Managing Complexity in Modern High End Scientific Computing through Component-Based Software Engineering

David E. Bernholdt
Oak Ridge National Laboratory
P. O. Box 2008, MS 6016, Oak Ridge, TN 37831-6016
bernholdtde@ornl.gov

Robert C. Armstrong, and Benjamin A. Allan
Sandia National Laboratories
7011 East Avenue, MS 9915, Livermore CA, 94550-0969
{*rob,baallan*}*@sandia.gov*

Abstract

The ever-increasing complexity of modern high-performance scientific simulation software presents a tremendous challenge to the development and use of this type of software, with significant impacts on productivity. Component-based software engineering is a means of addressing complexity that has been developed primarily in response to the needs of business and related software environments, but which has not yet had a significant impact on high-end computing. In this paper, we present the Common Component Architecture (CCA) as a component model designed to meet the special needs of high-performance scientific computing, focusing on how the CCA addresses issues of complexity. Unique among component architectures is the technique presented here by which a CCA component can act as a container to encapsulate and control other components without itself having to implement the functionality of a framework.

1. Introduction

Complexity of software is one of the greatest single challenges facing modern high performance scientific computing. It comes from several sources. As hardware manufacturers strive to provide ever faster systems, they become more complex, with deep non-uniform memory access hierarchies, CPU hierarchies (clusters of SMPs and similar models), widely varying architectures and capabilities for both interconnects and I/O systems. These "features" are almost always exposed to the programmer, and in order to achieve maximum performance, the programmer must take responsibility for tuning their code to each platform of interest. The second source of software complexity is the scientific problem being addressed. As computers have become more capable, and together with advances in software been able to deliver interesting and useful results through simulation, researchers demand more. So scientific simulation software expands to encompass larger problems, higher fidelity simulations, and the coupling of simulations across multiple time and length scales. In each case the complexity of the software must increase to answer the new challenges.

Studies have shown that the human mind is able to handle a limited amount of complexity [7, 13, 18], so that at some point the complexity of HPC software will outstrip the ability of programmers to deal with it and the pace of software development will slow. Assembling teams of programmers to create large-scale codes is a response to the size and complexity of the software and the breadth of knowledge required to successfully create it. However adding more workers, while necessary to deal with complexity, is not sufficient. The coordination required between workers imposes significant overheads that can limit the software scalability for the same reasons that Fred Brooks famously observed that "adding programmers to a late project only makes it later" [11].

Facing similar problems of software complexity, other communities, most notably the business and internet software communities, have invested heavily in *component-based software engineering* (CBSE) as a means to help address these issues. CBSE is based around the idea of software components, or units of programmatic functionality, that can be composed together to build an application. Components effectively break the complexity into people-sized chunks. Except to their developers, components are

treated as black boxes which interact with other components and the rest of the external environment only through well-defined interfaces. In this way, components encapsulate complexity with which users of the component need not concern themselves. Users create applications by composing components together in a "plug and play" fashion (which is very amenable to visual programming techniques) based on their interfaces. This provides a new level of abstraction for most software development, and thus a means of managing the complexity at a higher level.

The CBSE approach also provides a natural means to help control the complexity that arises due to multi-person interactions in team-developed software. Since interfaces are the key to component interoperability, the initial design of a component-based application can focus on the overall architecture and "componentization" of the problem and on defining the interfaces through which the components interact. With this task completed, individuals or small groups can then split off and focus on developing components conforming to the specifications without the need to interact with the creators of other components.

Component-based software engineering may seem like a natural approach to the creation of complex scientific software, and can be thought of as an extension of widely used approaches, such as the creation of software libraries, and object-oriented programming. But CBSE has not yet made significant inroads into HPC software, primarily because the "commodity" component models currently available, such as CORBA [14, 15, 23], COM/DCOM [19, 22], and Enterprise JavaBeans [20] were developed primarily for the business/internet software communities and do not address the needs of HPC scientific software very well [6]. Most commodity component environments have been designed primarily for distributed computing, and do not recognize or support the need for local performance and the use of tightly-coupled parallel computing as being more important than distributed computing. In scientific computing, it is common to have large codes which evolve over the course of many years, or even decades. Therefore, the ease with which "legacy" codebases can be incorporated into a component-based environment, and the cost of doing so, are also important considerations. Additional considerations include support for languages, data types, and computing platforms important to high-performance scientific computing.

The Common Component Architecture (CCA) [6, 12] was conceived in 1998 as a grass-roots effort to address the need of the scientific community for approaches to address the complexity of scientific software development and to facilitate and promote the creation of reusable, interoperable software for scientific high performance computing [3]. In this paper we describe features of the Common Component Architecture which simplfy the management of software complexity.

