Scientific Programming 16 (2008) 287–296
DOI 10.3233/SPR-2008-0264
IOS Press

A component approach to collaborative scientific software development: Tools and techniques utilized by the Quantum Chemistry Science Application Partnership

Joseph P. Kenny [a,*], Curtis L. Janssen [a], Mark S. Gordon [b,c], Masha Sosonkina [c] and Theresa L. Windus [b]

[a] *Scalable Computing Research and Design, Sandia National Laboratories, Livermore, CA, USA*
[b] *Department of Chemistry, Iowa State University, Ames, IA, USA*
[c] *Scalable Computing Laboratory, Ames Laboratory/DOE, Iowa State University, Ames, IA, USA*

Abstract. Cutting-edge scientific computing software is complex, increasingly involving the coupling of multiple packages to combine advanced algorithms or simulations at multiple physical scales. Component-based software engineering (CBSE) has been advanced as a technique for managing this complexity, and complex component applications have been created in the quantum chemistry domain, as well as several other simulation areas, using the component model advocated by the Common Component Architecture (CCA) Forum. While programming models do indeed enable sound software engineering practices, the selection of programming model is just one building block in a comprehensive approach to large-scale collaborative development which must also address interface and data standardization, and language and package interoperability. We provide an overview of the development approach utilized within the Quantum Chemistry Science Application Partnership, identifying design challenges, describing the techniques which we have adopted to address these challenges and highlighting the advantages which the CCA approach offers for collaborative development.

Keywords: Electronic structure, integral, component, software development

1. Introduction

Fueled by the progress of simulation in many scientific domains and the ever increasing availability of hardware resources, cutting edge scientific computing increasingly involves the coupling of multiple packages. Within a community or scientific domain, this coupling is desirable to employ multiple advanced techniques, while packages from multiple domains are required to interact when simulations span multiple physical scales. For instance, the development and acquisition of petascale hardware is enabling and driving large-scale coupled-physics simulations in areas such as fusion simulation [1]. The software required for such studies is gaining complexity to the extent that large amounts of effort are required just to interface packages, requiring a team uniquely capable of understanding not only the multiple scientific domains involved but also the technical issues arising in such complicated software. A comprehensive approach to software engineering provides a framework for interaction between multiple development groups and reinforces good programming practices.

Component-based software engineering has been advanced as an important tool for managing the complexity of software [2]. Components are reusable software units that encapsulate useful functionality. As a distinction between other software abstractions, com-

*Corresponding author: Joseph P. Kenny, Scalable Computing Research and Design, Sandia National Laboratories, MS 9158, P.O. Box 969, Livermore, CA 94551-0969, USA. E-mail: jpkenny@sandia.gov.

ponents conform to a particular specification, allowing components from diverse sources to coexist in a common framework. Within high-performance scientific computing, the Common Component Architecture (CCA) Forum has advocated the adoption of components and developed both specifications and middleware tailored to the needs of scientific software [3]. CCA technology has made inroads into numerous simulation domains including optimization and linear algebra, accelerator design, fire and explosives modeling and climate simulation [4].

Within the chemistry domain, the authors are participating in the Quantum Chemistry Scientific Application Partnership (QCSAP) which is building out a CCA component toolkit which enables the integration of quantum chemistry software. This work has included the development of uniform interfaces which allow both the interchangeability of the high-level chemistry models supported by each package [5–7] and interoperation through the sharing of low-level capabilities [8–10]. While programming paradigms can go a long way to encourage and enable good development practices, we have found that complex software projects require careful design and an engineering approach which addresses both technical issues and the practicalities of human interaction, regardless of programming paradigm. Approaches to interface and data standards, and language and package interoperability must be adopted. In this work we provide an overview of our component development approach, identifying the design patterns which have proved useful in both our component and packaging architectures and highlighting the advantages which the CCA middleware and engineering approach offer for large-scale collaborative software development.

