Tackling Component Interoperability in Quantum Chemistry Software

Fang Peng
Scalable Computing Laboratory
Ames Laboratory, US DOE
fangp@scl.ameslab.gov

Meng-Shiou Wu
Scalable Computing Laboratory
Ames Laboratory, US DOE
mswu@scl.ameslab.gov

Masha Sosonkina
Scalable Computing Laboratory
Ames Laboratory, US DOE
masha@scl.ameslab.gov

Theresa Windus
The Department of Chemistry
Iowa State University, USA
theresa@fi.ameslab.gov

Jonathan Bentz
Cray Inc.
1340 mendota Heights Rd.
Mendota Heights, MN, USA
jnbntz@cray.com

Mark S. Gordon
The Department of Chemistry and Scalable Computing Laboratory, Ames Laboratory, US DOE
mark@si.fi.ameslab.gov

Joseph P. Kenny
Sandia National Laboratories
Livermore, CA, USA
jpkenny@sandia.gov

Curtis Janssen
Sandia National Laboratories
Livermore, CA, USA
cljanss@sandia.gov

Abstract

The Common Component Architecture (CCA) offers an environment that allows scientific packages to dynamically interact with each other through components. Conceptually, a computation can be constructed with plug-and-play components from any componentized scientific package; however, providing such plug-and-play components from scientific packages requires more than componentizing functions/subroutines of interest, especially for large-scale scientific packages with a long development history. In this paper, we present our efforts to construct components for the integral evaluation - a fundamental sub-problem of quantum chemistry computations - that conform to the CCA specification. The goal is to enable fine-grained interoperability between three quantum chemistry packages, GAMESS, NWChem, and MPQC, via CCA integral components. The structures of these packages are quite different and require different approaches to construct and exploit CCA components. We focus on one of the three packages, GAMESS, delineating the structure of the integral computation in GAMESS, followed by our approaches to its component development. Then we use GAMESS as the driver to interoperate with integral components from another package, MPQC, and discuss the possible solutions for interoperability problems along with preliminary results.

Categories and Subject Descriptors D.2.13 [**Software**]: Software Engineering – reusable software

General Terms Measurement, Performance, Design.

Keywords components, integral, interoperability, quantum chemistry

1. Introduction

The advance of component technologies in high performance computing offers an opportunity for scientific packages to dynamically interact with each other without manually dumping files, converting data formats or painstakingly coupling codes on a case-by-case basis. With the Common Component Architecture (CCA) [1, 2], scientists are able to construct new computations or improve the performance of their software by using components provided by other research groups through well-defined interfaces. This potential of interoperability encourages application scientists from different scientific domains to explore mechanisms to couple existing packages that offer different computing capabilities.

The standards of CCA are defined by the CCA Forum [2], a group of scientists from different national laboratories and academic institutes who are researchers in the high performance computing community. The language interoperability of CCA is enabled by Babel [3], a tool for solving the interoperability of components that are implemented in different programming languages such as FORTRAN, C, C++, Python, and Java. Babel relies on the Scientific Interface Definition Language (SIDL) for defining interfaces for scientific components.

Quantum chemistry is one of the scientific disciplines that are actively involved in exploring the interoperability capability offered by CCA. The complexity in quantum chemistry computations results in a large number of noncommercial packages developed by research laboratories and universities (The General Atomic and Molecular Electronic Structure System - GAMESS [4], Massively Parallel Quantum Chemistry - MPQC [5], and NWChem [6] are three major quantum chemistry packages from DOE and DOD), each with unique capabilities and deficiencies. The development of a new method usually requires doctoral level researchers and is very time-consuming; it is thus an important task to integrate capabilities of different packages to enable new computations that are not possible with any single package.

While CCA offers an environment for scientific packages to interact with each other, a package must be componentized before it is able to provide/use components to/from other packages. With the long development history

HPC-GECO/CompFrame'07 October 21–22, 2007, Montréal, Québec, Canada.

of quantum chemistry packages, efforts for their componentization cannot be accomplished by any single research group. Scientists must join together to define a set of standardized interfaces and data structures for computations of interest, and then packages can be componentized accordingly.

Even with the standardized interfaces defined, componentizing a package with a long development history poses a big challenge, which must be conquered before enabling interoperability between packages. While componentizing quantum chemistry packages on a coarse-grain level was conducted in previous studies [7, 8], another important and useful approach for the quantum chemistry community is to componentize low-level computations such as molecular integral evaluations.

In this paper we detail the process of componentizing the integral computation in GAMESS, with discussion of the difficulties we encountered and preliminary results. With the initial interoperability accomplished, two future research avenues open up: constructing more complex computations and Computational Quality of Service (CQoS) [9] in quantum chemistry.

2. Integral Computation in Quantum Chemistry

The calculation of one-electron integrals (1- or 2-center integrals, where a center refers to a specific atom in a molecule) and two-electron integrals (1-, 2-, 3-, or 4-center integrals) is the basis of constructing the Fock matrix in any quantum chemistry package that uses the Self-Consistent Field (SCF) method. The one- and two-electron integrals in the atomic basis [10] are given in Eqs. (1) and (2), respectively:

$$\langle \chi_\alpha | h | \chi_\beta \rangle = \int \chi_\alpha(1)\left(-\frac{1}{2}\nabla^2\right)\chi_\beta(1)dr_1 + \sum_a \int \chi_\alpha(1)\frac{Z_a}{|R_a - r_1|}\chi_\beta(1)dr_1 \quad (1)$$

$$\langle \chi_\alpha \chi_\gamma | g | \chi_\beta \chi_\delta \rangle = \int \chi_\alpha(1)\chi_\gamma(2)\frac{1}{|r_1 - r_2|}\chi_\beta(1)\chi_\delta(2)dr_1 dr_2 \quad (2)$$