2. The Common Component Architecture

The Common Component Architecture is the nucleus of an extensive research and development program in the Dept. of Energy and academia. On the research side, the effort is focused on understanding how best to utilize and implement component-based software engineering practices in the high-performance scientific computing area. In addition to the definition of the CCA specification itself, the development effort is aimed at creating practical reference implementations conforming to the specification, helping scientific software developers use them to create CCA-compliant software, and, ultimately, at creating a rich "marketplace" of scientific components from which new component-based applications will be built. Space constraints require that we limit our presentation here to those aspects of the CCA which bear directly on dealing with complexity: a description of the basic elements of the CCA's component model, and the mechanism by which components are created, formed into applications, and executed. However, a comprehensive overview will be published soon [10] and tutorials are already available [1].

The specification of the Common Component Architecture defines the rights, responsibilities and the relationships between the various elements of the model. Briefly, these are as follows:

- *Components* are units of software functionality that can be composed together to form applications. Components encapsulate much of the complexity of the software inside a black box and expose only well-defined interfaces to other components.

- *Ports* are interfaces through which components interact. Specifically, CCA ports provide procedural interfaces that can be thought of as a class or an interface in object-oriented languages, or a collection of subroutines, or a module in a language such as Fortran 90. Components may provide ports, meaning they implement the functionality expressed in the port (called *provides ports*), or they may use ports, meaning they make calls on that port provided by another component (called *uses ports*).

- The *framework* holds CCA components as they are assembled into applications and executed. The framework is responsible for connecting uses and provides ports without exposing the components' implementation details. It also provides a small set of standard services, defined by the CCA specification, which are available to all components. The `BuilderService` and `AbstractFramework` ports are two of these

standard services which are both central and novel with respect to the way the CCA deals with complexity.

The CCA employs a minimalist design philosophy to simplify the task of incorporating pre-existing HPC software into the CCA environment. CCA components interact with the CCA framework via the `Services` interface, which provides the means for components to register the ports they provide and use (`addProvidesPort()`, `registerUsesPort()`), and to obtain "handles" to ports so that they can be used (`getPort()`). This makes it possible for the framework to effectively and efficiently mediate component connections. To be "CCA compliant", components are required to implement the `gov.cca.Component` class, which includes just one method: `setServices()`. This method is invoked by the framework immediately after the component is instantiated, passing in a CCA Services object (later referred to as `svc`). The primary purpose of `setServices()` is for the component to tell the framework what ports it provides and uses.

The uses/provides design pattern for ports and the framework's role in mediating the connection of ports is also important in the CCA's ability to transparently support both local high-performance and distributed computing models. Prior to actually invoking a method on another port, the component uses the `svc.getPort()` method to obtain a handle to the port. In the distributed computing case, the handle would be a pointer to a local proxy for the provides port created by the framework, and the framework is responsible for conveying the remote method invocations to the actual provider, including marshaling and unmarshaling arguments. In the local high-performance (also referred to as "direct connect" or "in-process") case, the framework typically loads components into separate *namespaces* within the address space of a single process, so in this case the handle can be a pointer to the virtual function dispatch table for the providing port. In this case the method invocations take place directly without intervention by the framework and without CCA-imposed overheads beyond the virtualization of the function call (common in object-oriented languages anyway). Since the caller and callee share the same address space, all arguments are commonly passed by reference without the loss of performance indirection entails. Measurements show that the CCA-imposed overhead on calls between components in the direct connect case is minimal, and does not impact performance relative to traditional (non-component) programs [9, 21]. In some cases, the Bable language interoperability tool [8] may need to translate datatypes between languages, but for most scientific computing these overheads can be avoided.

In the high-performance *parallel* context, the CCA's model is that of many of the local high-performance "in-process" component assemblies running in parallel across many processors. Components in each process operate via the usual CCA mechanisms, while the parallel instances of a given component can utilize whatever parallel communications model they prefer, without any CCA-imposed overheads. Both single-component/multiple-data and multiple-component/multiple data paradigms are supported, analogous to SPMD and MPMD programs without any CCA-imposed performance overheads [21].

