Bocca: A Development Environment for HPC Components

Wael Elwasif*
Oak Ridge National Laboratory
P.O. Box 2008, MS6164
Oak Ridge, TN 37831-6164
elwasifwr@ornl.gov

Boyana Norris
Argonne National Laboratory
9700 S. Cass Ave., Bldg. 221
Argonne, IL 60439
norris@mcs.anl.gov

Benjamin Allan
Sandia National Laboratories
7011 East Ave., MS 9158
Livermore, CA 94551
baallan@sandia.gov

Rob Armstrong
Sandia National Laboratories
7011 East Ave, MS 9158
Livermore, CA 94551
rob@sandia.gov

ABSTRACT

In high-performance scientific software development, the emphasis is often on short time to first solution. Even when the development of new components mostly reuses existing components or libraries and only small amounts of new code must be created, dealing with component glue code to obtain complete applications is still tedious and error-prone. Component-based software meant to reduce complexity at the application level increases complexity with the attendant glue code. To address these needs, we introduce Bocca, the first tool to enable application developers to perform rapid component prototyping while maintaining robust software-engineering practices suitable to HPC environments. Bocca provides project management and a comprehensive build environment for creating and managing applications composed of Common Component Architecture components. Of critical importance for HPC applications, Bocca is designed to operate in a language-agnostic way, simultaneously handling components written in any of the common HPC workstation languages: C, C++, Fortran, Fortran77, Python, and Java. Bocca automates the tasks related to the component glue code, freeing the user to focus on the scientific aspects of the application. Bocca embraces the philosophy pioneered by Ruby Rails for web applications: Start with something that works and evolve it to the user's purpose.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments

*Correspondence should be directed to bocca-dev@cca-forum.org

HPC-GECO/CompFrame 2007 Montreal, Canada 21-22 October 2007

General Terms

Common Component Architecture, component development environment

Keywords

CCA, components, Bocca

1. INTRODUCTION

While component-based software engineering seeks to manage complexity for an overall application or a suite of applications, it increases the complexity of the programming task by introducing automatically generated glue code that implements multilanguage or runtime functionality required by the component architecture. Because in a research environment HPC programmers are usually also the end users of their work, they are more impatient than the usual programmer for the results of their running application and less tolerant of delays caused by good software engineering practices. As the capacity and capability of HPC systems increase, the imperative to simulate nature to a higher degree of fidelity expands. Higher fidelity means higher complexity in simulation codes and an increasing need for component-based systems to control that complexity. Thus, sacrificing good software engineering practices in favor or obtaining quick results can no longer produce quick or reliable results. The challenge is to lessen the burden of adoption for HPC components, while preserving performance and accommodating parallel computing. Bocca addresses these concerns for the Common Component Architecture (CCA), a component model for high-performance computing. In the following we give a short description of CCA, the motivation for Bocca, some examples of its use, important design features, and future directions.

1.1 CCA Component Model

The Common Component Architecture is the nucleus of an extensive research and development program in the Department of Energy and academia. On the research side, the effort is focused on understanding how best to apply 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, 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 that 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 [7], and tutorials are already available [1].

The specification of the Common Component Architecture is defined entirely in SIDL and can be expressed in any of the supported languages. Briefly, the main elements of the specification are:

- *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 and publish a port instance to the framework. Components can also use ports, meaning they make calls on port instances obtained from the framework and published by another component.
- 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 [6].

A CCA Framework provides a number of services for components. It maintains the component repository, and creates and destroys components. It gathers the port information and maintains the connections between ports, as shown in Figure 1. Many component models evade the language interoperability issue by requiring components to uniformly written in a particular language (e.g., Java Beans). Recognizing that HPC developers use a variety of languages, CCA solves this problem by using SIDL, an scientific interface definition language, and the supporting code generation tool, Babel [5]. CCA interfaces are specified in SIDL, which enables interoperability between many languages simultaneously. For example, an interface can be defined by SIDL as follows:

```
package mydomain version 1.0 {
    interface StringSource{
        string getString(in int index);
    }
}
```