where χ is a basis function (or Atomic Orbital, or AO); α, β, γ, and δ are the indexes of the basis functions; h is the one-electron operator and g is the two-electron operator. The basis function is a linear combination of primitive Gaussians, all of the same type and all on the same nucleus, but with different exponents:

$$\chi_\alpha = \sum_k d_{k\alpha} x^l y^m z^n e^{-\delta_k r^2} \quad (3)$$

where k is the index of the primitive Gaussians, d_{ka} is a contraction coefficient, δ_k is the exponent, x, y, z are the Cartesian coordinates of the nucleus, and $r^2 = x^2 + y^2 + z^2$. The angular momentum of the shell type (S, P, D, F, G, …) is given by $l + m + n$. For example, when $l + m + n = 0$, we get an S-type basis function,

$$\chi_\alpha = \sum_k d_{k\alpha} e^{-\delta_k r^2} \quad (4)$$

and when $l + m + n = 1$, we have 3 types of different basis functions,

$$\chi_\alpha = \sum_k d_{k\alpha} x e^{-\delta_k r^2} \quad (5)$$

$$\chi_\alpha = \sum_k d_{k\alpha} y e^{-\delta_k r^2} \quad (6)$$

$$\chi_\alpha = \sum_k d_{k\alpha} z e^{-\delta_k r^2} \quad (7)$$

where the formulas (5), (6) and (7) correspond to the P_x, P_y and P_z basis functions, respectively

In practice, integrals are calculated in batches, where a batch is a collection of integrals having the same exponent [10] (in this paper, we use the term *Gaussian shell* or *shell* to represent a set of basis functions with the same exponent). For example, a <pp|pp> type batch has 81 individual integrals, where the basis function for a P-type shell has 3 types (3*3*3*3 = 81). We usually call a batch of one-electron integrals a *shell doublet* and a batch of two-electron integrals a *shell quartet*.

In short, to compute the one- and two-electron integrals, we need the ***one-electron operator***, the ***two-electron operator***, the ***basis set*** information and the ***coordinates of the atoms in the molecule (geometry).*** Different packages may use different techniques and can handle different sets of basis functions to calculate integrals.

2.1 Integral Computation in GAMESS

GAMESS is an *ab initio* program that is written mostly in FORTRAN 77, with a small portion designed in C. Using FORTRAN 77 to develop GAMESS was the best choice when the project started, and has enabled GAMESS to run on any platform. However, it also made the componentization of GAMESS a challenging task as no object-oriented concepts have been used in designing GAMESS. The development of the basic GAMESS CCA architecture was described in our previous work [8].

Global information in GAMESS, such as the program configuration, the basis set information and molecule coordinates, is stored as common blocks to be shared between subroutines. For some computations, intermediate data are stored as disk files to be used iteratively. The approach that GAMESS uses to handle global information complicates the componentizing process since we cannot simply pass pointers to global information between subroutines as in other object-oriented or modularized programs.

GAMESS computes two kinds of AO integrals, one- and two-electron integrals. For two-electron integrals,

GAMESS provides four computational methods, each of which has its strength for computing different sets of shell types. By default GAMESS chooses the most efficient one by picking the best method for each shell quartet. However, users can choose a specific integral code through the input options.

2.2 Integral Computation in MPQC and NWChem

The Massively Parallel Quantum Chemistry Program (MPQC), written in the C++ programming language, computes properties of atoms and molecules from first principles. MPQC has been designed as a massively parallel program from the beginning, and it can run on a wide range of platforms, from UNIX workstations, symmetric multi-processors, to massively parallel architectures.

The class libraries underlying the MPQC program are written in C++ using an object-oriented design. Following a class hierarchy very similar to the CCA integral interfaces [12], the integral packages are encapsulated by integral evaluator and integral factory interfaces described within the MPQC documentation [11]. This encapsulation insures a clean separation of the integrals code which greatly simplified packaging the integral packages within MPQC as stand-alone components.

NWChem is a quantum chemistry package that is written in FORTRAN 77. It uses an object-oriented design and programming approach to facilitate functionality reuse and hide internal data. One example of this is the integral abstract programming interface (API) of NWChem. The API exposes only specific aspects of the integral computation to the programmer and hides many of the details with regard to which integral programs are used (there are currently four different algorithms within NWChem) and how the computations are done. This API has initialization routines that require the geometry and the basis set as well as a termination routine that cleans up and terminates the integral computations. There is a set of routines based on the type of integrals to be computed (energy, first or second derivative). In addition, the API allows the programmer to select the accuracy (or the threshold for radial cutoffs) for the integrals. Once the API has been initialized there are specific routines to tell the programmer how much memory is needed for the buffers required by the API and then to call each of the different types of integrals that are available. This architecture allows any improvements or new integral routines to be automatically realized throughout the whole of NWChem.

NWChem also has basis set objects and geometry objects that must be properly populated so that the integral computations work. The population of these objects is usually initiated through an input file although they can also be created through functions associated with the objects. This is particularly useful in the context of CCA.

3. Development of Integral Components

3.1 Integral Evaluation Interfaces

The SIDL interfaces for integral evaluation are available in the ***cca-chem-generic*** package [12]. The ***cca-chem-generic*** package defines several chemistry interfaces that each chemistry package can implement to create chemistry components and classes. In the design of those chemistry interfaces, the interface for a "component" usually ends with "FactoryInterface" and acts as a driver to return references to some classes, while a "class" usually provides real computations. The implementation of a component is only different from the implementation of a class in that a component also needs to implement the *gov.cca.Component* and *gov.cca.Port* interfaces.

The ***cca-chem-generic*** package provides the implementation of several useful components and classes that are needed by most computations. For example, the *Chemistry.MoleculeFactory* component would create and return a *Chemistry.Molecule* class that provides the information of atomic and molecular coordinates for all packages to use.