2. Uniform interfaces

As in other software disciplines, it is common for multiple scientific packages to exist which provide similar functionality but have various features, strengths and weaknesses. Enabling interchangeability and interoperability between various packages increases code flexibility, allowing applications which combine advanced capabilities or have optimum performance [8,11]. Within the quantum chemistry domain, high-level drivers, such as optimization solvers or dynamics packages, can utilize the broad range of quantum chemistry algorithms [5–7], and these algorithms themselves depend on a number of low-level capabilities that can be shared between packages [8–10]. An often-touted strength of component architectures is the ability to construct applications in a plug-and-play manner from a pool or toolkit of conforming components contributed by many sources. While component architectures do indeed enable this toolkit approach to application construction, all component developers must agree upon and implement uniform interfaces for this ideal to be realized. The adoption of uniform interfaces is the central challenge to large-scale collaboration in scientific software, and the approach used within the QCSAP builds upon tools and techniques developed by the CCA Forum to facilitate the transition from monolithic legacy codes to community-wide toolkits.

In our efforts within the quantum chemistry domain, we have taken care to focus our development efforts on the design of flexible uniform interfaces. Other notable efforts to standardize component interfaces include the Towards Optimal Petscale Simulations (TOPS) project [12] for linear/nonlinear solvers and the Interoperable Technologies for Advanced Petascale Simulations (ITAPS) project [13–15] for unstructured meshes. In order to provide uniform interfaces for a set of components, implementation-neutral interfaces must be formulated and agreed upon by participating development groups. This interface standardization is a challenging step, as it amounts to creating a reference design for a particular functionality, and this design must be sufficiently flexible to encompass existing implementations with limited overhead while maintaining simplicity and clarity as much as possible. Personal and political concerns are just as likely as technical issues to frustrate this process. It is essential that participants embrace the community aspect of the component development process and make compromises as necessary.

A focus on interface design is encouraged by the CCA middleware technology. While not required for a compliant CCA component, the CCA community has widely adopted the Babel [16,17] tool to handle language interoperability. Babel generates implementation stubs and glue code for the languages it supports (Fortran 77, Fortran 90, C, C++, Python and Java) based on interface definitions provided in a Scientific Interface Definition Language (SIDL) file. Babel implements a set of fundamental types and an object-oriented programming model for all supported languages. The requirement of SIDL files focuses developers on the task of interface definition and provides a language-neutral way to specify such interfaces. The

first concrete step towards adding a new capability to our chemistry toolkit is invariably the drafting of an interface file through discussions among the participating developers. Though this initial draft often evolves as implementation tasks reveal issues which were not initially appreciated, the production of an initial SIDL interface file gives a specification which can be discussed and modified as needed.

The inheritance feature provided by Babel's object-oriented model has proved essential in creating components with uniform interfaces. While uniform interfaces can typically encompass the standard routines required for computation, augmented interfaces have frequently proved necessary for initialization and configuration. In our chemistry architecture, the Babel classes which perform data storage and computation can have specialized interfaces which support implementation-specific initialization and configuration, but these interfaces all inherit from a common uniform interface. Since the specialized interface is only needed during initialization, the factory design pattern [18] has proved effective in encapsulating the implementation-specific code. Each class implementation has a corresponding factory component which performs the implementation-specific start up tasks. The factory interfaces are uniform interfaces that client codes use to supply calculation parameters. Given these parameters, each factory implementation performs the specialized tasks necessary to create and initialize the desired class, then returns a reference to the class to the requesting client. The client proceeds to utilize the class through the uniform base interface.

The *Chemistry.QC.ModelInterface* and its associated *Chemistry.QC.ModelFactoryInterface*, described in detail by Kenny et al. [5], are examples of uniform interfaces used within the CCA chemistry project that follow the factory design pattern. Figure 1 provides abbreviated listings of the SIDL files for these interfaces, showing the methods that are used for the most basic mode of operation. In our naming convention, uniform interface types end in *Interface*; further references to such types will exclude the *Chemistry.QC* namespace. In the quantum chemistry context, we define a model as a method which can assign an energy value to a particular electronic state in the presence of a rigid framework of nuclear centers. Thus, *ModelInterface* provides the *get_energy()*, *get_gradient()*, and *get_hessian()* members which compute the value of the energy and its first and second derivatives with respect to displacement of the nuclear centers. While the full *ModelFactoryInterface* supports more detailed configuration, the most basic usage requires only the location of the nuclear centers, name of the particular theory to be employed and a description of the basis functions used to describe the electronic state. The client code provides this information to the model factory using the *set_molecule()*, *set_theory()*, and *set_basis()* methods. The client simply requests a component implementing the *ModelFactoryInterface* from the framework, sets the desired calculation parameters using that interface and is returned a reference to a class implementing *ModelInterface* upon calling *get_model()*. The model implementation which the client receives is determined by the factory component configuration, which is determined at runtime by input that the user provides to the framework. The client code requires no specific knowledge of implementation details which are hidden behind the uniform factory interface.