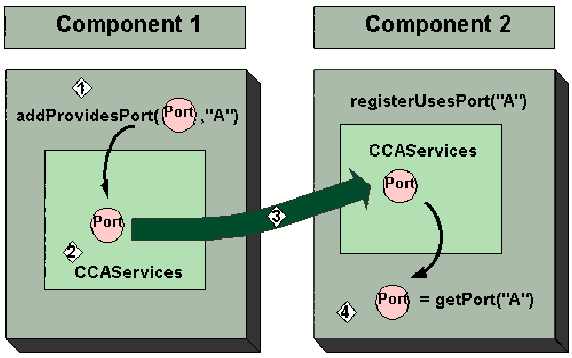

Figure 1. A schematic representation of the sequence of interactions between the component and framework via the CCA Services object that allow ports to be connected and used.

Figure 1 illustrates more specifically the sequence of interactions between the component and framework via the CCA Services object that allow ports to be connected and used. In step 1, Component 1 calls `svc.addProvidesPort()` (and Component 2 calls `svc.registerUsesPort()`) to express their intent. The CCA Services object caches the information about the port it got from `addProvidesPort()` (step 2). In the third step, the framework connects the uses port to the provides port, and the framework copies information about the provides port over to the user's (component 2's) CCA Services object. Finally, when Component 2 wants to invoke a method on the port provided by Component 1, it issues a `svc.getPort()` call to obtain a handle for the port. Not shown in the diagram is the `svc.releasePort()` call, informs the framework that the caller is (temporarily) done using the port. A port may be used only *after* a `getPort()` call is made for it, and before its companion `releasePort()` call; `getPort()` and `releasePort()` can be used repeatedly throughout the body of the component. This is considered better CCA programming practice than acquiring handles to all relevant ports once at the beginning of the component execution and releasing them only at the end, because it allows the use of a more dynamic component programming model, through the `BuilderService` port.

In "normal" use of the CCA model, steps 1–3 would take place during the "assembly" phase of the applications. Specifically, steps 1 and 2 would take place with the component's `setServices()`, invoked by the framework when the component is instantiated, and step 3 would take place as the component instructs the framework how to connect the uses and provides ports for the application. Step 4 would take place during execution of the component's code. Finally, when not within a `getPort()`/`releasePort()` block, connections between uses and provides ports may be broken, and components may be destroyed.

In general, components cannot use ports on other components during the assembly phase (i.e. within the component's `setServices()` routine) because there is no guarantee that the components providing those ports have been instantiated and connected to this component's "uses port". There is one exception, however. As a reuse of concepts, the CCA also casts framework services as ports, and such services *are* available to components as soon as they have been instantiated.

While this explanation has portrayed the phases of the lifecycle as "collective", with the entire application being assembled, executed, and then disassembled, this is not necessarily the case. Through the `BuilderService` framework service port, applications can have extremely dynamic behavior. The motivating example for the development of `BuilderService` was the desire to be able to swap out one numerical solver for another during a simulation because, for example, the solution might be moving into a region where another solver would provide better performance or numerical quality [16]. `BuilderService`, together with the `AbstractFramework` service also allow hierarchies of components to be created, encapsulating many components and treating them as one.

3. Application Complexity in the CCA

While the CCA does a very good job of encapsulating the complexity of thousands of lines of source code into black-box components, the model, as described in Section 2, has only one level. Modern HPC scientific applications often grow extremely large and involve the coupling of simulations at different time or length-scales. Eventually even componentized versions of such applications become too complex for software developers and users to deal with all at once.

As an example, consider the study of a reaction-diffusion simulation under varying numerical and geometric parameters, where the user may be interested in performance, convergence, and efficiency. Figure 2 shows the CCA "wiring diagram" for a modestly complex "production quality" application of this type, developed by Jaideep Ray, Sophia Lefantzi, and their co-workers in the Center for Reacting Flow Simulation lead by Sandia National Laboratory [2, 17].

This figure, derived from a screen capture of the visual programming interface currently available with the CCaffeine CCA framework [4, 5], shows the numerous components as dark boxes decorated with smaller boxes representing the provides ports (left side of each component) and uses ports (right side). Lines show connections between uses and provides ports. The component layout is very cluttered and quickly fills most of the screen. As is typical in component-based applications, multiple components are used to implement the various high-level elements of the application. In this case, three components together provide the reaction kinetics functionality (heavy oval) and a second group of four components that handle the diffusion equation (heavy rectangle). These components dominate the work area, but are of little interest to the planned study, because they will not be changed in any way.

This example makes the visual case for the need to be able to group components to further hide complexity. In this case, our purposes would be well served if we could group the three chemistry components together into one a single `ChemSolver`, and the four diffusion components into a single `DiffusionIntegrator` component. This would simplify the visual programming picture, making it easier to work with the remaining components, which are of interest in the planned study. It may also be of interest to export these groupings as components in their own right, available for use in other applications. Flexibility of the mechanism is important too: the target of the next study could be a comparison of the numerical and performance characteristics of the `CvodeSolver` against other equivalent solvers, looking for a possible replacement. In this case, we would want to see all of the structure for the chemistry part of the application, but could black-box other component groupings.

