used to generated a Molecule.
        */
       interface MoleculeFactoryInterface extends gov.cca.Port {

         /** Set the containing the Molecule data.
           @param filename The filename. */
         void set_molecule_filename(in string filename);

         /** Return the a Molecule object.
           @return A new instances of Molecule. */
         Chemistry.MoleculeInterface get_molecule();

          /** Return a  Molecule object.
              @return A new instance of Molecule.
              @param unitname unit name
              @param atomic_nums atomic numbers
              @param coords geometry coordinates
              @param pcharge_coords point charge coordinates
              @param pcharge_vals point charge values
               */
           Chemistry.MoleculeInterface get_molecule[_qm](in string unitname, in
array<int,1> atomic_nums, in array<double,1> coords,  in array<double,1> pcharge_coords,
in array<double,1> pcharge_vals);
```

```
      /** This should be called when the object is no longer needed.
          No other members may be called after finalize. */
      int finalize();
    }

     package MM {

     interface ModelInterface extends Chemistry.ModelInterface{};
     interface ModelFactoryInterface extends gov.cca.Port {

          /** Returns a newly created MM Model.
              @return The new Model instance.
       */

      Chemistry.MM.ModelInterface get_model();
     };
};

      package QMMM {
     interface ModelInterface extends Chemistry.ModelInterface{
       /** Sets a refernce of an instance of a MM  Model
               for combined QMMM calculation.
           @param mm_model MM Model instance
       */
       void set_mm_model(in Chemistry.MM.ModelInterface mm_model);

        /** Gets a refernce of an instance of a MM  Model
               for combined QMMM calculation.
           @return The refernce of a QM Model instance
       */

       Chemistry.MM.ModelInterface get_mm_model();

       /** Sets a refernce of an instance of a QC(a.k.a QM)  Model
               for combined QMMM calculation.
           @param qm_model QM Model instance
       */

       void set_qm_model(in Chemistry.QC.ModelInterface qm_model);
       /** Gets a refernce of an instance of a QM (a.ka. QC  Model
               for combined QMMM calculation.
        @return The reference of a QM Model instance
      */

      Chemistry.QC.ModelInterface get_qm_model();
      };

    interface ModelFactoryInterface extends gov.cca.Port {
       /** Returns a newly created QMMM Model.
          @return The new Model instance.
  */

     Chemistry.QMMM.ModelInterface get_model();
    };
};

/*
Database acts as a storage model for geometry, point charge coordinates, energies,
etc.
*/

package Database version 0.1

{
/*
  The Model interface provides the storage space for geometry coordinates,
  point charges and energies

*/
interface ModelInterface
{
 /* Connect to the database
   @param  mode : can take values like  read or write
   @return handle to control the database
 */
int connect(in string mode);

/*
 Disconnect from the database
 @param handle database handle created by connect method
 @return returns a flag value about the status of the termination
*/
int disconnect(in int handle);
```

```
/*
   insert cartesian coordinates in the database
   @param coords cartesian coordinates
*/
int insertCoords(in array<double,1> coords);
/*
   insert atomic numbers in the database
   @param atomic_nums atomic numbers
*/

int insertAtomicNums(in array<int,1> atomic_nums);
/*
   insert point charges in the database
   @param pcharges pcharges
*/

int insertPCharges(in array<double,1> pcharges);