Among the chemistry interfaces defined by the ***cca-chem-generic*** package there are four core interfaces for integral computations: *IntegralEvaluator1Interface* for 1-center integrals, *IntegralEvaluator2Interface* for 2-center integrals, *IntegralEvaluator3Interface* for 3-center integrals and *IntegralEvaluator4Interface* for 4-center integrals. We call any classes that implement the above interfaces *integral evaluators*. Another core interface is *IntegralEvaluatorFactoryInterface,* which serves as a driver that returns references to the *integral evaluators*. An *integral evaluator factory* that implements

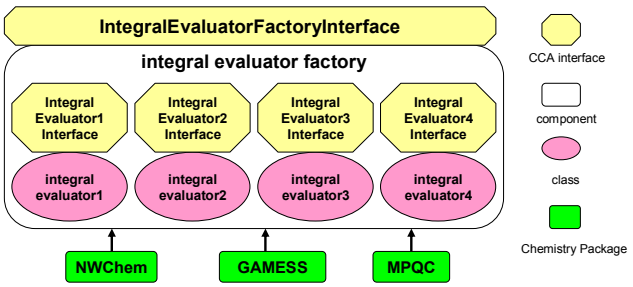

Figure 1. Each chemistry package can implement the *IntegralEvaluatorFactoryInterface* to provide an *integral evaluator factory* component and implement one or more of *IntegralEvaluatorNInterface* (*N*=1, 2, 3 and 4) to provide the *integral evaluatorN* classes. The *integral evaluator factory* component is a driver component to return the references to *integral evaluators* for integral computations.

IntegralEvaluatorFactoryInterface usually also extends the *gov.cca.Component* and *gov.cca.Port* interfaces and is used to provide *integral evaluators* for each chemistry package. Figure 1 shows the relationship among those five core integral interfaces and the three chemistry packages.

The integral evaluator interface provides a *compute* method for integral computation for a shell multiplet. For example, the *compute* method of *IntegralEvaluator2Interface* is for computing a shell doublet, which is illustrated below,

```
/** Compute a shell doublet of integrals.
@param shellnum1 Gaussian shell number 1.
@param shellnum2 Gaussian shell number 2. */
void compute(in long shellnum1, in long
shellnum2);
```

where two indexes of Gaussian shells are passed as parameters and the resulting integrals are stored in a buffer. Similarly, the *compute* method of *IntegralEvaluator4Interface* needs four indexes of Gaussian shells as parameters to compute integrals for a shell quartet.

Several auxiliary interfaces are also important to the initialization of integral evaluators: *CompositeIntegralDescrInterface*, *MoleculeInterface*, *MolecularInterface*, *AtomicInterface* and *ShellInterface*. Through these interfaces, the information required for computing integrals can be passed from one package to another package without initializing every package. Figure 2 shows an example of how molecule coordinates and the basis set are stored in CCA integral objects. The ***cca-chem-generic*** package provides implementations for *CompositeIntegralDescrInterface* and *MoleculeInterface* that will be used directly in our experiments. Detailed information about the integral evaluation interfaces is described in Kenny *et al*. [12].

3.2 The Componentizing Approach for GAMESS

In general, the first step of componentizing a package is to create the SIDL interfaces. In our case, we need to extend the pre-defined chemistry interfaces in the ***cca-chem-generic*** package. Next, the implementation files of the specified programming languages (C, C++, f77, f90, python, or java) are generated based on those interfaces by using Babel, the language interoperability tool. The auto-generated implementation files are initially empty; they include only function headers and comments. Programmers need to insert implementation codes into each implementation file with the specified programming language; in our case C++.

To componentize a large-scale FORTAN 77 based code such as GAMESS, wrapper functions are necessary as a bridge between CCA interfaces and the native GAMESS code. Since there is no object-oriented design in the

ATOM	ATOMIC CHARGE	COORDINATES (BOHR) X	Y	Z
O	8.0	0.0000000000	0.0000000000	0.1239321808
H	1.0	1.4305200000	0.0000000000	-0.9834468192
H	1.0	-1.4305200000	0.0000000000	-0.9834468192

SHELL	TYPE	PRIMITIVE	EXPONENT	CONTRACTION COEFFICIENT(S)	
O					
1	S	1	130.7093214	0.154328967295	
1	S	2	23.8088661	0.535328142282	
1	S	3	6.4436083	0.444634542185	
2	L	4	5.0331513	-0.099967229187	0.155916274999
2	L	5	1.1695961	0.399512826089	0.607683718598
2	L	6	0.3803890	0.700115468880	0.391957393099
H					
3	S	7	3.4252509	0.154328967295	
3	S	8	0.6239137	0.535328142282	
3	S	9	0.1688554	0.444634542185	
H					
4	S	10	3.4252509	0.154328967295	
4	S	11	0.6239137	0.535328142282	
4	S	12	0.1688554	0.444634542185	

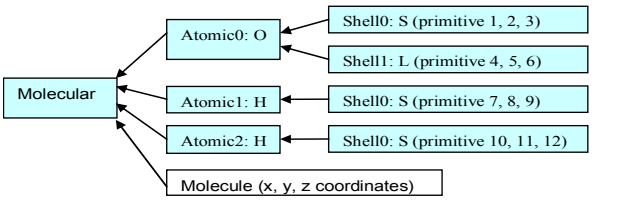

Figure 2. When using the water molecule and the "STO-3G" basis set as inputs, the information of molecule coordinates and the molecular basis sets in the GAMESS program is shown in the upper table. The upper block of the table shows the X, Y, Z coordinates of the water molecule. The bottom block of the table contains several columns. The information shown in the order from left to right is: the atomic symbols, the index of Gaussian shells, the Gaussian shell types, the primitive Gaussian shells, the exponents and contraction coefficients. Following each atom symbol is a block of Gaussian shells associated with it. The corresponding CCA integral components that store the same information are shown in the lower graph. The molecule coordinates are stored in a *Molecule* object (implements *MoleculeInterface*). The basis set information is stored in three *Atomic* (implements *AtomicInterface*) objects with the references to the corresponding *Shell* (implements *ShellInterface*) objects. A *Molecular* (implements *MolecularInterface*) object contains the references to the *Molecule* object and three *Atomic* objects.