Along with NWChem [19,20] and GAMESS-US [21], the MPQC [22–24] quantum chemistry package implements a model class that provides access to its available theoretical models through the *ModelInterface*. Each package has a different underlying architecture, however, requiring that each package's factory object perform different tasks to initialize their model object implementation. The MPQC implementation of *ModelInterface* exemplifies how such tasks are handled through specialization of the uniform interface. Figure 2 provides an abbreviated SIDL listing of the *MPQC.Model* and *MPQC.ModelFactory* classes, which implement the *ModelInterface* and *ModelFactoryInterface* uniform interfaces, respectively. Further references within this section to types not ending in *Interface* exist within the *MPQC* namespace. As with factories for other implementations, the *ModelFactory* component implements only the uniform interface, adding no additional methods. In contrast, the *Model* class requires MPQC-specific configuration information so that it can be initialized. Such initialization is handled through the addition of specialized methods to the uniform interface. One such method, *initialize_pasedkeyval()*, is shown in Fig. 2. Based on input provided through its uniform interface by client objects or the framework, *ModelFactory* constructs a specialized MPQC input string. This input string contains the necessary parameters (theory name, basis set and molecular framework) required to completely initialize the MPQC model implementation. When client code calls *get_model()* on the fac-

```
package Chemistry version 0.4.0 {

  package QC {

    interface ModelInterface {

      // Sets the molecule.
      void set_molecule( in Chemistry.MoleculeInterface molecule );

      // Returns the molecule.
      Chemistry.MoleculeInterface get_molecule();

      // Returns the energy.
      double get_energy();

      // Returns the Cartesian gradient.
      array<double,1> get_gradient();

      // Returns the Cartesian Hessian.
      array<double,2> get_hessian();
    }

    interface ModelFactoryInterface {

      // Set the theory name for Models created with get_model.
      void set_theory(in string theory);

      // Set the basis set name for Models created with get_model.
      void set_basis(in string basis);

      // Set the Molecule to use for Models created with get_model.
      void set_molecule(in MoleculeInterface molecule);

      // Returns a newly created Model.  Before get_model can be called,
      //  set_theory, set_basis, and set_molecule must be called.
      ModelInterface get_model();
    }
  }
}
```

Fig. 1. SIDL snippet defining *Chemistry.QC.ModelInterface* and *Chemistry.QC.ModelFactoryInterface*. The methods defined are used by client code to request a chemistry model from a model factory and obtain molecular energies, gradients and Hessians from the model.

tory component, the factory instantiates a *Model*, uses *initialize_parsedkeyval()* to initialize the model instance using the MPQC-specific input string and returns a model reference to the client which proceeds to perform computations using the uniform *ModelInterface*.

3. Implementation-specific interfaces

The broad adoption of uniform interfaces is clearly the preferred path to a capable and flexible scientific toolkit, realizing the plug-and-play idea put forth by component advocates. Due to a lack of funding, manpower or agreement on development approaches, creating uniform interfaces is not always possible. In cases where a proliferation of implementation-specific interfaces exists, the creation of adaptors is an effective approach to allow components to function in application environments which do not support their native interfaces.

The first major effort within the chemistry component project was to leverage the optimization solver component provided by the Toolkit for Advanced Op-

```
package MPQC version 0.2 {

  class Model implements-all Chemistry.QC.ModelInterface
  {
    // Initialize using MPQC keyval input string.
    void initialize_parsedkeyval(in string keyword, in string input);
  }

  class ModelFactory implements-all Chemistry.QC.ModelFactoryInterface,
    gov.cca.Component, gov.cca.Port
  { }
}
```

Fig. 2. SIDL snippet defining the *MPQC.Model* and *MPQC.ModelFactory* classes. These classes provide implementations of *Chemistry.QC.ModelInterface* and *Chemistry.QC.ModelFactoryInterface* which are defined in Fig. 1. The *MPQC.Model* class extends the generic *Chemistry.QC.ModelInterface* with the *initialize_parsedkeyval()* method which allows MPQC-specific configuration to occur.

timization (TAO) [25,26]. TAO offers a solver component implementing quasi-Newton optimization methods which are the workhorses of molecular structure optimization. We found that the line search implementation provided within the TAO solver did in many cases reduce the time to solution when compared with the existing capabilities within our quantum chemistry packages [5]. However, practical concerns about continued maintenance along with ease of installation and use make the implementation of additional solver components using the native capabilities within the quantum chemistry packages prudent.