/*
   insert cartesian coordinates for point charges in the database
   @param coords cartesian coordinates for point charges
*/
int insertPChargeCoords(in array<double,1> pcharge_coords);
/*
   insert units of the coordinates in the database
   @param units units of the coordinates
*/
int insertUnits(in string units);
/*
  insert energy of a particular region
  @param region region which can be either QM, MM or QMMM
  @param energy energy value of the specified region
*/
  int insertEnergy(in string region, in double energy);
/*
   return the carteisan coordinates stored in the database
   @return cartesian coordinates
*/
array<double,1> retrieveCoords();
```

# APPENDIX C.  PROTIEN DATA BANK (PDB) FILE FORMAT

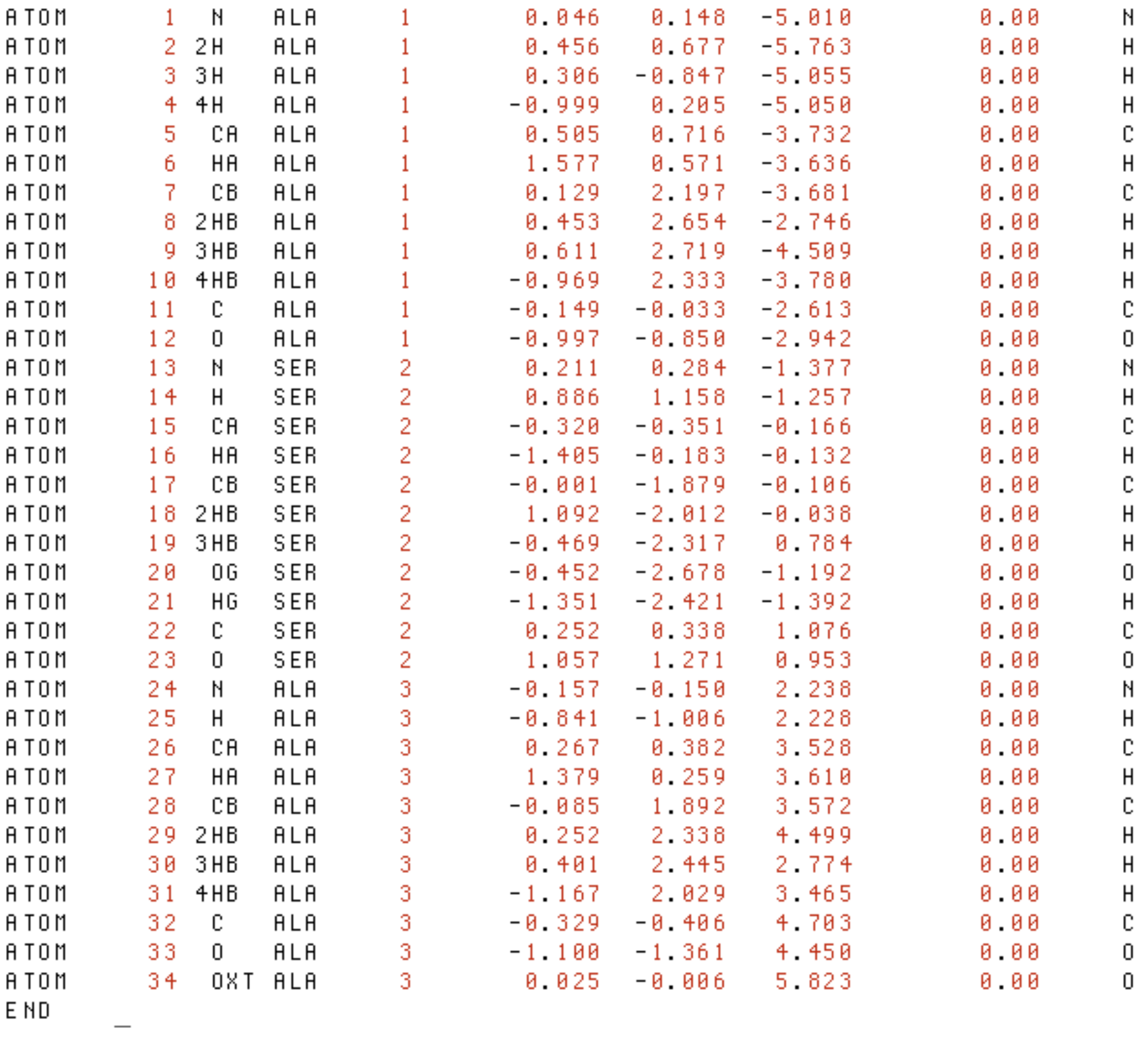
The prepare module in NWChem analyses the coordinates specified in PDB formatted file. The typical extension for a PDB file is .pdb. The PDB format is generally used for proteins but it can be used for other molecules as well. PDB file contains information about the molecules in the form of 3D structures.

The PDB file can be viewed as collection of records. All the records must appear in a defined order. Mandatory records are present in all the entries. HEADER, TITLE, REMARK and CRYST1 are some of the mandatory records present in a PDB file. There is another record type called ATOM, which is critical in QM/MM calculation. The ATOM records present the atomic coordinates in angstroms. They also present the occupancy and temperature factor for each atom. The record format is