4. Reusing Component Concepts for Aggregation and Scalability

For a peer object model like CCA, there is really only one option to deal with the need to provide multiple levels of encapsulation: a peer container object for networks of components. Although there are notable exceptions (e.g. Visual Basic™), it was considered a best practice to make each container itself a component and therefore achieve a self-similar answer to component aggregation.

Because the `BuilderService` port exports framework functionality to the user, it serves two vital functions that normally are under framework control: containment and composition. Containment allows an entire component composition (network of connected components) to be black-boxed as a single component. Dynamic composition allows changes in the way a component network is

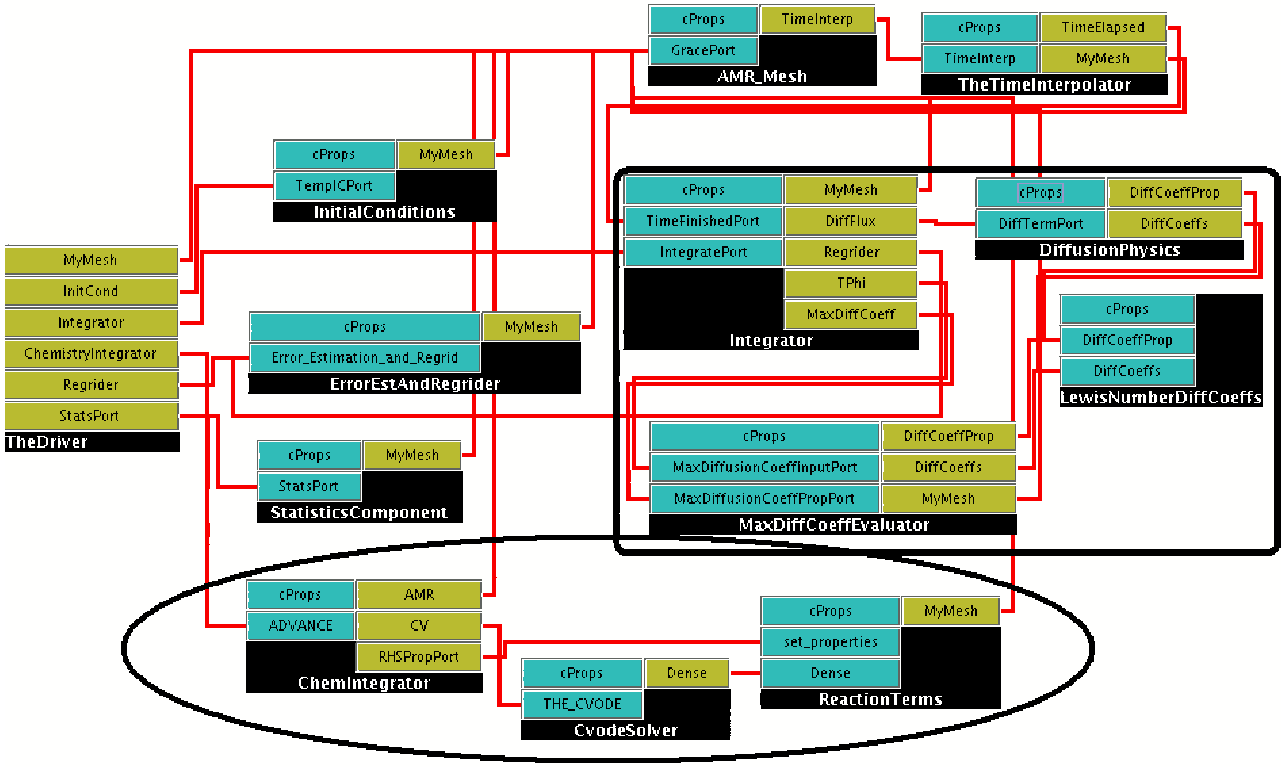

Figure 2. Assembly diagram of a reaction-diffusion simulation using an implicit/explicit integration scheme on an adaptively refined grid. The heavy lines highlight the components related to the reaction kinetics (oval) and the diffusion equation (rectangle).

connected at any time during the execution of the program.

4.1. *BuilderService*: Component Containers in a High Performance Setting

The `BuilderService` interface allows the high-performance computational scientist the ability to take on the role of the framework programmer. `BuilderService` is a standard framework service port and the user requests this interface through the usual CCA mechanism of `svc.registerUsesPort()` and `svc.getPort()`. The entire interface for BuilderService can be found on the web at `http://www.cca-forum.org/specification/`. Figures 3–6 show an entire scenario for containing a more complicated component network within a controlling component that uses `BuilderService`.