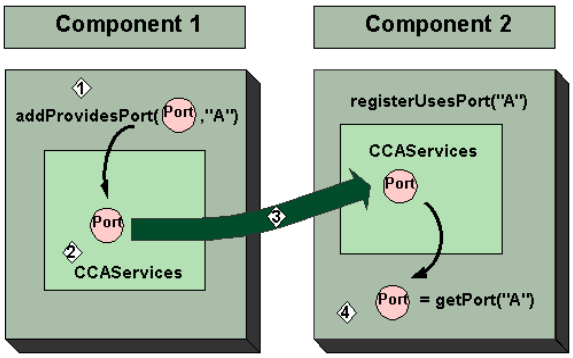

Figure 1: Exchange of a port instance between two components via their CCAServices handles, forming a connection.

where method getString is defined in the StringSource interface. This SIDL code can then be used to generate compatible code in any of Java, C, C++, Python, F77, or Fortran9X. With the aid of the Babel runtime system, invocations on SIDL interfaces from any one of these languages can be passed through to any other. For more information about SIDL see the Babel website [5].

While part of the power of the CCA component model is its language independence, it is also the source of a great deal of complexity in implementation and project maintenance. To support the most general case for a CCA application, the build system must be accommodate all of the compilation and linking idiosyncrasies of all of the languages simultaneously. For that reason, Bocca is indispensable because it creates a build structure around the developer's generated components. Prior to Bocca, the user whad to create, modify, and maintain the component skeleton and build system manually. Enormous effort was expended on understanding the CCA glue code before even the simplest component could be built and executed. Now, by typing a few commands (see Section 2), components are created that build and run immediately. As the developer adds functionality, the build may break, but always the working code can be recovered by backing up and discovering the problem by trial and error. The purpose of Bocca is to let the user create and maintain useful HPC components without the need to learn the intricacies of CCA and waste time and effort in low-level software development and maintenance tasks.

1.2 Motivation for Bocca

Bocca lays down the scaffolding for a complete componentized application without any attendant scientific or mathematical implementation. Although Bocca uses the Common Component Architecture, the concept can be broadly applied. The typical use case proceeds in three parts:

1. *A user must conceive of the application as a collection of components.* Typically a subset or all of those components will be unimplemented, or the implementation will exist only as importable functionality from libraries. Previously, users had to learn the API and syntax associated with CCA and construct the component themselves. Now, given a name and an implementation language, Bocca creates a fully loadable component and integrates it into the build system. The user

can proceed implementation immediately, ignoring all the generated glue code.

2. *The user next identifies conceptually what interfaces are to be exchanged between components.* Normally users would have to learn how to express even empty prototype CCA port interfaces in SIDL and understand how these interfaces are converted into loadable form by the build system. Again, given an interface name and an implementation language, ports can be generated and integrated into the build system with a single command.

3. *The user must then associate these ports with the components.* Normally the user would have to use CCA glue code to express these port providing and using associations. Bocca automatically encodes port associations, associating them with components and updating the dependencies in the build system.

4. *Finally, the user must connect ports among instances of the defined components to complete the application.* Normally this is done in an interactive environment (a CCA GUI or shell). Bocca will automatically generate and build a stand-alone main program suitable for batch execution, producing a log of a GUI or shell session. The main program will be in the Babel-supported language of the user's choice.

At each of these steps Bocca automates code management tasks and keeps the generated glue code out of the way of the code developer, leaving the focus on the application functionality and not on the component or build infrastructure.

Bocca belongs to an emerging class of development tools whose purpose is to embed users' structure and code in a pre-existing architecture. Ruby Rails, a tool for constructing database-backed web applications, is probably the premier example of this class. Relying heavily on ideas associated with Extreme Programming, the idea is to start with a working application that is vaguely similar to the eventual goal and then evolve the application incrementally. This approach has its roots in the observation that most useful code evolves from other code. Bocca provides a similar tool for HPC CCA applications. The relationship of Bocca to general interactive development environments is discussed further in Section 5.

2. CCA APPLICATIONS WITH BOCCA

As a command-line tool, Bocca provides HPC developers with a familiar interface that greatly reduces the barrier to adoption. In keeping with this philosophy, Bocca makes extensive use of default settings that can be set (and reset) by the user at will to further simplify the commands used to manage various application entities. Bocca offers users the ability to refine an initial application design whenever the need for such refinement arises. The user is able to modify and remove Bocca entities, relying on Bocca to properly propagate such changes throughout the application project.