While TAO does provide interfaces which could be adopted as standard uniform interfaces for optimization, these interfaces are not packaged separately. The packaging of the TAO interfaces along with the TAO code creates several disadvantages, discouraging us from adopting these as our native optimization interfaces. In order to use the interfaces, TAO and its dependencies, including PETSc [27], a major linear algebra package, need to be installed. Thus, in the case that only the solver components from the chemistry packages were desired, the significant additional effort of installing TAO and its dependencies would still be required. The continued functioning of our solver components would also depend upon the continued timely maintenance and support of TAO and PETSc, which is not a comfortable position given the realities of funding and the support required to keep up with software and hardware upgrades. Finally, there is simply the fact that by placing TAO later in our dependency chain we can expose fewer of our packages to any changes that might occur in the TAO package. For these reasons, we chose to define our own optimization interfaces within the *ChemistryOpt* namespace.

For reasons such as those that led us to create our own optimization interfaces within the chemistry project, some level of interface proliferation is bound to occur within the scientific component community. An approach to realizing polymorphism under these circumstances is the creation of interface adaptors. Interface adaptors are simple components which take two different interfaces for the same functionality and translate calls from one interface to the other. Figure 3 illustrates how the MPQC solver implementation can be combined with chemistry interface adaptors to function in application environments which expect a solver utilizing the TAO interfaces. In TAO application environments, the client uses the *Solver.OptimizationSolver* interface to call into the solver and the solver uses the *Optimize.OptimizationModel* interface to invoke the model. Within chemistry application environments, *ChemistryOpt.SolverInterface* and *ChemistryOpt.ModelInterface* are the corresponding interfaces. By placing interface translation components before and after the *MPQC.OptimizationSolver* component, the native solver interfaces can be adapted for use within a TAO application environment.

Using the adaptor approach, the number of adaptors to maintain does grow with the square of the number of communities for which interfaces need to be translated. Thus, this approach does create an n-squared problem. Nevertheless, in situations where the broad adoption of a single uniform interface is not practical, the creation and maintenance of adaptor components is a viable approach to realizing component polymorphism and maintaining flexibility within component toolkits.

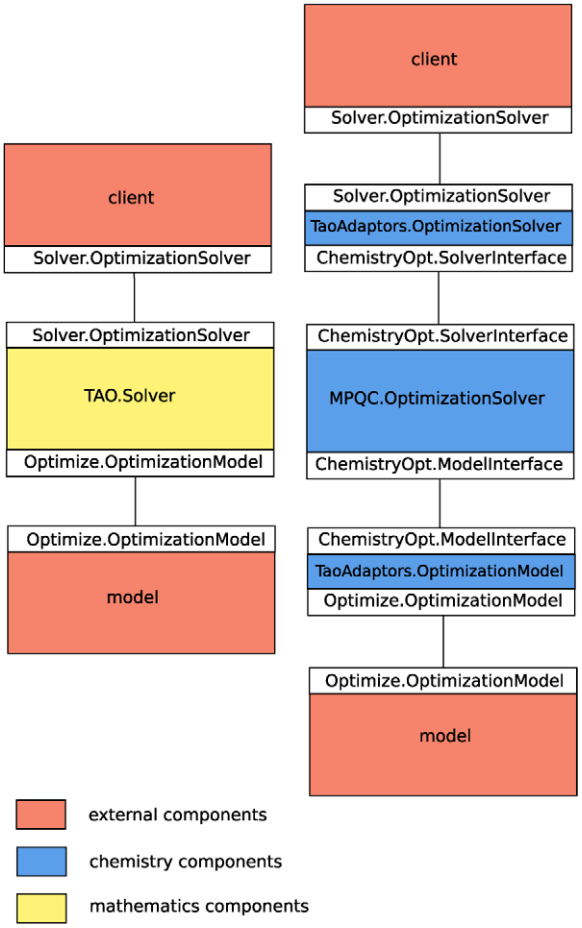

Fig. 3. The *TAO.Solver* component provides solver capability for client and model components which utilize the *Solver.OptimizationSolver* and *Optimize.OptimizationModel* interfaces defined by TAO (as depicted on the left). The *TaoAdaptor.OptimizationSolver* and *TaoAdaptor.OptimizationModel* adaptor components allow solvers implementing the *ChemistryOpt.SolverInterface* and *ChemistryOpt.ModelInterface* interfaces to function in environments utilizing the TAO interfaces (as depicted on the right).