GAMESS code, it is difficult for the implementation of GAMESS CCA components to utilize GAMESS subroutines directly. The use of wrapper functions divides GAMESS subroutines into smaller and less interleaving functions and therefore makes the componentization possible.

The Execution Sequence of GAMESS. To understand the design of GAMESS wrapper functions, we need to know the execution sequence of GAMESS and how the corresponding wrapper functions are created. First, the version information and the Distributed Data Interface (DDI) [13] are initialized. GAMESS uses DDI as its parallel communication mechanism, which mainly relies on TCP/IP sockets for communication, and can also utilize available communication libraries such as MPI or LAPI. The wrapper function *gamess_start* is created for wrapping the initialization steps of GAMESS. Second, molecule coordinates, basis sets and other user input options are read from an input file and the corresponding common blocks are initialized based on those inputs. The wrapper function

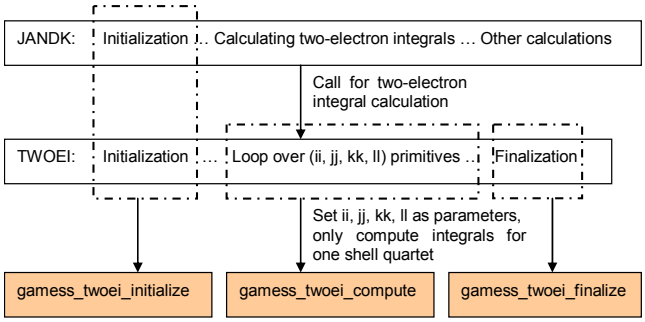

Figure 3. The componentization of two-electron integral calculations in GAMESS.

gamess_read_input is generated for this step. Next, depending on the type of computation, the execution follows different branches, such as energy, gradient, Hessian, optimize, or saddle point. Several wrapper functions are generated for those computations, such as *gamess_get_energy*, *gamess_get_gradient* and *gamess_get_hessian*. This list can be expanded by creating a wrapper function for each computation type. Finally, the control returns to the main program for finalizing computations and the communication layer. The wrapper function *gamess_end* is created for the finalization step.

GAMESS Integral Wrapper Functions. For ease of presentation, we omit details of data structures and functions used in integral computations, but list only the driver subroutines for one- (1-center) and two-electron (4-center) integral calculations in the GAMESS code and the corresponding wrapper functions in Table 1. The procedure of creating wrapper functions for one- and two-electron integral calculations are similar and we only present the approach of wrapping two-electron integral calculations.

The subroutine JANDK (Table 1) is the main driver for computing two-electron integrals. It first allocates memory for integral buffers and initializes integral calculations. TWOEI is then called for calculating two-electron integrals over four basis functions. However, the ***cca-chem-generic*** package defines the *compute* method of *IntegralEvaluator4Interface* to return integrals for only one shell quartet. In order to create a wrapper function that computes only one shell quartet while making minimum modification to the original GAMESS subroutine, the initialization, finalization, and computation steps are separated into three wrapper functions.

Combining the initialization steps in JANDK and TWOEI (Figure 3), a wrapper function is used for initializing two-electron integrals. The computation code in TWOEI is wrapped into a function that calculates integrals for one shell quartet with variables *(ii,jj,kk,ll)* in the loops as parameters. The wrapper functions are invoked by the *GAMESS.IntegralEvaluator4* (implements *IntegralEvaluator4Interface*) class. Finally, a wrapper function is created for finalization of two-electron integral calculations.

Table 1. The subroutines for computing integrals

Computation		Subroutine	Description
one-electron integral computation	GAMESS	ONEEI	the driver subroutine for the one-electron integral calculation
		HSANDT	calculate integrals over all shell doublets
	GAMESS Wrapper Functions	gamess_1e_initialize	initialize the one-electron integral calculation
		gamess_dblet_integral	compute integrals for a shell doublet
		gamess_1e_finalize	finalize the one-electron integral calculation
two-electron integral computation	GAMESS	JANDK	the driver subroutine for two-electron calculation
		TWOEI	calculate integrals over all shell quartets
	GAMESS Wrapper Functions	gamess_twoei_initialize	initialize the two-electron integral calculation
		gamess_twoei_compute	compute integrals for a shell quartet
		gamess_twoei_finalize	finalize the two-integral calculation

The reason we separate initialization steps from the computation steps is to reduce the overhead of the wrapper functions. The wrapper functions are designed to compute integrals for a shell doublet or a shell quartet, so they can be called $O(N^2)$ times for one-electron integral calculation and $O(N^4)$ times for two-electron integral calculation. Without separating the initialization step from computation steps, there would be a significant amount of overhead for computing integrals.

3.3 The Design of GAMESS CCA Integral Components

The implementation of GAMESS CCA integral components is straightforward as long as the integral wrapper functions have been constructed in a way that can be used by the CCA interfaces. The *GAMESS.IntegralEvaluatorFactory* component implements *IntegralEvaluatorFactoryInterface,* and is able to return the *GAMESS.IntegralEvaluator2* and *GAMESS.IntegralEvaluator4* classes for GAMESS integral computations (Figure 1). *The compute* method of the *GAMESS.IntegralEvaluator2* class invokes the wrapper function *gamess_dblet_integral* for computing a shell doublet and the *GAMESS.IntegralEvaluator4* class calls the wrapper function *gamess_twoei_compute* for calculating a shell quartet. Through the *IntegralEvaluatorFactoryInterface* Uses/Provides port, the functionality of the integral calculation can be shared between GAMESS and other chemistry packages.