2.1 Example Using Bocca

In this section, we present an example that illustrates the use of Bocca to create the working skeleton of an HPC component-based application. The application represents a recurring pattern in HPC, where a *driver* component interacts with a *model* component that represents the physical system being studied. The *model* component uses a *solver* port to compute an approximation of the mathematical problem description. The functionality of the *solver* port can be provided using a *linearSolver* or a *nonLinearSolver* component, depending on the type of problem and desired solution fidelity. Note that we provide details pertaining to the use of Bocca to create the initial working application skeleton. We also briefly discuss some of the ways Bocca can be used to modify the application in ongoing development subsequent to creation.

Creating a New Project.

The complete sequence of Bocca commands used to create the example application is shown in Figure 2. These commands create the application described above, compile all generated code, and then instantiate the constituent components to verify the absence of any component framework runtime problems that would interfere with the execution of the application. All of these steps are performed before any user code is introduced; that is, Bocca automatically creates buildable and runnable empty components.

Bocca uses the *project* abstraction as a container for a collection of components, ports, and other artifacts. The creation of a new Bocca project accomplished by using the command `bocca create project` is shown on line `2` in Figure 2. A new project resides in a directory that has the same name as the project. Bocca allows the specification of a default language for the new project (FORTRAN 90/95 in this example), as well as a default SIDL package containing project elements (the project name is used as the default top-level package name if it is not specified). Default project settings are stored in a file in a `BOCCA` subdirectory in the top-level project directory and can be modified by the user at any time. We note that the selection of the build system to be used for the newly created project is done at project creation time. Bocca uses a pluggable approach that allows for multiple build systems, as outlined in Section 3. In this example, the default GNU Make-based Bocca project build system is used.

Creating New Ports.

The next stage in the example script (lines 5–8) shows the creation of new SIDL ports that belong to the new project. Executing the `bocca create port` commands anywhere in the project directory associates the newly created ports with the project (and thus inherit project defaults). Bocca allows a newly created port to extend other ports (or plain SIDL interfaces), whether they belong to the current project or not. Upon creating a port, Bocca generates a new SIDL file that contains a functioning skeleton of the new port. The developer can then fill method details in the newly created SIDL port file.

Creating New Components.

Component creation for the sample application is shown on lines 10–25 in Figure 2. The listing shows some of the options available to fully describe the new component. The developer can specify the component implementation language (or default to the project's default language as is the case with the `HPCModel` component). Additional component arguments include the ports it provides or uses, as well as

```
1  # Project Creation
2  bocca create project sim --language=F90
3  cd sim
4
5  # Ports creation
6  bocca create port ConfigPort
7  bocca create port ModelPort
8  bocca create port SolverPort
9
10 # Component creation
11 bocca create component Driver \
12     --language=CXX \
13     --provides gov.cca.ports.GoPort:GOPORT \
14     --uses ConfigPort:CONFIG \
15     --uses ModelPort:MODEL
16 bocca create component HPCModel \
17     --provides ModelPort:MODEL \
18     --provides ConfigPort:CONFIG \
19     --uses SolverPort:SOLVER
20 bocca create component LinearSolver \
21     --language=C \
22     --provides SolverPort:SOLVER
23 bocca create component NonLinearSolver \
24     --language=F77 \
25     --provides SolverPort:SOLVER
26
27 # Project configuration and build
28 ./configure
29 make
30
31 # Test new project skeleton
32 make check
```

Figure 2: Example project script.

any non-CCA interfaces and classes that the component will implement (in the case of SIDL interfaces) or extend (in the case of SIDL classes). One can also specify external (to the project) dependencies that are needed by the component.

When creating a new component, Bocca generates the initial version of the SIDL file containing the definition of the component. Bocca also controls the generation of the appropriate language binding for the new component. In addition, Bocca populates the newly generated language binding with the code necessary to integrate the new component into any CCA-compliant framework.

Bocca not only supports the management of CCA ports and components, it also manages plain SIDL interfaces and classes that are not CCA components. This support allows HPC developers to use a unified application development platform for all their SIDL code. While this simple example does not illustrate this aspect of Bocca, such support is crucial for the majority of component-based HPC applications, which typically use components only to encapsulate a subset of the collection of objects that make up the bulk of the application.

Assembling and Validating the Application.

Lines 27–32 in Figure 2 show the construction and testing of the newly created components. The default Bocca build system used in this example is based on GNU make and autoconf [14]. A sample application can then be constructued with a script or with the Ccaffeine GUI; the result is shown in Figure 3.