4. Data layout standardization

In our experience, high-level interfaces, such as that provided by the *ModelInterface*, require enough work per method call through the interface that the overhead associated with adapting to a standard data layout is not significant. As toolkits gain functionality and low-level capabilities are shared between packages, implementation details are more often exposed and the costs of adapting to standard data layouts becomes more significant. Unfortunately, these increased costs are inherent to low-level integration. Component models and development techniques cannot eliminate the need to adapt between various implementation details which are exposed by low-level interfaces.

The evaluation of integrals over Gaussian basis functions for physical quantities such as electron repulsion and nuclear–electron attraction is a fundamental step for traditional quantum chemistry methods and was chosen as the first low-level capability to be shared within the QCSAP. For efficiency, integrals are typically evaluated in batches which include all possible angular functions for a given choice of radial functions. The number of batches which must be evaluated for even moderate size problems ranges easily into the millions. The ordering of integrals within these buffers is an arbitrary implementation detail for which codes desiring to share integral evaluators must adapt, and our work did indeed encounter this. While we have chosen to standardize on a convention which results in an ordering of xyz for p-type Cartesian angular functions, we wished to provide integrals to CCA models from the IntV3 integral package within MPQC which follows a convention resulting in p-type Cartesian functions ordered yzx. Figure 4 illustrates the reorder that the CCA interface layer must perform in a very simple case: two centers with p-shells on each center. With four center integrals with angular functions numbering greater than ten quite typical, the overhead involved in this reorder can be substantial.

Our work has demonstrated that the overhead due to integral reordering, as large as 12%, is comparable to the overhead due to the component interface layer. We expect that overheads of this magnitude are unavoidable with low-level integration. While a short-term view could result in pessimism regarding the utility of community-wide toolkits, the long-term view is that progress in science requires larger collaborations and the coupling of disparate packages, hence the development and adoption of community wide standards for data layouts is a critical step in the evolution of simulation capability. While there will be some large costs

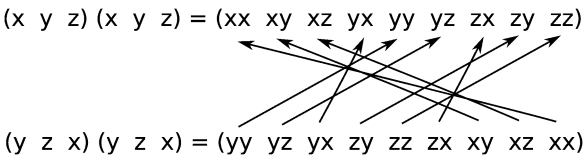

Fig. 4. The QCSAP project has adopted a convention in which p-type Cartesian angular functions are ordered xyz, while the IntV3 (MPQC) convention results in functions ordered yzx. The wrapper code which uses IntV3 to implement CCA integral evaluators is required to reorder each buffer to comply with the standard ordering. The relocations required to reorder a simple two center integral buffer with p-type shells on each center is illustrated.

to pay in performance for early toolkit applications, rapid development and adoption of standards will pay off as new development efforts choose to implement these standards natively.

5. Packaging

We have found the practical aspects of managing software dependencies and build systems to be a substantial challenge when moving to large-scale component efforts. The approach which we finally adopted is to create a *generic* package for the chemistry domain. The primary roles of the generic package are to hold the SIDL files which describe uniform interfaces through which conforming chemistry packages are expected to interact and build a library which contains the Babel glue code used to access those interfaces in the supported programming languages. The SIDL files in the generic package are thus the *de facto* standard interfaces for quantum chemistry component efforts. The glue code is required by all packages that wish to use these uniform interfaces, so providing one library upon which all other implementations can depend streamlines the build process. Finally, a range of utility classes which are useful to all chemistry packages are implemented in the generic package, promoting code reuse. These utility classes range from simple data containers, including a standard data container for the exchange of molecular data, to more complicated components, such as a management layer simplifying the use of multiple factory components that create molecular integral evaluators [5,10]. Each chemistry package which supports the component architecture provides server implementations which are matched up to the appropriate glue code by the framework and Babel runtime. These implementations leverage the existing utility classes provided by the generic package.

