Software Architecture Issues in Scientific Component Development

Boyana Norris[1]

Mathematics and Computer Science Division, Argonne National Laboratory
9700 South Cass Ave., Argonne, IL 60439, U.S.A.
norris@mcs.anl.gov

Abstract. Commercial component-based software engineering practices, such as the CORBA component model, Enterprise JavaBeans, and COM, are well-established in the business computing community. These practices present an approach for managing the increasing complexity of scientific software development, which has motivated the Common Component Architecture (CCA), a component specification targeted at high-performance scientific application development. The CCA is an approach to component development that is minimal in terms of the complexity of component interface requirements and imposes a minimal performance penalty. While this lightweight specification has enabled the development of a number of high-performance scientific components in several domains, the software design process for developing component-based scientific codes is not yet well defined. This fact, coupled with the fact that component-based approaches are still new to the scientific community, may lead to an ad hoc design process, potentially resulting in code that is harder to maintain, extend, and test and may negatively affect performance. We explore some concepts and approaches based on widely accepted software architecture design principles and discuss their potential application in the development of high-performance scientific component applications. We particularly emphasize those principles and approaches that contribute to making CCA-based applications easier to design, implement, and maintain, as well as enabling dynamic adaptivity with the goal of maximizing performance.

1 Introduction

Component-based software engineering (CBSE) practices are well established in the business computing community [1–3]. Component approaches often form the heart of architectural software specifications. Garlan and Shaw [4] define a software architecture as a specification of the overall system structure, such as "gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements, physical distribution; composition of design elements; scaling and performance; and selection among design alternatives." This decade-old definition largely holds today.

Recently, the emergence of a component specification targeted at high-performance computing has led to initial experimentation with CBSE in scientific software. The Common Component Architecture Forum [5, 6] has produced a component architecture specification [7] that is semantically similar to other component models, with an emphasis on minimizing programming requirements and not imposing a significant performance penalty as a result of using components. The CCA specification is defined by using SIDL (Scientific Interface Definition Language), which provides language interoperability without sacrificing performance.

CCA employs the notion of *ports*, public interfaces that define points of interaction between components. There are two types of ports: *provides* and *uses*, used to specify functionality provided by a component, and to access functionality of other components that provide the matching port type. To be CCA-compliant, a component must implement the `gov.cca.Component` interface, which uses the method `setServices` to pass a reference to a framework services handle to the component. This handle subsequently is used for most framework interactions, such as registering *uses* and *provides* ports and obtaining and releasing port handles from the framework. The `setServices` method is normally invoked by the framework after the component has been instantiated. A CCA framework provides an implementation of the `Services` interface and performs component management, including dynamic library loading, component instantiation, and connection and disconnection of compatible ports.

In addition to the benefits of a component system for managing software complexity, the CCA offers new challenges and opportunities. In the remainder of this paper, we present some software architecture principles and approaches that build on the current CCA specification and help better define the component design and development process, as well as enable better dynamic adaptivity in scientific component applications. Going beyond the software architecture specification, we briefly discuss other important aspects of scientific component development, including code generation, implementation, compilation, and deployment.

2 Software Architecture Principles

In this section we examine software design approaches in the context of their applicability to high-performance scientific component development. One of the main distinctive features of the CCA is its minimal approach to component specification. This was partly motivated by the need to keep component overhead very low. Another motivation is to make the software design process more accessible to scientists who are concerned with the human overhead of adopting an approach new to the scientific computing community. The result is a working minimal-approach component model that has been used successfully in diverse applications [8–11]. The CCA specification incorporates a number of principles that Buschmann et al. [12] refer to as *architecture-enabling* principles, includ-

ing abstraction, encapsulation, information hiding, modularization, coupling and cohesion, and separation of interface and implementation.

While the minimal CCA approach does make scientific component development possible, it is not a complete design methodology, as is, for example, the Booch method [13]. Furthermore, other than the minimal requirements of the component interface and framework-component interactions, the actual component design is not dictated by any set of rules or formal specifications. This makes the CCA very general and flexible but imposes a great burden on the software developer, who in many cases has no previous component- or object-oriented design experience. We believe that a number of widely accepted software architecture principles, such as those summarized in [12, 14], can contribute greatly to all aspects of scientific application development—from interface specification to component quality-of-service support. These *enabling technologies* [12] can be used as guiding design principles for the scientific component developer. Moreover, they can enable the development of tools that automate many of the steps in the component development life cycle, leading to shorter development cycles and to applications that are both more robust and better performing.

Although the following software architecture principles are applicable to a wide range of domains, we focus on their impact on high-performance components, with an emphasis on adaptability. We briefly examine each approach and discuss an example context of its applicability in scientific component development.