2.2 Application Maintenance with Bocca

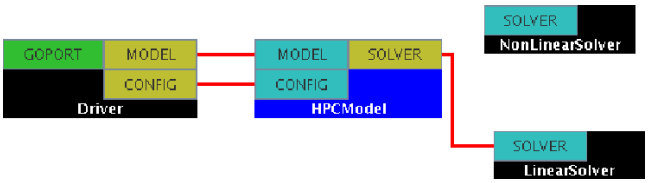

Figure 3: Sample application component connectivity wiring diagram.

We have illustrated the use of Bocca to create skeleton CCA component applications. While this functionality is important to lower the CCA component methodology adoption barrier for HPC developers, it is by no means sufficient to address the entire lifecycle of typical HPC applications. As ports, components, and the overall application design evolve over time, developers will undoubtedly need to change aspects of their ports and components that were specified at time of creation.

Bocca provides facilities to support refactoring the component aspects of the application (those aspects that Bocca generated in the first place). Developers can rename Bocca entities, change some of their attributes (e.g., adding/removing ports to/from components, changing the implementation language of a component, and using a component as a template to create a new component). Bocca propagates the effects of such changes throughout the assembly of entities that constitute a Bocca project, thus giving HPC developers a high degree of flexibility in experimenting with and evolving their design. These operations are not limited to entities that share a common implementation language.

To further support application maintenance, Bocca exports the dependency information of a managed project using the widely supported graph representation of Graphviz [2, 10]. This allows developers to visually analyze their application and explore potential modifications. The dependency graph for the sample application can be seen in Figure 4.

3. DESIGN AND REQUIREMENTS FOR RAPID DEVELOPMENT WITH CCA

Our aim is to decrease to near-zero the time required to define, build, and maintain the component glue aspects of CCA/SIDL-based applications. We design Bocca to capture knowledge about the component framework-related aspects of an HPC application and to use that knowledge to automate as much as possible the process of application construction and maintenance. The information to be captured and managed falls broadly into three categories:

1. The application structure, which is a list of component instances, port connections, input parameter data, and locations of installed components.

2. The ports, classes, and interfaces required to build a given component, as expressed in SIDL and in the uses/provides pattern; the external dependencies (typically libraries) required to build the component,

3. The choice of build tools to which compilation processes are delegated.

Our software design reflects the different types of data that must be managed, as shown in Figure 5. Users can

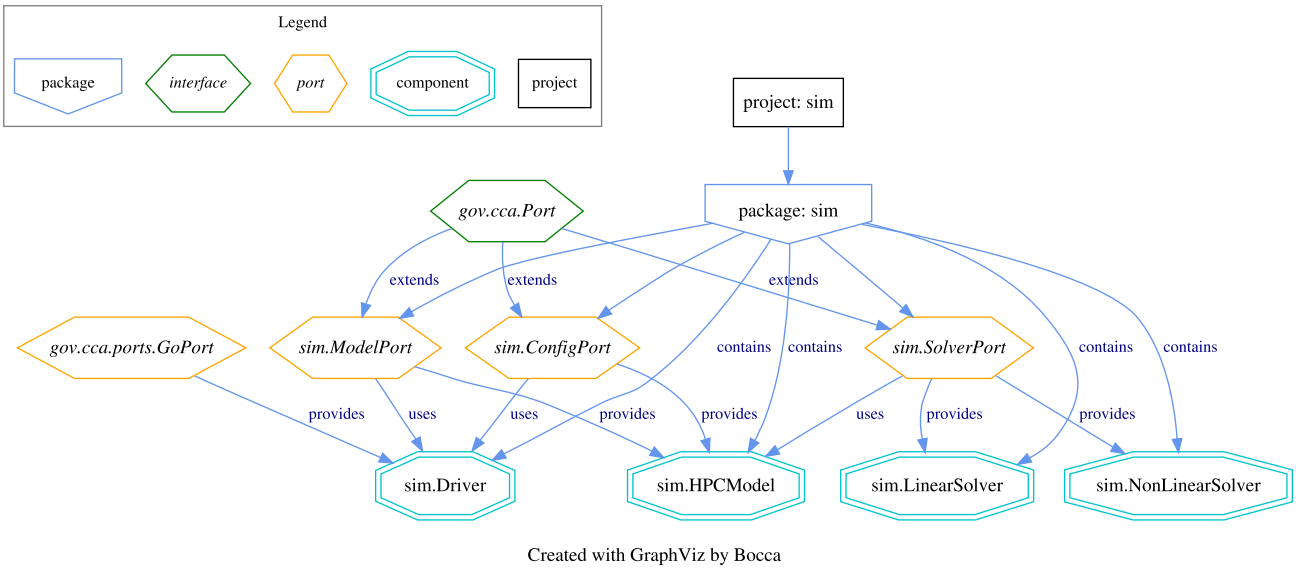

Figure 4: Graph representation of example application. Vertices correspond to project entities managed by Bocca, while edges indicate dependencies between project artifacts.