The idealized scenario of these figures is to encapsulate a two-component network, but proxy an unconnected provides port and an unconnected uses port on the outside of the container making them available for connections by a user. In the figures, a component called Container Component is located and instantiated in the usual way of Section 2. Container Component requests a `BuilderService` port (Fig. 3). Because `BuilderService` is a CCA service port provided by the framework, it can be retrieved immediately, during its initial `setServices()` call. During the same call, two components are instantiated, and then connected together through use of the `BuilderService` interface (Fig. 4). Next the Container Component connects the component network to itself by exporting the same type of ports on itself (Fig. 5) and connecting them to the contained network. Finally, the single Container Component presents the two proxied ports encapsulating, in this case a two component network (Fig. 6). The Container Component here does not have any functionality other than as a program to create an interior encapsulated network of components, re-exporting provides ports and proxying uses ports that are left unconnected.

It is worth noting that once all of the connections to the containing component are made, there is no further involvement by the container in the execution of the program. All of the encapsulated components are dealt with directly. This means that taking advantage of the CCA containment mechanism inflicts no performance penalty on the user's application.

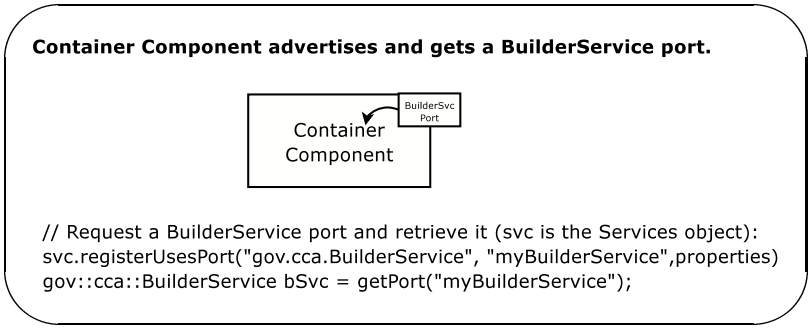

Figure 3. CCA Containment Mechanism Using `BuilderService`: Step 1: Create the container component.

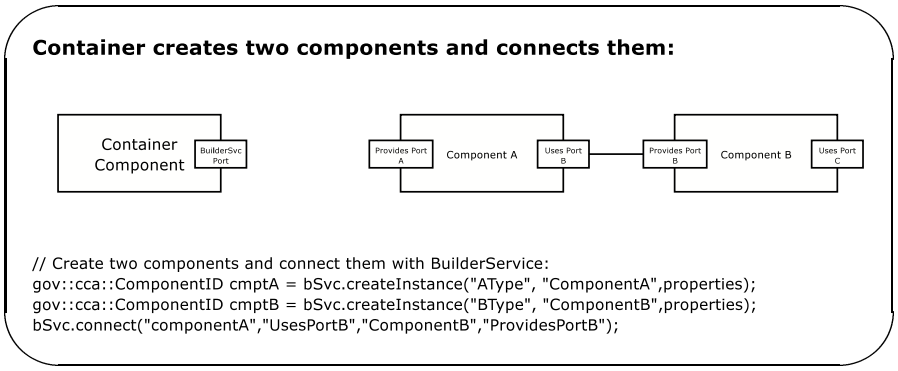

Figure 4. CCA Containment Mechanism Using `BuilderService`: Step 2: Container component composes a network of components.

4.2. Using CCA from Main: AbstractFramework

Component-oriented programming implies that there is an overarching framework that instantiates, manipulates, and destroys components and otherwise manages components on the behalf of the user. The downside is that the user is unable to write or control the `main()` program. In most cases a well-written CCA-compliant framework will cover what 90% of the users would like to do. However since high performance computing involves ever more sophisticated hardware, and hence runtime environments, some setup may be needed ahead of the framework to prepare it for queuing systems, message passing layers, or other nonstandard facilities that could not be anticipated by the framework developers. The CCA's answer to this requirement is another interface called `AbstractFramework`. This interface is not a `gov.cca.Port` but one that allows an instantiation of a CCA framework from a library. Beyond creation and destruction there is only one method on the interface: `getServices()` which returns a `Services` object identical to the one received in the `setServices` call by a normal component. The `getServices()` call effectively creates an image of the main program inside the framework allowing `addProvidesPort()` and `registerUsesPort()` calls from the main program the same as any other component. From then on the `BuilderService` interface can be requested and the process can proceed as before. Listing 1 shows an example of this in Python.