Separation of concerns. Currently the CCA provides a basic specification for components and does not deal directly with designing ports and components so that different or unrelated responsibilities are separate from each other and that different roles played by a component in different contexts are independent and separate from each other within the component. Several domain-specific interface definition efforts are under way, such as interfaces for structured and unstructured mesh access and manipulation, linear algebra solvers, optimization, data redistribution, and performance monitoring. These common interfaces ideally would be designed to ensure a clear separation of concerns. In less general situations, however, this principle is often unknown or ignored in favor of quicker and smaller implementation. Yet clearly separating different or unrelated responsibilities is essential for achieving a flexible and reliable software architecture. For example, performance monitoring functionality is often directly integrated in the implementation of components, making it difficult or impossible to change the amount, type, and frequency of performance data gathered. A better design is to provide ports or components that deal specifically with performance monitoring, such as those described in [15], enabling the use of different or multiple monitor implementations without modifying the implementation of the client components, as well as generating performance monitoring ports automatically. Similar approaches can be used for other types of component monitoring, such as that needed for debugging. Although in some cases the separation of concerns is self-evident, in others it is not straightforward to arrive at a good design while maintaining a granularity that does not deteriorate the performance of the appli-

cation as a whole. For example, many scientific applications rely on a discretized representation of the problem domain, and it would seem like a good idea to extract the mesh management functionality into separate components. Depending on the needs of the application, however, accessing the mesh frequently through a fine-grained port interface may have prohibitive overhead. That is, not to say that mesh interfaces are doomed to bad performance; rather, care must be taken in the design of the interface, as well as the way in which it is used, in order to avoid loss of performance.

Separation of policy and implementation. A *policy* component deals with context-sensitive decisions, such as assembly of disjoint computations as a result of selecting parameter values. An *implementation* component deals only with the execution of a fully specified algorithm, without being responsible for making context-dependent decisions internally. This separation enables the implementation of multimethod solution methods, such as the composite and adaptive linear solvers described in [16, 17]. Initial implementations of our multimethod linear system solution methods were not in component form (rather, they were based directly on PETSc [18]), and after implementing a few adaptive strategies, adding new ones became a very complex and error-prone task. The reason was that, in order to have nonlinear solver context information in our heuristic implementation of the adaptive linear solution, we used (or rather, "abused") the PETSc nonlinear user-defined monitor routine, which is invoked automatically via a call-back mechanism at each nonlinear iteration. Without modifying PETSc itself, this was the best way both to have context information about the nonlinear solution, while monitoring the performance and controlling the choice of linear solvers. While one can define the implementation in a structured way, one is still limited by the fact that the monitor is a single function, under which all adaptive heuristics must be implemented. Recently we have reworked our implementation to use well-defined interfaces for performance data management and adaptive linear solution heuristics. This separation of policy (the adaptive heuristic, which selects linear solvers based on the context in the nonlinear solver) and implementation (the actual linear solution method used, such as GMRES) not only has lead to a simpler and cleaner design but has made it possible to add new adaptive heuristics with a negligible effort compared to the noncomponent version. Design experiences, such as this nonlinear PDE solution example, can lead to guidlines or "best practices" that can assist scientists in achieving separation of policy and implementation in different application domains.

Reflection. The *Reflection* architectural pattern enables the structure and behavior of software systems to be changed dynamically [12]. Although Reflection is not directly supported by the CCA specification, similar capabilities can be provided in some cases. For example, for CCA components implemented with SIDL, interface information is available in SIDL or XML format. Frameworks or automatically generated ports (which provide access to component metainformation) can be used to provide Reflection capabilities in component software. Reflection also can be used to discover interfaces provided by components, a par-

ticularly useful capability in adaptive algorithms where a selection of a suitable implementation must be made among several similar components. For example, in an adaptive linear system solution, several solution methods implement a common interface and can thus be substituted at runtime, but knowing more implementation-specific interface details may enable each linear solver instance to be tuned better by using application-specific knowledge. We note, however, that while reflection can be useful in cases such as those cited, it may negatively affect performance if used at a very fine granularity. As with other enabling architectural patterns, one must be careful to ensure that it is applied only at a coarse-enough granularity to minimize the impact on performance.

While the omission of formal specifications for these and other features makes the CCA approach general and flexible, it also increases the burden on the scientific programmer. While in theory it is possible to design and implement CCA components in a framework-independent fashion, in practice the component developer is responsible for ensuring that a component is written and built in such a way that it can be used in a single framework that is usually chosen a priori[1].