leverage their favorite existing tools to perform all the functions that are generic to component software development: compilation, source code editing and version management, and execution of the completed applications in some framework.

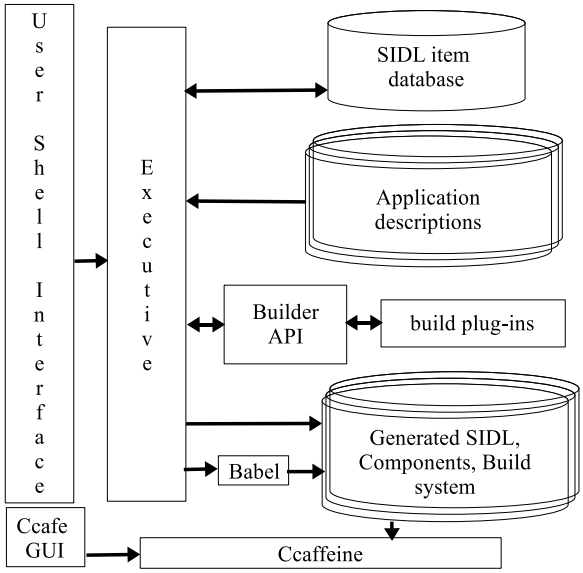

Figure 5: Bocca design overview.

Our implementation is constrained by key requirements with higher priorities in HPC software development than in most circumstances. Specifically, Bocca must do the following:

1. Function well in ill-defined development processes where the project requirements rarely stabilize and where the participants are scientific applications programmers rather than experienced software engineers. Iteration and evolution of application, interface, and component design must be supported.

2. Have low adoption costs: neither a steep learning curve nor complex, unusual, or proprietary software prerequisites are allowable.

3. Have low abandonment costs. Bocca must be able to trivially export individual components or entire applications as packages that may be configured, built, used, and maintained as part of larger, Bocca-free projects that run on petascale and exascale platforms.

4. Remain fully functional in the spartan development environments typical for high-performance architectures. HPC environments frequently lack adequate support for graphical tools displaying on remote desktops and often do not have the very latest versions of common open-source tools available for production use.

5. Be queryable for syntax help and examples as well as the current project state.

6. Support coexistence of distinct projects without interference in the same directory space. The user must be able to specify the directory housekeeping rules to facilitate peer-level integration among Bocca and non-Bocca managed sources in larger projects.

Our solution is to design a command-line tool, similar in look and feel to many version control systems. We address the details of implementing this tool in the next section. The tool performs various actions (create, remove, rename, etc.) on a small set of subjects (component, port, class, interface, application, etc.). All the CCA-related SIDL definitions are

generated by the tool once the user specifies the desired type names, leaving the user to fill in domain-specific method names and implementation code. Graphs (or trees) are a natural choice for organizing SIDL entities and representing their relationships. A small language for describing application construction with a given set of components [3, 4] is already in common use in the CCA community and is easily wrapped into Bocca. Critical for both maintenance and rapid prototyping, Bocca has sufficient project data to handle renaming or removing any SIDL entity and perform automatic updates of both the user-customized sources within the project and the noneditable Bocca-generated sources.

4. IMPLEMENTATION

The primary goals for the design of Bocca are extensibility and portability. From the very start, Bocca development has involved multiple, geographically distributed participants, and we envision having an increasing number of developers contributing functionality and bug fixes. To facilitate this distributed development model, Bocca allows the easy addition of new functionality with minimal changes to the existing code base. Python was chosen for the implementation language of core Bocca functionality based on a combination of criteria, including developer familarity, availability on target platforms, object-orientation, dynamic module handling, built-in data structures and functionality, and ease of use.

Project representation.