The generic package is a simple package with minimal dependencies and implementations for generally useful basic utilities. Maintaining this generic material within one of the chemistry packages would bring along the overhead of a lengthy and complicated build process, likely with numerous additional dependencies, for any developer wishing to implement the uniform interfaces. Our experience has shown that breaking out the uniform interfaces into a generic package is critical for broad adoption of standard interfaces. With several teams contributing changes to a generic package, careful versioning and documentation of interface changes is required, and we would like to adopt project tracking tools for this purpose.

Once all the core chemistry packages are built with component support, a final package which we term the *apps* package is built. This package supports combining the available components into usable applications. Sitting at the end of the toolchain, the apps package finds the available component packages and runs a verification suite to test the functionality of the components.

Figure 5 illustrates the dependency tree for the chemistry project. Adaptors for components from external packages, such as the Toolkit for Advanced Optimization (TAO) [25,26], are maintained in the apps package to keep the dependency tree manageable. It is important to note that, while dependency trees become quite large as the number of packages contributing to a toolkit grows, the CCA approach and our packaging approach do manage the complexity for both developers and users. The generic package provides not just interfaces and glue code, but also all needed CCA tools configuration information. Thus the generic package is one dependency that satisfies CCA requirements both from the chemistry and middleware perspective. From the user's perspective, there are many ways to access the capabilities of the QCSAP toolkit. The most powerful and advanced approach is to use CCA framework commands to directly construct component applications. A second approach which allows the chemistry developers to shield users from a great deal of the complexity of component application construction is the embedding of a CCA framework within an existing chemistry program. In this case the framework interaction is handled by the programmer, allowing the user to use the familiar input format native to the chemistry program. User friendly front-ends are another example of component applications that can be provided. As a proof of concept, we implemented a GUI which allows the user to select the QC package to optimize a given molecule and provides a visualization of the molecule throughout the optimization (see Fig. 6).

6. Conclusions

The efforts undertaken within the quantum chemistry domain have revealed several techniques to promote and manage uniform interfaces. Simple aspects such as code organization have proved extremely important in enabling interface adoption. The creation and maintenance of a simple generic package to pro-

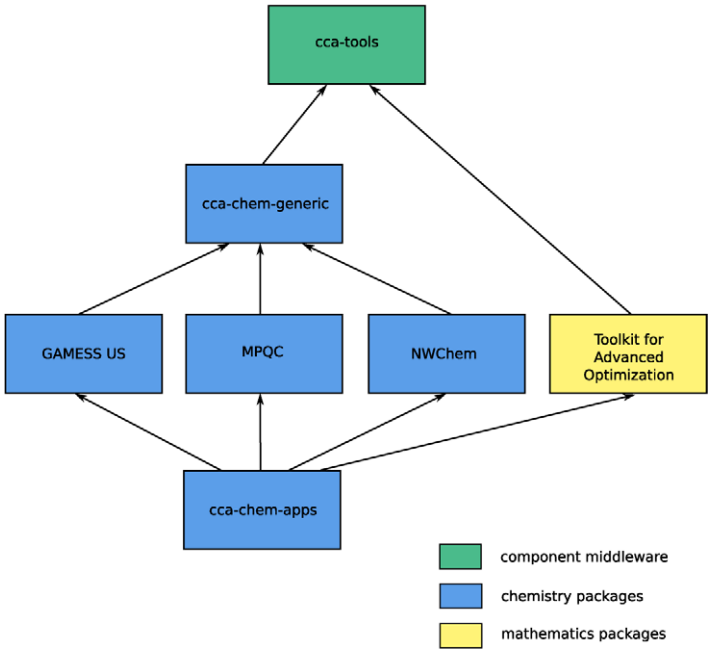

Fig. 5. Diagram of software package dependencies for the QCSAP project. Interface definitions, language interoperability glue code and widely useful component/class implementations are provided by the cca-chem-generic and cca-chem-apps packages. While the definition of optimization interfaces in both cca-chem-generic and the Toolkit for Advanced Optimization (TAO) does result in interface proliferation, this approach insulates all but one of the chemistry packages from changes to the TAO package.