`BuilderService` and `AbstractFramework` interfaces are considered by the CCA working group to be "advanced" behavior, and would probably only be undertaken by a user that is already well versed in CCA component semantics and behavior. In essence the user is taking over the role usually occupied by a CCA compliant framework. It is important that the user of the `BuilderService` containment be cognizant of what CCA components are entitled to expect and to respect the component life cycle

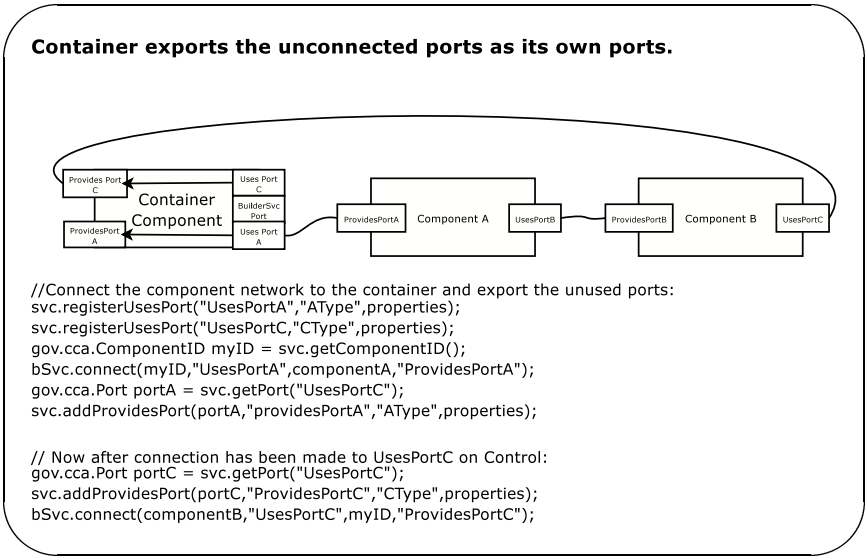

Figure 5. CCA Containment Mechanism Using `BuilderService`: Step 3: Container component connects itself to unconnected ports.

that the component writers depend on.

Mentioned previously, another important function of `BuilderService` is the automation of the componentized programs at the component level. For example, an equation solver component used in the solution of a PDE might work fastest with an **LU** preconditioner for some number of time steps, but later in the calculation might require a multigrid method. A component that monitors the convergence behavior (possibly the condition number) could disconnect the **LU** component and plug in the multigrid method as the need arises. `BuilderService` allows high-performance components to be programmed dynamically, as any other object in the calculation [16].

4.3. The Simplified Reaction-Diffusion Application

The mechanisms described in this section can be applied to our reaction-diffusion simulation example, producing the result shown in Figure 7. The core chemistry and physics of the problem are now encapsulated within the black-box `ChemSolver` and `DiffusionIntegrator` components. The `ErrEstAndRegrid` and `TimeInterpolator` components are readily accessible, and there is enough screen real estate available to easily manipulate the components of interest as needed.

5. Conclusions

A certain amount of complexity is unavoidable in high-performance scientific computing due to the complexity of the problems being solved. The Common Component Architecture is design specifically to meet the needs of this community, including the need to better manage complexity.

Because CCA's target developers are computational scientists who wish to focus not on software development, but on their scientific simulations, the tools and concepts of the CCA must be both simple to grasp and scalable to the problems of interest to the computational scientists. Reuse of concepts is an important means for the CCA to achieve this necessary simplicity – in other words, to reduce the complexity inherent in the CCA itself. An example is the use of the uses/provides concept for ports to transparently enable both high-performance local component assembly and distributed computing. In the parallel computing case, the CCA's approach allows the programmer to reuse the tools and techniques with which they are most comfortable for parallel programming, rather than imposing a new model or tools on them.

The CCA also deals with software complexity directly. At the first level, components provide black-box encapsulation of complex pieces of source code so that the user of the component (as opposed to its developer) need not be concerned about its internals. Through the `BuilderSer-`

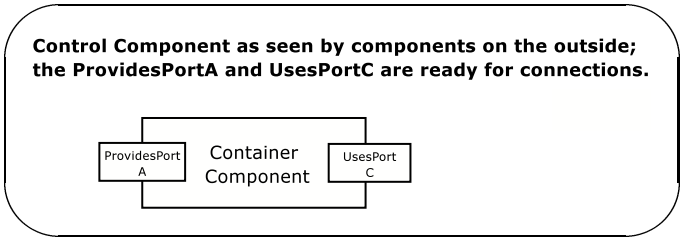

Figure 6. CCA Containment Mechanism Using `BuilderService`: Step 4: Finally, only proxied ports are available for further connections.