Bocca uses a general graph data structure for representing projects and their elements based on a Python graph package developed by R. Dick and K. Gaitanis [8]. The vertices in the graph correspond to CCA project entities, or the *noun* in a Bocca command. The UML diagram in Figure 6 shows the classes involved in the project graph representation. Each vertex type is implemented as a class that extends the default `BVertex` class. Directed edges between vertices represent dependencies. For example, an edge between a `Component` vertex and a `Port` vertex is used when a component uses or provides that port. Edges are implemented in a `BEdge` class, which can be annotated with the type of dependency, for example, "extends," "implements," "uses," or "provides" (see example graph in Fig. 4). Unlike graphical interface development environments, Bocca has the project state in memory only during the execution of a command. At the successful completion of each command that modifies that state, the graph is stored by using Python's `cpickle` module. The graph is loaded in the beginning of command execution and contains a complete description of the current state of the project.

Command Dispatching.

The Bocca dispatcher loads the saved project graph and performs some rudimentary command-line error checking before instantiating the vertex type corresponding to the *noun* specified on the command line. The convention for each `BVertex` subclass is that its name is the capitalized *noun* from the command-line, This allows new *BVertex* subclasses (and thus commands) to be added by simply defining a new Python module in the appropriate location, without having to modify any existing Bocca classes.

Build System.

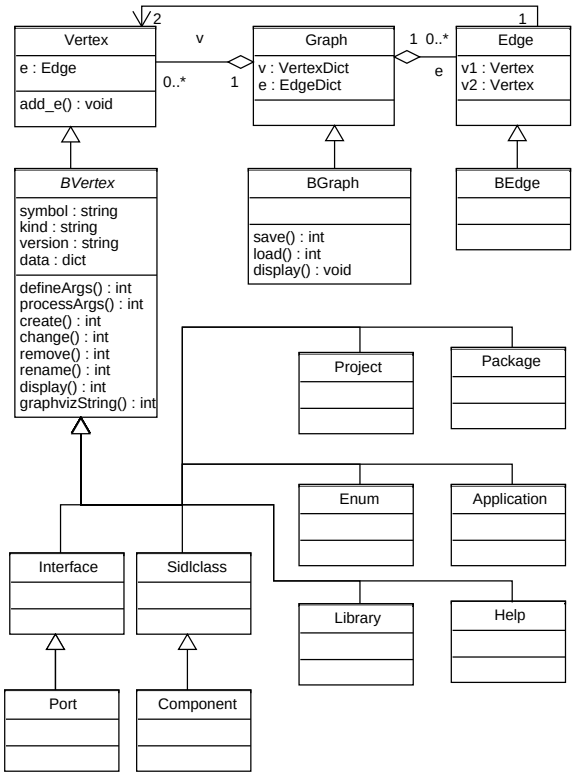

Figure 6: Bocca class hierarchy used for internal project representation.

Bocca does not mandate a specific build system approach. Instead, it defines a set of interfaces (in the `builders` Python module) that allow the interaction with different build systems. This feature broadens the portability of the user's project to more exotic HPC platforms that do not wholly support standard build tools. The initial Bocca implementation includes a GNU make-based build system that fully automates the build of multi-language component projects. New implementations can be added by implementing the `builders` interfaces; the build system can be chosen with a command-line option at the time of project creation.

Help System.

Bocca's help system relies on Python documentation strings, eliminating the need to maintain separate code documentation and help system. The `dispatcher` module treats commands containing `help` or `--help/-h` differently from other Bocca commands in order to provide appropriate help even when the command is executed outside of a Bocca project or the syntax is not correct. Since help information is dynamically obtained from the modules corresponding to each command, Bocca ensures that help will be available for any new modules that simply document their implementations.

Code Generation.

Several Bocca commands require modifications to Bocca-generated code, which may also contain code added by the user. For example, when a port type is renamed, the SIDL

file containing the port definition must be updated, as well as the SIDL and implementation files for any dependent project entities, such as components using or providing the renamed port type.

Bocca contains a `splicers` module, which provides classes for managing changes to different types of source code. The first concern of both Bocca and Babel code generators is never to lose anything a user writes by hand. A block is a set of lines that begins with a line `key.begin(symbol)` where the `key` is a splicer type, and `symbol` is the SIDL symbol associated with the particular splice. Each block ends with a line `key.end(symbol)`. The code between these keys, presumably added by the user, is considered to be the splice. In a Bocca-generated SIDL file, users can add method declarations to interfaces and classes and fields in enum types within the Bocca splicer block. For example, the splicer block for the `ModelPort` in the example in Fig. 2 begins with a comment containing `bocca.splicer.begin(simPorts.Model)`.