The Structure of GAMESS CCA Components. GAMESS stores basis set and molecule coordinates in common blocks, through which the values required for

integral computation - the indexes of Gaussian shells, exponents, contraction coefficients, and Cartesian coordinates - are shared among different subroutines, and integral calculations can be performed. The GAMESS program initializes common blocks, memory, and communications by reading the user input options from an input file. The input file is read for many subroutines during a computation; without this file there is no way GAMESS can be initialized and perform computations. Even though the GAMESS components we developed are based on the interface for a "theoretically independent" component, the underlying wrapper function depends on the original design for initializing the GAMESS computations.

To deal with the common "input file" issue, our approach is to have the *GAMESS.ModelFactory* component (implements *ModelFactoryInterface*) create a disk file with the format of the GAMESS input file, based on the user options that are passed from the CCA parameters. This disk file will be passed to the GAMESS wrapper function *gamess_start* to initialize GAMESS computations. Figure 4 shows the dependencies among GAMESS CCA components, GAMESS wrapper functions and the GAMESS program. GAMESS CCA components are built on top of GAMESS wrapper functions, which wrap the functionalities of GAMESS into non-interleaving functions. To construct an application of GAMESS CCA integral computations, a *GAMESS.ModelFactory* component and a *GAMESS.IntegralEvaluatorFactory* component (implements *IntegralEvaluatorFactoryInterface*) are instantiated in a CCAFFEINE framework. This framework is middleware implementing a CCA model [14]. The *GAMESS.ModelFactory* component reads user input options from CCA parameters, creates a GAMESS input file on disk based on those input options and calls the wrapper function *gamess_start* to read the input file and initialize GAMESS common blocks and communications. The *GAMESS.ModelFactory* component also provides a *GAMESS.Model* class (implements *ModelInterface*) for calculating the energy, gradient and Hessian. After GAMESS computations are initialized successfully, the *GAMESS.IntegralEvaluatorFactory* component is able to provide the *GAMESS.IntegralEvaluator2* class (implements *IntegralEvaluator2Interface*) and the *GAMESS.IntegralEvaluator4* class (implements *IntegralEvaluator4Interface*) for integral computations.

The drawback of this approach is that GAMESS CCA applications are tightly coupled with the *GAMESS.ModelFactory* component to initialize GAMESS computations. While it is possible to change common block structures in the original GAMESS codes to get initialization information through a *ModelFactory* component from another package, we may be risking the robustness of GAMESS as information in these common blocks may also be used by many other computations (developed by developers a long time ago). Thus, for robustness reasons, we decided to use a less flexible approach.

3.4 MPQC Integral Components

MPQC components are derived in a straightforward manner from the class libraries underlying the MPQC package. For example, the *IntegralEvaluator4* CCA object simply wraps a class derived from *sc::TwoBodyInt*. On the client side, CCA integral factories are wrapped by the *sc::IntegralCCA* class and CCA evaluators, such as *IntegralEvaluator4*, are wrapped by the appropriate evaluator class, such as *sc::TwoBodyIntCCA*. Thus, MPQC has no code that directly uses CCA integral interfaces, with all function calls to CCA objects occurring through a wrapper object implementing an abstract interface. There are two integral evaluator factories available within MPQC, *IntV3EvaluatorFactory* and *CintsEvaluatorFactory*, providing access to the native *IntV3* integral package and the *Libint* package [15]. Details about the design of MPQC integral components are described in a previous publication [16].

3.5 NWChem Integral Components

As with the GAMESS code, the NWChem component software essentially consists of wrappers to access the capabilities of the NWChem integral API. Currently, the *NWChem.ModelFactory* needs to be created and initialized so that NWChem has the proper information concerning the basis sets and the molecular configuration. It is anticipated that this will change in the future. Once the Model Factory has created a Model, then NWChem has also initiated its other functionalities such as memory management (global array allocation), communication

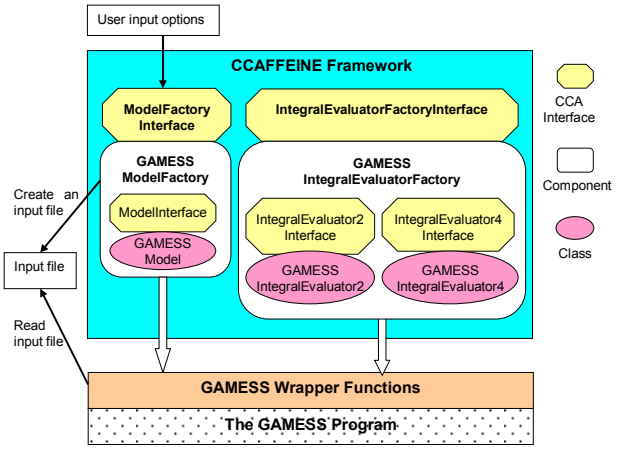

Figure 4. GAMESS CCA components are built on top of GAMESS wrapper functions.

protocols and run-time database management. This is currently essential for the integral components to function properly.

A significant portion of the CCA integral interface is similar to the NWChem API and there is a fairly direct one-to-one mapping. However, the *IntegralDescrInterface* is significantly different with no analog in NWChem, so the specifics of the types of computations that the API is to perform are kept in the components and translated to the appropriate API calls.

The integral termination is straightforward. However, the appropriate Model also needs to be terminated to end all of the NWChem processes. Since NWChem CCA components are currently being upgraded from working with the older version of Babel tools and the CCAFFEINE framework to working with the newest version of those packages, the integration of GAMESS and NWChem will be part of our future work.