Listing 1. Abstract Framework example in Python

```
#!/usr/bin/python
import ccaffeine.AbstractFramework # load the framework
# Framework-specific portion, in this case Ccaffeine:
a = ccaffeine.AbstractFramework.AbstractFramework() # create it
# initialize telling the framework what components we will use and
# where they are located.
args = "--path ./home/rob/cca/lib/components"
a.initialize(args)
# From here on, this main program is using only standard CCA,
# nothing implementation specific.
# We create this main python program as a component in the framework by
# getting gov.cca.Services:
svc = a.getServices("main", "MainComponent", properties);
myid = svc.getComponentID(); # this is our ComponentID
svc.registerUsesPort("bs", "gov.cca.BuilderService", properties)
port = svc.getPort("bs")
import gov.cca.ports.BuilderService
bs = gov.cca.ports.BuilderService.BuilderService(port)
# From here on everything is the same as if it were
# a "normal" CCA component.
```

vice and `AbstractFramework` interfaces, the CCA also provides an approach to hierarchically encapsulate a network of components as a single component, a design pattern which is unique (as far as we know) in the world of components.

In this way, application developers can manage the complexity presented by their CCA-based applications in a flexible and general fashion. It is hoped that by introducing fewer new concepts, it will be easier to employ `BuilderService` in applications, making even large-scale multi-physics applications more manageable. An important side effect of this approach is that CCA-compliant frameworks need to add little to support this style of containment because most of the existing infrastructure can be reused. Because there are numerous CCA-compliant frameworks specialized in various areas of high end computing, a beneficial artifact of the `BuilderService/AbstractFramework` approach is that disparate frameworks can be linked together using only CCA ports.

6. Acknowledgments

The CCA has been under development since 1998 by the CCA Forum and represents the contributions of many people, all of whom are gratefully acknowledged. We further acknowledge our collaborators outside the CCA Forum and the early adopters of the CCA for the important contributions they have made both to our understanding of CBSE in the high-performance scientific computing context, and to making the CCA a practical and usable environment. We