```
ATOM      1  N   ALA     1       0.046   0.148  -5.010       0.00     N
```

where first column is the record name or type (in this case ATOM), second column denotes the atom serial number, third column denotes the atom, fourth denotes the structure, fifth  is the alternate location indicator. Columns sixth, seventh and eight define X, Y and Z orthogonal coordinates respectively in angstroms. The last two columns specify the charge on the atom and the atom name.

An example of PDB file of tripeptide alanine-serine-alanine is shown in Figure 14.



```
ATOM      1  N   ALA   1       0.046   0.148  -5.010       0.00     N
ATOM      2 2H   ALA   1       0.456   0.677  -5.763       0.00     H
ATOM      3 3H   ALA   1       0.306  -0.847  -5.055       0.00     H
ATOM      4 4H   ALA   1      -0.999   0.205  -5.050       0.00     H
ATOM      5  CA  ALA   1       0.505   0.716  -3.732       0.00     C
ATOM      6  HA  ALA   1       1.577   0.571  -3.636       0.00     H
ATOM      7  CB  ALA   1       0.129   2.197  -3.681       0.00     C
ATOM      8 2HB  ALA   1       0.453   2.654  -2.746       0.00     H
ATOM      9 3HB  ALA   1       0.611   2.719  -4.509       0.00     H
ATOM     10 4HB  ALA   1      -0.969   2.333  -3.780       0.00     H
ATOM     11  C   ALA   1      -0.149  -0.033  -2.613       0.00     C
ATOM     12  O   ALA   1      -0.997  -0.850  -2.942       0.00     O
ATOM     13  N   SER   2       0.211   0.284  -1.377       0.00     N
ATOM     14  H   SER   2       0.886   1.158  -1.257       0.00     H
ATOM     15  CA  SER   2      -0.320  -0.351  -0.166       0.00     C
ATOM     16  HA  SER   2      -1.405  -0.183  -0.132       0.00     H
ATOM     17  CB  SER   2      -0.001  -1.879  -0.106       0.00     C
ATOM     18 2HB  SER   2       1.092  -2.012  -0.038       0.00     H
ATOM     19 3HB  SER   2      -0.469  -2.317   0.784       0.00     H
ATOM     20  OG  SER   2      -0.452  -2.678  -1.192       0.00     O
ATOM     21  HG  SER   2      -1.351  -2.421  -1.392       0.00     H
ATOM     22  C   SER   2       0.252   0.338   1.076       0.00     C
ATOM     23  O   SER   2       1.057   1.271   0.953       0.00     O
ATOM     24  N   ALA   3      -0.157  -0.150   2.238       0.00     N
ATOM     25  H   ALA   3      -0.841  -1.006   2.228       0.00     H
ATOM     26  CA  ALA   3       0.267   0.382   3.528       0.00     C
ATOM     27  HA  ALA   3       1.379   0.259   3.610       0.00     H
ATOM     28  CB  ALA   3      -0.085   1.892   3.572       0.00     C
ATOM     29 2HB  ALA   3       0.252   2.338   4.499       0.00     H
ATOM     30 3HB  ALA   3       0.401   2.445   2.774       0.00     H
ATOM     31 4HB  ALA   3      -1.167   2.029   3.465       0.00     H
ATOM     32  C   ALA   3      -0.329  -0.406   4.703       0.00     C
ATOM     33  O   ALA   3      -1.100  -1.361   4.450       0.00     O
ATOM     34  OXT ALA   3       0.025  -0.006   5.823       0.00     O
END    _
```

**Figure 14. Sample PDB file for tripeptide alanine-serine-alanine**

The national PDB format is extremely complex and contains much more information. I have extracted the information from the point of view of a QM/MM calculation.