3.6 Interoperability between GAMESS and MPQC

To test interoperability between packages, we pass the basis set information, the type of integrals, and molecule coordinates from a GAMESS *model factory* component to a MPQC *integral evaluator factory* component by invoking a *get_evaluator* method. For example, the SIDL definition for the *get_evaluator4* method of *IntegralEvaluatorFactoryInterface* is showed as follows:

```
/** Get a 4-center integral evaluator
    @param desc Integral set descriptor
    @return 4-center integral evaluator */
IntegralEvaluator4Interface get_evaluator4(
    in CompositeIntegralDescrInterface desc,
    in MolecularInterface bs1,
    in MolecularInterface bs2,
    in MolecularInterface bs3,
    in MolecularInterface bs4);
```

Using MPQC *integral evaluators* is expected to be as straightforward as using GAMESS *integral evaluators*, as long as everything is initialized properly. For example, our current testing is to pass a *GAMESS.GaussianBasisMolecular* object to the *MPQC.IntV3EvaluatorFactory* component through the *IntegralEvaluatorFactoryInterface* provides/uses connection. If the initialization in the *GAMESS.GaussianBasisMolecular* object is correct, then the *MPQC.IntV3EvaluatorFactory* component should be able to return an *integral evaluator* and do the same computation as a GAMESS *integral evaluator*.

The integration steps are as follows:

a) Instantiate a *GAMESS.ModelFactory* component and a *MPQC.IntV3EvaluatorFactory* component in a CCAFFEINE framework.
b) *GAMESS.ModelFactory* component reads user options through CCA parameters and initializes GAMESS common blocks, memory and parallel layers.
c) Create a *GAMESS.GaussianBasisMolecular* object and a *CompositeIntegralDescr* (implemented by the ***cca-chem-generic*** package) object.
d) Pass the *GAMESS.GaussianBasisMolecular* and *CompositeIntegralDescr* objects to the *MPQC.IntV3EvaluatorFactory* component and get the reference to a *MPQC.IntegralEvaluator4* object.
e) Invoke the *compute* method of the *MPQC.IntegralEvaluator4* object inside a four-level loop structure that computes integrals over all shell quartets.
f) Finalize and remove all objects and components.

The goal of this experiment is to test interoperability only. The results of an integral computation in each iteration are usually used by some other computation. With initial interoperability established, our future work will turn to componentizing GAMESS code that utilizes GAMESS/MPQC/NWChem integral components. The performance of GAMESS integral components and issues in the interoperability of GAMESS with MPQC integral components are discussed in Section 4.

4. Performance Evaluation

In this section we present only the performance of the two-electron integral computation since this computation takes significantly more CPU time than the one-electron integral computation does. We measure the wall-clock time for calculating all shell quartets of a molecule by using the GAMESS program, GAMESS wrapper functions, GAMESS CCA integral components and GAMESS & MPQC CCA components. First, we examine the performance overhead incurred by the design of the wrapper functions. This is done by invoking the *gamess_twoei_compute* wrapper function inside the four-level nested loop structure, and comparing the results with the time of the same computation by using the original GAMESS two-electron integral computations. Second, we examine the performance overhead caused by the CCAFFEINE framework when running the GAMESS CCA integral computations. This is done by evaluating the performance overhead of *GAMESS.IntegralEvaluator4* class, which in turn uses the wrapper functions for calculation. Finally, we present the performance data for the integration of GAMESS and MPQC.

The TAU performance tools are used for measuring the performance of two-electron integral computations in our testing. We insert TAU timers in both component-level methods and in GAMESS subroutines. The wall-clock time of looping over all shell quartets is used as the performance data and the time is measured in seconds.

Table 2. Molecule Basis Set Information

molecule	basis set	# of atoms	# of shells	# of basis functions	# of shell quartets
Ergosterol	6-31G*	73	204	523	2.18625E+08
Darvon	6-31G*	54	158	433	7.88956E+07
Luciferin	6-31G*	26	90	294	8.38656E+06
Nicotine	6-31G*	44	76	208	4.2822E+06

Table 3. Wall-clock Times (sec) for Two-electron Integral Computations

molecule	GAMESS	GAMESS Wrapper Functions	GAMESS CCA Components
Ergosterol	801.52	921.35	980.16
Darvon	361.47	422.72	445.15
Luciferin	63.39	74.11	77.06
Nicotine	22.93	26.71	28.50

The platform used for testing is a SMP cluster of 4 nodes, where each node has two dual-core 2.0GHZ Xeon "Woodcrest" CPUs and 8GB of RAM. The nodes are interconnected with both Gigabit Ethernet and DDR Infiniband. The operating system is Red Hat Enterprise Linux 4. Since both NWChem and MPQC parallelize the routines that call the integral computations, instead of parallelizing the integral computations themselves, we have decided to show only sequential performance data here. .

Test Cases. Four molecules are used as our test cases. Table 2 shows the names of the molecules, the basis set, the number of atoms, the number of shells, the number of basis functions, and the number of shell quartets. The test cases are listed in descending order according to the number of two electron integrals.

The Integral Screening in GAMESS Two-Electron Integral Computation. *Integral screening* is a technique to ignore calculating integrals which are estimated to have little or no contribution to the final results of the Fock matrix [10]. GAMESS by default uses integral screening techniques to screen out small integrals in the two-electron integral computation. In the design of CCA integral components, the integral screening has been separated from the integral computation, and is used as an independent option. Since the three chemistry packages use different screening techniques and default thresholds for small integrals, the number of non-zero two-electron integrals being calculated by each package is different from each other. We turn off the integral screening in every package when conducting interoperability testing to make sure the number of non-zero integrals computed by every integral component to be as close as possible.