Code Refactoring.

As discussed in Section 2.2, in addition to project creation, Bocca automates other refactoring and maintenance tasks, which constitute the bulk of a component's lifecycle. For example, simply renaming an interface or a component requires updates to many files, including the SIDL definition, header and source files in various languages, and makefiles. Changes in SIDL symbols can also result in changes to the project directory structure, which are typically tedious and error prone when done manually. To automate support for multilanguage refactoring operations, Bocca performs code generation in all the languages supported by Babel (Fortran, C, C++, Java, and Python). In addition to propagating refactoring and other changes through Bocca -generated code, Bocca modifies the user's implementation to reduce further the amount of manual labor involved in each change. The graph representation of the project structure is crucial to maintaining a consistent project state by enabling each change to a project element (e.g., a port) to be propagated to all dependent project entities (e.g., ports that extend it and components that use or provide it).

Bocca and Other Tools.

To support scripting, all Bocca commands return 0 upon success or an appropriate error code otherwise. Bocca uses two subdirectories for internal bookkeeping: `BOCCA` and `.bocca`. These subdirectories exist at all levels of the project directory structure. The visible `BOCCA` directory contains user-editable project settings, which can be under revision control if desired. The hidden `.bocca` directory contains the saved binary project graph and other files that are not meant to be edited by the user. The `.bocca` directory includes a project-specific file that contains the relative local path with respect to top project directory. Before any Bocca command is executed, these files are used to verify that the command is being executed within a valid Bocca project directory. This approach aims to reduce problems caused by manual changes to the directory structure.

5. RELATED WORK

Bocca is inspired largely by the same concerns as Ruby Rails [13], a tool for creating database-backed web applications. Principles are also borrowed from the Extreme Programming methodology, where it is assumed that all useful code evolves from other useful code. The idea is to create an entire running application given only brief user input, producing code that is as close as possible to the developer's final goal. From there, the developer evolves the automatically generated code into the desired product. Indeed both Rails and Bocca accommodate the continuing evolution of the code, recognizing that code that does not evolve is dead code.

While Bocca provides some functionality normally available in IDEs, such as Visual Studio [11] and Eclipse [9], it also differs significantly. Most general-purpose IDE frameworks provide project management capabilities for developing general applications in a single language. Most are not devoted to a particular software architecture, though a few explicitly support component-based development (e.g. EJB or Eclipse plugins). Unlike Bocca, most IDEs do not normally support portable multilanguage builds. In contrast to general-purpose IDEs, neither Rails nor Bocca provide any special way of editing or running user code. In fact, much of the functionality of a general purpose IDE is orthogonal to Bocca functionality. For example, the facile editing capability for which IDEs are famous can be used alongside Bocca; the build system tuned to CCA requirements, however, is managed by Bocca, though it is directly callable from IDEs which recognize `make` (or the chosen build system with which the project has been configured).

Bocca and Rails belong to a different class of software development tool than general-purpose IDEs. Both tools concentrate not on code development per se but on creating and maintaining the necessary infrastructure to support their targeted framework. In the case of Rails it is a particular database-backed CGI framework written in Ruby [12] to generate web applications. Bocca generates the SIDL, CCA, and build infrastructure necessary for a component HPC application. The developer is participating in a framework that is the means of delivering the intended application but not the end. Both Rails and Bocca free the user from the need to intimately understand the frameworks in which their code is embedded.

6. QUANTIFYING BOCCA'S IMPACT

Quantitative studies of Bocca's impact on developer productivity are not yet possible, because of the rapid and iterative nature of Bocca 's development. However, we can quantify the artifacts whose creation, modifications, compilation, and testing Bocca automates to a large extent. This process can give us some idea of Bocca's effectiveness in reducing the complexity of developing component software.

We show in Table 1 the quantity and sizes of files generated or provided by Bocca in both the build system (using the default builder plugin) and the application code generated to handle the complexity of assembling a multilanguage component simulation. For the four components and three ports in our example, we see that the set of user-customizable files is small and the code in them compact, while the glue code is spread over a large number of big files. Portable HPC software build systems are notoriously difficult to create, debug, and maintain. Neither the CCA standard nor the Babel tool prescribes complete build processes that yield a running application or even a component library. Bocca fills this void using the pattern set by the builder-plugin of the user's choice, completely hiding the

Table 1: Source complexity managed with Bocca.