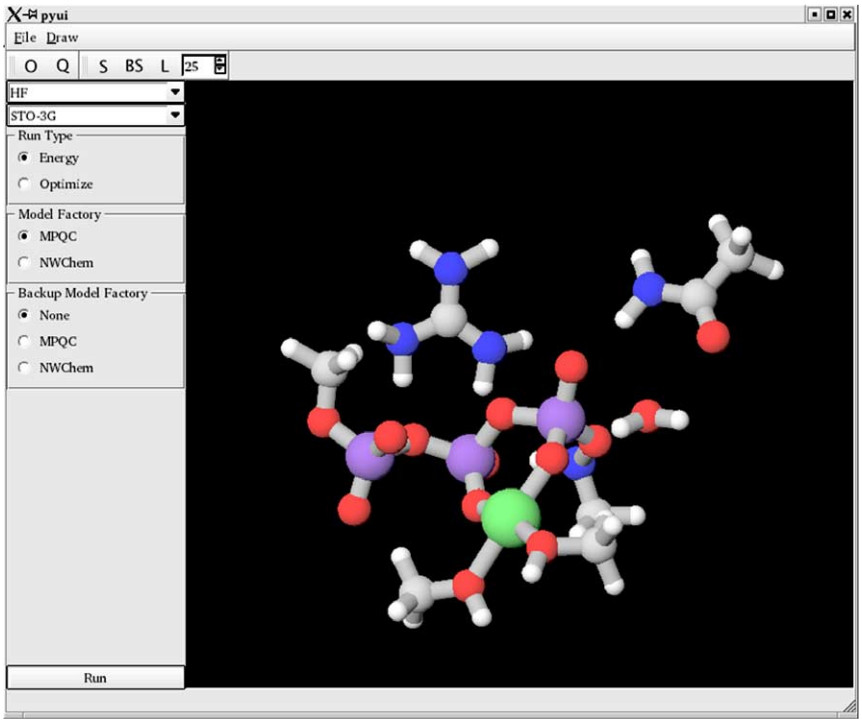

Fig. 6. Screenshot of a prototype graphical user interface developed by the QCSAP. The use of shared interfaces provides uniform access to the capabilities of several quantum chemistry packages, facilitating the development of highly capable toolkit front-ends.

vide interfaces and support code provides standardized interfaces and facilitates the efficient installation of a complex software environment. When the adoption of uniform interfaces is not practical due to a lack of funding, staffing or agreement in development approaches, the creation of interface adaptor components is a viable approach to managing interface proliferation. Interface adaptors allow components to function in application environments which do not support their native interfaces. While component-based software engineering does encourage and enable sound software engineering practices, a comprehensive development approach that addresses interface and data standardization, and language and package interoperability must be adopted to enable large-scale collaborative development.

Acknowledgements

This work has been supported in part by the US Department of Energy's *Scientific Discovery through Advanced Computing (SciDAC)* initiative [28], through the *Chemistry Framework using Common Component Architecture* scientific application project, of which Ames Laboratory and Sandia National Laboratories are participants. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the US Department of Energy under contract DE-AC04-94AL85000.

References

[1] W.R. Elwasif, D.E. Bernholdt, L.A. Berry and D.B. Batchelor, Component framework for coupled integrated fusion plasma simulation, in: *CompFrame'07: Proceedings of the 2007 Symposium on Component and Framework Technology in High-Performance and Scientific Computing*, ACM, New York, NY, USA, 2007.

[2] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, New York, NY, USA, 2002 (Chapter 4).

[3] 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, J.A. Kohl, M. Krishnan, G. Kumfert, J.W. Larson, 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 S. Zhou, A component architecture for high-performance scientific computing, *Intl. J. High-Perform. Comput. Appl.* **20** (2006), 163–202.

[4] L.C. McInnes, B.A. Allan, R. Armstrong, S.J. Benson, D.E. Bernholdt, T.L. Dahlgren, L.F. Diachin, M. Krishnan, J.A. Kohl, J.W. Larson, S. Lefantzi, J. Nieplocha, B. Norris, S.G. Parker, J. Ray and S. Zhou, Parallel PDE-based simulations using the common component architecture, in: *Solutions of PDEs on Parallel Computers*, Springer-Verlag, New York, 2005.

[5] 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, *J. Comput. Chem.* **25** (2004), 1717–1725.

[6] M. Krishnan, Y. Alexeev, T.L. Windus and J. Nieplocha, Multilevel parallelism in computational chemistry using common component archituecture and global arrays, in: *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, IEEE Computer Society, Washington, DC, USA, 2005.

[7] F. Peng, M.-S. Wu, M. Sosonkina, R.A. Kendall, M.W. Schmidt and M.S. Gordon, Coupling GAMESS via standardized interfaces, in: *HPC-GECO/Compframe*, Paris, France, 2006.