4.1 Test GAMESS Integral Computations

In GAMESS, a native buffer (in memory), GHONDO, is allocated for storing 2e-integrals of one shell quartet. The results of GHONDO are either read and saved to a disk file, or used immediately, and the values of GHONDO are reset to zeros and used for storing 2e-integrals for another shell quartet in the next iteration. However, to componentize 2e-integral calculations for a shell quartet, the results should be stored in a buffer passed from a calling function (or an *integral evaluator4*). Instead of using GHONDO for storing the results of computing a shell quartet, we use the buffer passed to the wrapper function. The resulting integrals of each shell quartet can be accessed through the reference to the buffer by the end of each iteration and no disk I/O is needed for writing the results to a disk file.

To compare the performance of the original GAMESS subroutine and the wrapper function, we modified the original GAMESS code to ignore disk I/O after computing each shell quartet (to be compatible with our design in the wrapper function). The second column of Table 3 shows the performance data for computing 2e-integrals in GAMESS.

Test GAMESS Wrapper Integral Computation. The third column of Table 3 shows the performance for 2e-integral computation using wrapper functions. The overhead of the 2e-interal computation using the wrapper functions is about 17% of the 2e-integral computation with the original GAMESS code. In the original GAMESS code, statements that are inside the first, second or third-level of the four-level loop structure, now need to be executed for each shell quartet, about $O(N^4)$ times. If there is an overhead introduced by each single call to the compute method, the overall performance overhead can be significant.

Test GAMESS CCA Integral Computation. The goal of this experiment is to test the performance overhead of the CCAFFEINE framework. The GAMESS wrapper functions are used for implementing GAMESS CCA components. Thus, a buffer is passed from a *GAMESS.IntegralEvaluator4* object to the GAMESS wrapper functions for storing results of a shell quartet and the reference to the buffer is returned. The fourth column of Table 3 shows the running time of the 2e-integral calculation obtained using GAMESS CCA integral components. It shows that the performance overhead is relatively small, since all times are within 10% of the original running time.

4.2 The Integration of GAMESS & MPQC

Integral computations using CCA components from both MPQC and GAMESS are conducted through the process outlined in Section 3.5. In our testing, we produced the wall-clock time for computing two-electron integrals by using GAMESS CCA components, and GAMESS &

Table 4. Wallclock Times for Testing the Water Molecule with GAMESS and MPQC (sec)

basis set	GAMESS CCA Components	GAMESS & MPQC CCA Components
cc-pVQZ	3.63	3.65
aug-cc-pVQZ	16.07	15.96

MPQC components. Here we choose the water molecule with the cc-pVQZ and aug-cc-pVQZ basis sets. Table 4 shows that the discrepancy of the 2e-integral computation for the water molecule is very small between GAMESS CCA components and GAMESS & MPQC CCA components.

5. Issues in Integrating Packages

In the process of developing integral components, several issues affected our design of components, or delayed the progress of component development. We discuss these issues in this section.

5.1 Low-level Interoperability

Ideally if similar functions from different packages are componentized, complying with the same interface, we should be able to use these components interchangeably. However, if components are designed without substantial modifications to existing applications (e.g., using wrapper functions), the "plug-and-play" goal may be difficult to achieve.

The differences in the approaches to develop integral components provide a good example of the difficulties faced in interfacing low-level components in a "plug-and-play" fashion. For the MPQC integral component, the underlying software architecture is object-oriented and is more amenable to the encapsulation concepts of component architectures. For GAMESS, a package with over two decades of development history and developers scattered around the world, encapsulation into components may be error-prone in part because the subroutines to be encapsulated may be entangled with other subroutines developed by many scientists over a long period of time. To solve this problem, we chose to tightly couple the initialization processes of the original GAMESS program and the GAMESS CCA architecture, even though, in the standardized interfaces, it may be possible to use components from other packages for initialization.

5.2 Issues for Code Efficiency

The integral screening improves the efficiency of integral computations. In GAMESS, screening is a 'built-in' function that is integrated with integral computations and can be turned on or off by setting a flag in the input file. In MPQC, screening is not coupled with integral computations but rather may be performed by the caller of integral computations.

The interfaces for integral and other quantum chemistry computations are defined from a chemistry algorithm point of view. That is, the interfaces for data and methods performing electronic structure calculations are defined, but not for the procedures to improve code efficiency, such as using of screening. On one hand, we want to keep the interfaces as clean as possible, so they should include only data and methods that are essential to a computation; on the other hand, if a technique to improve code efficiency is widely used by every package, we may want to include this technique somewhere in the interface. How to seamlessly integrate via common interfaces computations and their efficient implementations, is a difficult design choice.

5.3 Version Control and Testing Procedure

Figure 5 shows the package dependence in this project. Besides three chemistry packages, we also use performance tools provided by TAU [17] to conduct component level performance evaluations. All packages, even compilers, are constantly updated with new versions. Whenever a certain package is updated, all the other packages may require rebuilding, and we have to conduct stability and compatibility testing all over again.

The process of rebuilding packages is time consuming; if errors occur during stability and compatibility testing, locating the source of the error is equally time-consuming. When some bugs are found in a new version of a package, we may have to roll back to an older stable version to continue the development process.

With the scope of quantum chemistry computations and the capabilities provided by the three packages, we expect more components will be developed. Exploring/developing a capable tool to minimize efforts in maintaining/testing packages is essential in a real-size project such as this one.

6. Conclusion

In this paper, we present our experience in developing CCA components based on a large-scale quantum

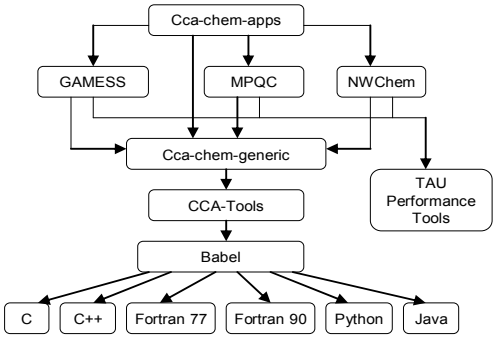

Figure 5. The package dependence for the CCA chemistry project.

chemistry package. The process of componentizing integral computation is delineated in detail and issues of interoperability are discussed. This will provide application scientists a perspective about the problems they may be facing when componentizing their packages to explore interoperation with other software. We are extending our experiments to integrate GAMESS and NWChem at the fine-grained level and also build a complete chemistry computation, such as calculating the energy, by using any two of the three chemistry packages through the CCA interfaces.