# BIBLIOGRAPHY

[1] E. J. Bylaska, W. A. de Jong, N. Govind, K. Kowalski, T. P. Straatsma, M. Valiev, D. Wang, E. Apra, T. L. Windus, J. Hammond, P. Nichols, S. Hirata, M. T. Hackler, Y. Zhao, P.-D. Fan, R. J. Harrison, M. Dupuis, D. M. A. Smith, J. Nieplocha, V. Tipparaju, M. Krishnan, Q. Wu, T. Van Voorhis, A. A. Auer, M. Nooijen, E. Brown, G. Cisneros, G. I. Fann, H. Fruchtl, J. Garza, K. Hirao, R. Kendall, J. A. Nichols, K. Tsemekhman, K. Wolinski, J. Anchell, D. Bernholdt, P. Borowski, T. Clark, D. Clerc, H. Dachsel, M. Deegan, K. Dyall, D. Elwood, E. Glendening, M. Gutowski, A. Hess, J. Jaffe, B. Johnson, J. Ju, R. Kobayashi, R. Kutteh, Z. Lin, R. Littlefield, X. Long, B. Meng, T. Nakajima, S. Niu, L. Pollack, M. Rosing, G. Sandrone, M. Stave, H. Taylor, G. Thomas, J. van Lenthe, A. Wong, and Z. Zhang, "NWChem, A Computational Chemistry Package for Parallel Computers, Version 5.1", Pacific Northwest National Laboratory, Richland, Washington 99352-0999, USA. A modified version, 2007

[2] Kendall, R.A.; Apra, E.; Bernholdt, D.E.; Bylaska, E.J.; Dupuis, M.; Fann, G.I.; Harrison, R.J.; Ju, J.; Nichols, J.A.; Nieplocha, J.; Straatsma, T.P.; Windus, T.L.; Wong, A.T., "High Performance Computational Chemistry: An Overview of NWChem a Distributed Parallel Application,", Computer Phys. Comm. 2000, 128, 260-283.

[3] M.W. Schmidt, K.K. Baldridge, J.A. Boatz, S.T. Elbert, M.S. Gordon, J.H. Jensen, S. Koseki, N. Matsunaga, K.A. Nguyen, S. Su, T.L. Windus, M. Dupuis, and J.A. Montgomery, Jr., "The General Atomic and Molecular Electronic Structure System", J. Comp. Chem., 14, 1347, 1993.

[4] Curtis L. Janssen, Ida B. Nielsen, Matt L. Leininger, Edward F. Valeev, Joseph P. Kenny, Edward T. Seidl, "The Massively Parallel Quantum Chemistry Program (MPQC)", version 3, Sandia National Laboratories, Livermore, CA, USA, 2008.

[5] COM: Component Object Method Technologies. http://www.microsoft.com/com/default.mspx

[6] Enterprise JavaBeans Technology. http://java.sun.com/products/ejb/

[7] CORBA Component Technolgy, version 4.0, 2008. http://www.omg.org/technology/documents/formal/components.htm

[8] Common Component Architecture Forum. http://www.cca-forum.org

[9] SS Shende, AD Malony, "The TAU Parallel Performance System", International Journal of High Performance Computing Applications, 2006.

[10] J Nieplocha, B Carpenter, "ARMCI : A portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems", Lecture Notes in Computer Science, 1999.

[11] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease and Edo Apra, "Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit", International Journal of High Performance Computing Applications, Vol. 20, No. 2, 203-231p, 2006.

[12] Jarek Nieplocha, Manojkumar Krishnan, Bruce Palmer, Vinod Tipparaju, and Jialin Ju, "The Global Arrays User's Manual", 2006.

[13] Global Arrays Webpage, http://www.emsl.pnl.gov/docs/global/.

[14] Levine, Ira N, *"Quantum Chemistry"*. Pearson Prentice Hall, New Jersey, 2008.

[15] Andrew Cleary, Scott Kohn, Steven G. Smith, and Brent Smolinski, "Language Interoperability Mechanisms for High-Performance Scientific Applications", SIAM Workshop on Object-Oriented Methods for Inter-operable Scientific and Engineering Computing, Yorktown Heights, NY, 1998. LLNL document UCRL-JC-131823.

[16] Madhusudhan Govindaraju, Michael R. Head, Kenneth Chiu, "XCAT-C++: Design and Performance of a Distributed CCA Framework", The 12th Annual IEEE International Conference on High Performance Computing (HiPC) 2005, pp: 270-279.