[8] 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, *J. Phys.: Conf. Ser.* **46** (2006), 220–228.

[9] F. Peng, M.-S. Wu, M. Sosonkina, T. Windus, J. Bentz, M. Gordon, J. Kenny and C. Janssen, Tackling component interoperability in quantum chemistry software, in: *Proceedings of the 2007 Symposium on Component and Framework Technology in High-Performance and Scientific Computing*, ACM, New York, NY, USA, 2007.

[10] J.P. Kenny, C.L. Janssen, T.L. Windus and E.F. Valeev, Components for integral evaluation in quantum chemistry, *J. Comput. Chem.* **29** (2008), 562–577.

[11] L. McInnes, J. Ray, R. Armstrong, T. Dahlgren, A. Malony, B. Norris, S. Shende, J. Kenny and J. Steensland, Computational quality of service for scientific CCA applications: Composition, substitution, and reconfiguration, Preprint ANL/MCS-P1326-0206, Argonne National Laboratory, 2006.

[12] Towards Optimal Petascale Simulations (TOPS) Center, TOPS homepage: http://www.scidac.gov/math/TOPS.html.

[13] D. Brown, L. Freitag and J. Glimm, Creating interoperable meshing and discretization software: The terascale simulation tools and technology center, Preprint UCRL-JC-147812, Lawrence Livermore National Laboratory, 2002.

[14] L. Diachin, A. Bauer, B. Fix, J. Kraftcheck, K. Jansen, X. Luo, M. Miller, C. Ollivier-Gooch, M.S. Shephard, T. Tautges and H. Trease, Interoperable mesh and geometry tools for advanced petascale simulations, *J. Phys.: Conf. Ser.* **78** (2007), 012015.

[15] Interoperable Technologies for Advanced Petascale Simulations (ITAPS) Center, ITAPS homepage: http://www.tstt-scidac.org/.

[16] Lawrence Livermore National Laboratory, Babel homepage: http://www.llnl.gov/CASC/components/babel.html.

[17] T. Dahlgren, T. Epperly and G. Kumfert, Babel User's Guide, version 0.9.0, 2004.

[18] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Creational patterns, in: *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, New York, NY, USA, 1998.

[19] R.A. Kendall, E. Apra, D.E. Bernholdt, E.J. Bylaska, M. Dupuis, G.I. Fann, R.J. Harrison, J.L. Ju, J.A. Nichols, J. Nieplocha, T.P. Straatsma, T.L. Windus and A.T. Wong, High performance computational chemistry: An overview of NWChem a distributed parallel application, *Comput. Phys. Commun.* **128** (2000), 260–270.

[20] Pacific Northwest National Laboratory, NWChem homepage: http://www.emsl.pnl.gov/docs/nwchem/.

[21] Gordon research group, Iowa State University, GAMESS homepage: http://www.msg.ameslab.gov/GAMESS/.

[22] Sandia National Laboratories, MPQC homepage: http://www.mpqc.org/.

[23] C.L. Janssen, I.M.B. Nielsen and M.E. Colvin, Parallel processing for *ab initio* quantum mechanical methods, in: *Encyclopedia of Computational Chemistry*, John Wiley & Sons, Chichester, UK, 1998.

[24] C. Janssen, E. Seidl and M. Colvin, Object-oriented implementation of parallel *ab initio* programs, in: *ACS Symposium Series, Parallel Computing in Computational Chemistry*, Vol. 592, American Chemical Society, Washington, DC, USA, 1995.

[25] S. Benson, L.C. McInnes, J. Moré and J. Sarich, TAO User's Manual, Technical Report ANL/MCS-TM-242, Mathematics and Computer Science Division, Argonne National Laboratory, 2004; see: http://www.mcs.anl.gov/tao.

[26] S.J. Benson, L.C. McInnes and J.J. Moré, A case study in the performance and scalability of optimization algorithms, *ACM Trans. Math. Software* **27** (2001), 361–376.

[27] S. Balay, K. Buschelman, W. Gropp, D. Kaushik, M. Knepley, L. McInnes, B.F. Smith and H. Zhang, PETSc User's Manual, Technical Report ANL-95/11, Revision 2.1.5, Argonne National Laboratory, 2003; see: http://www.mcs.anl.gov/petsc.

[28] US Department of Energy, SciDAC Initiative homepage: http://www.osti.gov/scidac/.