Based on our experience, community-agreed interfaces and data standards provide only the first step to componentization of a package; substantial efforts are needed to improve the usability of components, control versions of the underlying software, minimize overhead caused by extra layers of function calling, and standardize testing procedures to efficiently explore the errors in coupling many software packages. Componentizing a large-scale legacy software package is an especially challenging task. In other words, comprehensive scientific software engineering is essential in developing components that are truly shareable between scientific packages.

Acknowledgments

We thank Mike Schmidt from the department of Chemistry and Ames Laboratory, Iowa State University for the information on GAMESS and DDI. This work was supported by a SciDAC grant from the Department of Energy via the Ames Laboratory and Sandia National Laboratory.

References

[1] D. E. Bernholdt, B. A. Allan, R. Armstrong, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W.R. Elwasif, T. G. W. Epperly, M. Govindaraju, D. S. Katz, L. F. Diachin, J. A. Kohl, M. Krishnan, G. Kumfert, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. McInnes, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, S. Shende, T. L. Windus, and Zhou. S., "A Component Architecture for High-Performance Scientific Computing", international Journal of High Performance Computing Applications Vol. 20, No. 2, 163-202 (2006)

[2] CCA-Forum, the Common Component Architecture Forum. http://www.cca-forum.org

[3] Babel, http://www.llnl.gov/CASC/components/babel.html

[4] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Cordon, J.H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. J. Su, T. L. Windus, M. Dupuis, J. A. Montgomery, "General Atomic and Molecular Electronic Structure System", Journal of Computational Chemistry Vol. 14, Issue 11, 1347-1363 (1993)

[5] The Massively Parallel Quantum Chemistry Program (MPQC), Version 2.3.1, Curtis L. Janssen, Ida B. Nielsen, Matt L. Leininger, Edward F. Valeev, Edward T. Seidl, Sandia National Laboratories, Livermore, CA, USA, 2004.

[6] Aprà, E.; Windus, T.L.; Straatsma, T.P.; Bylaska, E.J.; de Jong, W.; Hirata, S.; Valiev, M.; Hackler, M.; Pollack, L.; Kowalski, K.; Harrison, R.; Dupuis, M.; Smith, D.M.A; Nieplocha, J.; Tipparaju V.; Krishnan, M.; Auer, A.A.; Brown, E.; Cisneros, G.; Fann, G.; Fruchtl, H.; Garza, J.; Hirao, K.; Kendall, R.; Nichols, J.; Tsemekhman, K.; Wolinski, K.; Anchell, J.; Bernholdt, D.; Borowski, P.; Clark, T.; Clerc, D.; Dachsel, H.; Deegan, M.; Dyall, K.; Elwood, D.; Glendening, E.; Gutowski, M.; Hess, A.; Jaffe, J.; Johnson, B.; Ju, J.; Kobayashi, R.; Kutteh, R.; Lin, Z.; Littlefield, R.; Long, X.; Meng, B.; Nakajima, T.; Niu, S.; Rosing, M.; Sandrone, G.; Stave, M.; Taylor, H.; Thomas, G.; van Lenthe, J.; Wong, A.; Zhang, Z.; "NWChem, A Computational Chemistry Package for Parallel Computers, Version 4.7" (2005), Pacific Northwest National Laboratory, Richland, Washington 99352-0999, USA.

[7] J. P. Kenny, S. J. Benson, Y. Alexeev, J. Sarich, C. L. Janssen, L. C. McInnes, M. Krishnan, J. Nieplocha, E. Jurrus, C. Fahlstrom and T. L. Windus, "Component-Based Integration of Chemistry and Optimization Software," Journal of Computational Chemistry, 24(14) 1717-1725 (2004).

[8] Fang Peng, Meng-Shiou Wu, Masha Sosonkina, Ricky A. Kendall, Michael W. Schmidt, Mark S. Gordon, Coupling GAMESS via Standardized Interfaces, HPC-GECO/Compframe, Paris, France, June 19-20 2006

[9] Boyana Norris, Jaideep Ray, Robert C. Armstrong, Lois C. McInnes, David E. Bernholdt, Wael R. Elwasif, Allen D. Malony, Sameer Shende: Computational Quality of Service for Scientific Components. CBSE 2004: 264-271

[10] Frank Jensen, Introduction to computational chemistry, John Wiley & Sons, ISBN-0471984256

[11] MPQC, The Massively Parallel Quantum Chemistry Program, http://www.mpqc.org

[12] Joseph P. Kenny, Curtis L. Janssen, Edward F. Valeev, and Theresa L. Windus,."Components for Integral Evaluation in Quantum Chemistry", Journal of Computational Chemistry, submitted.

[13] Ryan M. Olson, Michael W. Schmidt, Mark S. Gordon, Alistair P. Rendell, "Enabling the Efficient Use of SMP Clusters: The GAMESS/DDI Model", SC'03, Phoenix, Arizona, USA, November 15-21, 2003

[14] CCAFFEINE, a CCA Component Framework for Parallel Computing, http://www.cca-forum.org/ccafe/

[15] Libint library, http://www.chem.vt.edu/chem-dept/valeev/software/libint/libint.html

[16] 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)

[17] S. Shende and A. D. Malony, "The TAU Parallel Performance System," International Journal of High Performance Computing Applications, SAGE Publications, 20(2):287-331, Summer 2006