[17] M. Govindaraju, S. Krishnan, K. Chiu, A. Slominski, D. Gannon, and R. Bramley, "Merging the cca component model with the ogsi framework," in Proceedings of CCGrid2003, 3rd International Symposium on Cluster Computing and the Grid, 2003, pp. 182–189.

[18] Felipe Bertrand and Randall Bramley, "DCA: A distributed CCA framework based on MPI", Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04), 2004.

[19] Gary Kumfert, Scott Kohn, Tammy Dahlgren, Tom Epperly, Steve Smith, and Bill Bosl, "Introducing Babel Decaf", Common Component Architecture Forum, Indiana University, Bloomington, IN, 2001. LLNL document UCRL-PRES-145982.

[20] The CCA Integration Framework, https://www.cca-forum.org/wiki/tiki-index.php?page=CCAIN.

[21] B Allan, R Armstrong, S Lefantzi, J Ray, E Walsh, P Wolfe, “Ccaffeine – a CCA component framework for parallel computing”, 2003, http://www.cca-forum.org/ccafe.

[22] Szyperski, Clemens “Component Software: Beyond Object-Oriented Programming”, Addison-Wesley, 2002.

[23] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, C. L. Janssen, J. P. Kenny, I. M. B. Nielsen, M. Krishnan, V Gurumoorthi, E. F. Valeev, and T. L. Windus, “Enabling new capabilities and insights from quantum chemistry by using component architectures”, Journal of Physics: Conference Series, 46 220-228, 2006.

[24] Manojkumar Krishnan, Yuri Alexeev, Theresa Windus and Jarek Nieplocha, “Multilevel Parallelism in Computational Chemistry using Common Component Architecture and Global Arrays”, proceedings of Supercomputing, Seattle, WA, 2005.

[25] Joseph P. Kenny, Curtis L. Janssen, Edward F. Valeev, and Theresa L. Windus, “Components for Integral Evaluation in Quantum Chemistry”, Journal of Computational Chemistry, 2007.

[26] Fang Peng, Meng-Shiou Wu, Masha Sosonkina, Ricky A. Kendall, Michael W. Schmidt, Mark S. Gordon, Coupling GAMESS via Standardized Interfaces, HPC-GECO/Compframe, 2006.

[27] Fang Peng, Meng-Shiou Wu, Masha Sosonkina, Theresa Windus, Jonathan Bentz, Mark S. Gordon, Joseph P. Kenny, Curtis Janssen, “Tackling Component Interoperability in Quantum Chemistry Software”, HPC-GECO/CompFrame, 2007.

[28] Norris, B., Ray, J., McInnes, L., Bernholdt, D. Elwasif, W., Malony, A. and Shende, S., "Computational quality of service for scientific components", International Symposium on Component-based Software Engineering (CBSE7), 2004.

[29] Elwasif W, Norris B, Allan B and Armstrong R, "Bocca: a development environment for HPC components", Compframe, 2007.

[30] World Wide Protien Data Bank, 2007, http://www.wwpdb.org/documentation/format30/index.html.

[31] Juval Lowy, "Programming .NET Components: Design and Build .NET Applications Using Component-Oriented Programming", O'Reilly, 2005

[32] Robert Bell, Allen D. Malony, and Sameer Shende, "ParaProf: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis", Euro-Par 2003.

# ACKNOWLEDGEMENTS

I would like to acknowledge Dr. Theresa Windus for her enthusiastic supervision throughout this entire journey without whom I would have struggled to find the inspiration needed to complete this thesis. I would also like to thank my committee members, Dr. Masha Sosonkina, Dr. Les Miller and Dr. Simanta Mitra for their contribution and advice. Finally, I would like to thank the members of the CCA forum for their support and comments.