User customizable		
Files	Lines	Code type
7	74	SIDL Files
7	165	Build system
7	1251	Babel Impl source
0	331	CCA source (inserted into Impl)

Uneditable		
Files	Lines	Code type
24	486	Babel build system fragments
16	1054	Bocca build system fragments
28	15501	Babel source glue code

size and complexity of the build infrastructure required to assemble the different makefile fragments into a complete build process.

7. CONCLUSIONS

We have created Bocca to enable CCA users to prototype complete codes rapidly and then evolve those codes over the life of their projects. We have demonstrated a path to sharply decreased development times in HPC software creation and maintenance using CCA technology, automating many necessary steps that heretofore could only be performed manually. Bocca is an extensible, command-line-based, interactive development environment for CCA that is focused on five points:

- Automated collection and maintenance of CCA framework related data for use in maintaining component infrastructure and build tree.
- Loose integration, rather than re-creation, of existing tools drawn from the larger HPC and open-source software development worlds.
- Avoidance of lock-in: Bocca users are free to abandon the Bocca tool and its meta-data and maintain their code manually at any time.
- Ease of use, without resorting to multi-context graphical views that inhibit automation and reduce portability.
- Portability and ease of maintenance of Bocca itself.

Bocca may be viewed as an integration platform or as a piece of an even bigger integration platform for CCA-based software development. Directions of ongoing research in the CCA arena that would provide natural extensions to the Bocca tool include automated generation of proxy components (for use in debugging, regression testing, and developing parallel performance models) and tools for mostly automated wrapping of legacy code into CCA interfaces and components using simple patterns. Also beneficial would be tools to do more automated refactoring of complex existing codes by reaching inside to replace all calls on a library with calls on CCA interfaces. These tools would allow more plug-and-play experimentation with existing codes and would be especially useful for perfecting componentization in difficult implementation languages such as Java and C++ (see Section 4).

Future development will be directed at broadening the range of applicability and increasing HPC user acceptance of Bocca and CCA tools in general. Builder plug-ins to support other build tools on exotic hardware will be required by many HPC projects. A lightweight graphical or even web-based interface layer driving the Bocca command-line will lower the barrier to first adoption and be a welcome addition to a CCA tutorial.

Acknowledgments

This work was supported 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 DE-AC02-06CH11357; Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the U.S. Dept. of Energy under contract DE-AC05-00OR22725.

8. REFERENCES

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

[2] Graphviz - Graph Visualization Software. `http://www.graphviz.org/`.

[3] B. A. Allan and R. C. Armstrong. CCA tutorial: Introduction to the Ccaffeine framework. `http://www.cca-forum.org/tutorials/archives/2002/tutorial-2002-06-24/tu%torialModFramework.pdf`, 2002.

[4] 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):1–23, 2002.

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

[6] D. E. Bernholdt, R. C. Armstrong, and B. A. Allan. Managing complexity in modern high end scientific computing through component-based software engineering. In *Proc. of HPCA Workshop on Productivity and Performance in High-End Computing (PPHEC 2004), Madrid, Spain*. IEEE Computer Society, 2004.

[7] D. E. Bernholdt and et al. A component architecture for high-performance scientific computing. *Intl. J. High Perf. Comp. Appl.*, 20(2):163–202, 2006.

[8] R. Dick and K. Gaitanis. graph - Directed and undirected graph data structures and algorithms. `http://ziyang.ece.northwestern.edu/~dickrp/python/mods.html`, 2005.

[9] Eclipse Foundation. Eclipse. `http://www.eclipse.org`, 2007.

[10] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.

[11] Microsoft. VisualStudio. `http://msdn2.microsoft.com/en-us/vstudio/default.aspx`, 2007.

[12] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, second edition, Oct. 2004.

[13] D. Thomas, D. H. Hansson, L. Breedt, M. Clark, J. D. Davidson, J. Gehtland, and A. Schwarz. *Agile Web Development with Rails*. PragmaticProgammer, second edition, Dec. 2006.

[14] G. V. Vaughn, B. Ellison, T. Tromey, and I. L. Taylor. *GNU Autoconf, Automake and Libtool*. New Riders Publishing, Thousand Oaks, CA, 2000.