3 Implementation and Deployment

Implementation and deployment of CCA components are potentially complex and time-consuming tasks compared to the more traditional library development approaches. Therefore, CCA Forum participants have recently begun streamlining the component creation and build process. For example, CHASM [19] is being used to automate some of the steps required to create language-independent components from legacy codes or from scratch [20]. We have also recently begun automating the build process by generating most of the scripts needed to create a build system based on GNU tools, such as automake and autoconf [21, 22]. Currently we are integrating the component generation tools based on CHASM [19] and Babel [23] with the build automation capabilities. Other efforts are making the code generation, build automation, and deployment support tools easy to use, extensible, portable, and flexible. Until recently, little attention was given to the human overhead of component design and implementation. Many of the above-mentioned efforts aim to increase the *software developers'* efficiency. Most ongoing work focuses on the tasks of generating ports and components from existing codes, generating code from existing interfaces, compiling, and deploying components. At this point, however, little is available on the techniques for *designing* ports and components. We believe that the architecture-enabling principles mentioned in this paper, as well as others, can help guide the design of scientific applications. Furthermore, we hope that in the near future, when more and more component scientific applications go through this design and implementation process, domain-specific patterns will emerge that will lead to

[1] The CCA Forum has defined a number of interfaces for framework interoperability, so that while components are written in a way that may not be portable across frameworks, interaction between components executing in different frameworks is possible in some cases.

the same type of improvement in the software development approaches and efficiency that design and architectural patterns have led to in the business software development community.

Scientific applications may need to execute in environments with different capabilities and requirements. For example, in some cases it is possible and desirable to build components as dynamic libraries that are loaded by the framework at runtime. In other cases, the computing environment, such as a large parallel machine without a shared file system, makes the use of dynamic libraries onerous, and one statically linked executable for the assembled component application is more appropriate. Being able to support both dynamic and static linking in an application makes the build process complex and error-prone, thus necessitating automation. Furthermore, support of multiple deployment modes (e.g., source-based, RPM or other binary formats) is essential for debugging, distribution, and support of multiple hardware platforms. Many of these issues can be addressed by emerging CCA-based tools that generate the required component meta-information, configuration, build, and execution scripts.

4 QoS-Enabled Software Architecture

As more functionally equivalent component implementations become available, the task of selecting components and assembling them into an application with good overall performance becomes complex and possibly intractable manually. Furthermore, as computational requirements change during the application's execution, the initial selection of components may no longer ensure good overall performance. Some of the considerations that influence the choice of implementation are particular to parallel codes, for example, the scalability of a given algorithm. Others deal with the robustness of the solution method, or its convergence speed (if known) for a certain class of problems.

Recent work [24–26] on computational quality of service (QoS) for scientific components has defined a preliminary infrastructure for the support of performance-driven application assembly and adaptation. The CCA enables this infrastructure to augment the core specifications, and new tools that exploit the CCA ports mechanism for performance data gathering and manipulations are being developed [15]. Related work in the design and implementation of QoS support in component architectures includes component contracts [27], QoS aggregation models [28], and QoS-enabled distributed component computing [29, 30].

5 Conclusion

We have examined a number of current challenges in handling the entire life cycle of scientific component application development. We outlined some software architecture ideas that can facilitate the software design process of scientific applications. We discussed the need and ongoing work in automating different stages of component application development, including code generation

from language-independent interfaces, automatic generation of components from legacy code, and automating the component build process. We noted how some of the software architecture ideas can be used to support adaptivity in applications, enabling computational quality-of-service support for scientific component applications. Many of these ideas are in the early stages of formulation and implementation and will continue to be refined and implemented as command-line or graphical tools, components, or framework services.

Acknowledgments

Research at Argonne National Laboratory was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract W-31-109-ENG-38. We thank Matt Knepley, Lois McInnes, Paul Hovland, and Kate Keahey of Argonne National Laboratory for many of the ideas that led to this work, Sanjukta Bhowmick and Padma Raghavan of the Pennsylvania State University for their ongoing contributions, and the CCA Forum for enabling this collaborative effort.

References

1. Anonymous: CORBA component model. http://www.omg.org/technology/documents/formal/components.htm (2004)
2. Anonymous: Enterprise JavaBeans downloads and specifications. http://java.sun.com/products/ejb/docs.html (2004)
3. Box, D.: Essential COM. Addison-Wesley Pub. Co. (1997)
4. Garlan, D., Shaw, M.: An Introduction to Software Architecture. In: Advances in Software Engineering and Knowledge Engineering. World Scientific Publishing Company (1993)
5. Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes, L. C., Parker, S., Smolinski, B.: Toward a common component architecture for high-performance scientific computing. In: Proceedings of High Performance Distributed Computing. (1999) 115–124
6. Common Component Architecture Forum: CCA Forum website. `http://www.cca-forum.org` (2004)
7. CCA Forum: CCA specification. `http://cca-forum.org/specification/` (2003)
8. Norris, B., Balay, S., Benson, S., Freitag, L., Hovland, P., McInnes, L., Smith, B.: Parallel components for PDEs and optimization: Some issues and experiences. Parallel Computing **28 (12)** (2002) 1811–1831
9. Lefantzi, S., Ray, J.: A component-based scientific toolkit for reacting flows. In: Proceedings of the Second MIT Conference on Computational Fluid and Solid Mechanics, Boston, Mass., Elsevier Science (2003)
10. Benson, S., Krishnan, M., McInnes, L., Nieplocha, J., Sarich, J.: Using the GA and TAO toolkits for solving large-scale optimization problems on parallel computers. Technical Report ANL/MCS-P1084-0903, Argonne National Laboratory (2003)