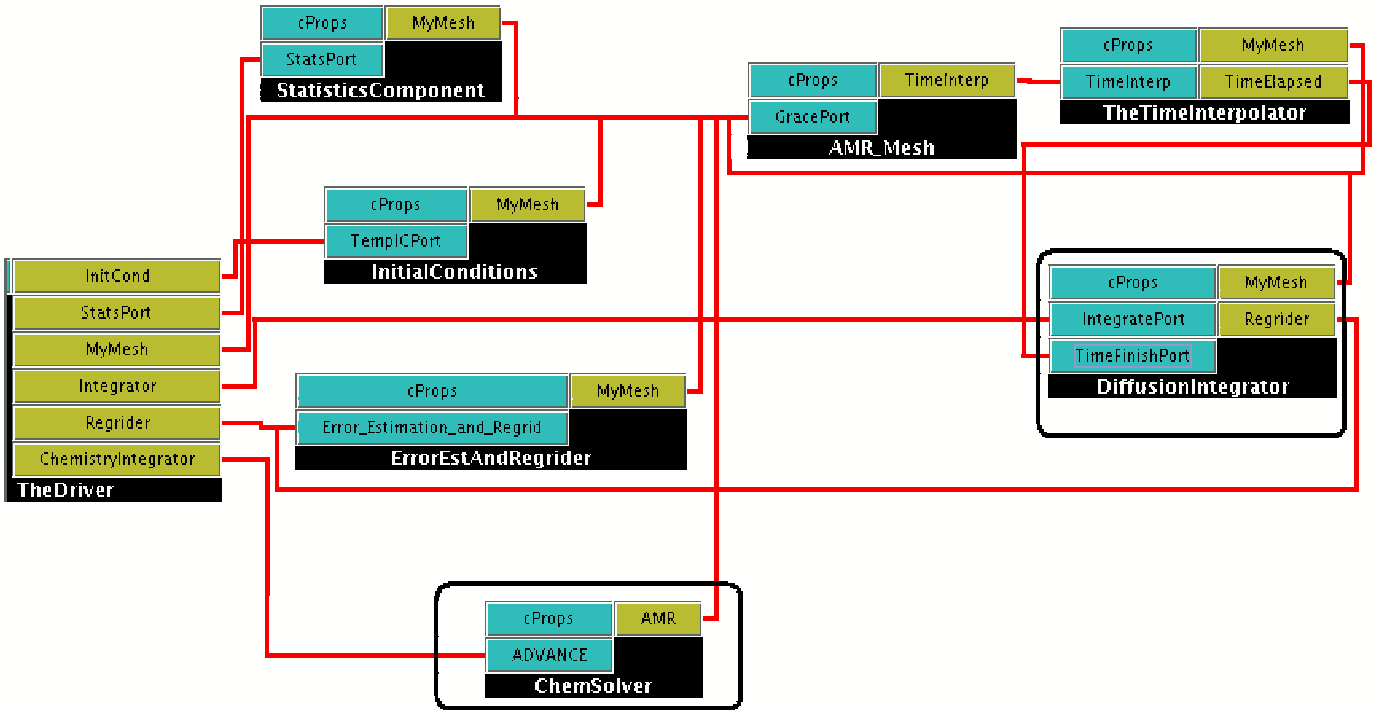

Figure 7. Assembly diagram of the simulation with reaction and diffusion black boxes.

particularly thank Jaideep Ray and Sophia Lefantzi for providing the original components used in our chemical sciences example.

This work has been supported in part by the U. S. Dept. of Energy's Scientific Discovery through Advanced Computing initiative, through the Center for Component Technology for Terascale Simulation Software, of which ORNL and SNL are members.

Oak Ridge National Laboratory is managed by UT-Battelle, LLC for the US Dept. of Energy under contract DE-AC-05-00OR22725.

References

[1] CCA tutorials. `http://www.cca-forum.org/tutorials/`.

[2] CFRFS webpage. `http://cfrfs.ca.sandia.gov`.

[3] Requirements of component architectures for high-performance computing. `http://www.cca-forum.org/documents/requirements.shtml`.

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

[5] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt, and J. A. Kohl. The cca core specification in a distributed memory spmd framework. *Concurrency and Computation: Practice and Experience*, 14(5):323–345, 2002.

[6] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. C. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings of High Performance Distributed Computing*, pages 115–124, 1999.

[7] R. Armstrong and R. B. McCoy. The common component architecture: Fostering an open source community in high performance computing. In *Proc. of the Advanced School for Computing and Imaging*. Center Parcs Het Heijderbos, Heijen, Netherlands, 4–6 June 2003.

[8] Babel homepage. `http://www.llnl.gov/CASC/components/babel.html`.

[9] D. E. Bernholdt, W. R. Elwasif, and J. A. Kohl. Communication infrastructure in high-performance component-based scientific computing. In D. Kranzlmüller, P. Kacsuk, J. Dongarra, and J. Volkert, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 9th European PVM/MPI User's Group Meeting Linz, Austria, September/October 2002. Proceedings*, volume 2474 of *Lecture Notes in Computer Science*, pages 260–270. Springer, September 2002.

[10] D. E. Bernholdt et al. A component architecture for high-performance scientific computing. *Intl. J. High Perf. Comp. Appl.*, in preparation for ACTS Collection special issue.

[11] F. P. Brooks, Jr. *The Mythical Man-Monday: Essays on Software Engineering*. Addison Wesley Professional, second edition, 1995.

[12] Common Component Architecture Forum. http://www.cca-forum.org.
[13] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *ICSM*, pages 131–142, 2000.
[14] O. M. Group. *The Common Object Request Broker: Architecture and Specification*. OMG Document, 1998. http://www.omg.org/corba.
[15] O. M. Group. *CORBA Components*. OMG TC Document orbos/99-02-05, March 1999.
[16] P. Hovland, K. Keahey, L. C. McInnes, B. Norris, L. F. Diachin, and P.Raghavan. 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, June 20 2003.
[17] S. Lefantzi, J. Ray, and H. N. Najm. Using the common component architecture to design high performance scientific simulation codes. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France*. IEEE Computer Society, 2003. Distributed via CD-ROM.
[18] M. M. Lehman. A brief introduction to the FEAST hypothesis and projects (feedback, evolution and software technology). http://www.doc.ic.ac.uk/~mml/feast.
[19] Microsoft COM Web page. http://www.microsoft.com/com/about.asp.
[20] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, June 1999.
[21] B. Norris, S. Balay, S. Benson, L. Freitag, P. Hovland, L. McInnes, and B. Smith. Parallel components for PDEs and optimization: Some issues and experiences. *Parallel Computing*, 28 (12):1811–1831, 2002.
[22] R. Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Sons, 1997.
[23] J. Siegel. OMG overview: CORBA and the OMG in enterprise computing. *Communications of the ACM*, 41(10):37–43, 1998.