11. Larson, J. W., Norris, B., Ong, E. T., Bernholdt, D. E., Drake, J. B., Elwasif, W .R., Ham, M. W., Rasmussen, C. E., Kumfert, G., Katz, D. S., Zhou, S., DeLuca, C., Collins, N. S.: Components, the Common Component Architecture, and the climate/weather/ocean community. In: 84th American Meteorological Society Annual Meeting, Seattle, Washington, American Meteorological Society (2004)
12. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. John Wiley & Sons Ltd. (1996)
13. Booch, G.: Unified method for object-oriented development Version 0.8. Rational Software Corporation (1995)
14. Medvidovic, N., Taylor, R. N.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering **26** (2000)
15. Shende, S., Malony, A. D., Rasmussen, C., Sottile, M.: A performance interface for component-based applications. In: Proceedings of International Workshop on Performance Modeling, Evaluation and Optimization, International Parallel and Distributed Processing Symposium (2003)
16. Bhowmick, S., Raghavan, P., McInnes, L., Norris, B.: Faster PDE-based simulations using robust composite linear solvers. Future Generation Computer Systems **20** (2004) 373–387
17. McInnes, L., Norris, B., Bhowmick, S., Raghavan, P.: Adaptive sparse linear solvers for implicit CFD using Newton-Krylov algorithms. In: Proceedings of the Second MIT Conference on Computational Fluid and Solid Mechanics, Massachusetts Institute of Technology, Boston, USA, June 17-20, 2003
18. Balay, S., Buschelman, K., Gropp, W., Kaushik, D., Knepley, M., McInnes, L., Smith, B. F., Zhang, H.: PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory (2003). `http://www.mcs.anl.gov/petsc`.
19. Rasmussen, C. E., Lindlan, K. A., Mohr, B., Striegnitz, J.: Chasm: Static analysis and automatic code generation for improved Fortran 90 and C++ interoperability. In: 2001 LACSI Symposium (2001)
20. Rasmussen, C. E., Sottile, M. J., Shende, S. S., Malony, A. D.: Bridging the language gap in scientific computing: the chasm approach. Technical Report LA-UR-03-3057, Advanced Computing Laboratory, Los Alamos National Laboratory (2003)
21. Bhowmick, S.: Private communication. Los Alamos National Laboratory (2004)
22. Wilde, T.: Private communication. Oak Ridge National Laboratory (2004)
23. Anonymous: Babel homepage. http://www.llnl.gov/CASC/components/babel.html (2004)
24. Trebon, N., Ray, J., Shende, S., Armstrong, R.C., Malony, A.: An approximate method for optimizing HPC component applications in the presence of multiple component implementations. Technical Report SAND2003-8760C, Sandia National Laboratories (2004) Also submitted to 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments, held during the 18th International Parallel and Distributed Computing Symposium, 2004, Santa Fe, NM, USA.
25. Hovland, P., Keahey, K., McInnes, L. C., Norris, B., Diachin, L. F., Raghavan, P.: A quality of service approach for high-performance numerical components. In: Proceedings of Workshop on QoS in Component-Based Software Engineering, Software Technologies Conference, Toulouse, France (2003)

26. Norris, B., Ray, J., Armstrong, R., McInnes, L.C., Bernholdt, D.E., Elwasif, W.R., Malony, A.D., Shende, S.: Computational quality of service for scientific components (2004). In: Proceedings of International Symposium on Component-Based Software Engineering (CBSE7), Edinburgh, Scotland.
27. Beugnard, A., Jézéquel, J.M., Plouzeau, N., Watkins, D.: Making components contract aware. IEEE Computer (1999) 38–45
28. Gu, X., Nahrstedt, K.: A scalable QoS-aware service aggregation model for peer-to-peer computing grids. In: Proceedings of High Performance Distributed Computing. (2002)
29. Loyall, J. P., Schantz, R. E., Zinky, J. A., Bakken, D. E.: Specifying and measuring quality of service in distributed object systems. In: Proceedings of the First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '98). (1998)
30. Raje, R., Bryant, B., Olson, A., Auguston, M., , Burt, C.: A quality-of-service-based framework for creating distributed heterogeneous software components. Concurrency Comput: Pract. Exper. **14** (2002) 1009–1034

U.S. Government License (not to be included in printed version).