Scientific Programming 16 (2008) 315–327
DOI 10.3233/SPR-2008-0270
IOS Press

# Managing scientific software complexity with Bocca and CCA

Benjamin A. Allan [a,*], Boyana Norris [b], Wael R. Elwasif [c] and Robert C. Armstrong [a]

[a] *Sandia National Laboratories, Livermore, CA, USA*
[b] *Argonne National Laboratory, Argonne, IL, USA*
[c] *Oak Ridge National Laboratories, Oak Ridge, TN, USA*

**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 the component glue code and software build processes to obtain complete applications is still tedious and error-prone. Component-based software meant to reduce complexity at the application level increases complexity to the extent that the user must learn and remember the interfaces and conventions of the component model itself. 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 high-performance computing (HPC) applications, Bocca is designed to operate in a language-agnostic way, simultaneously handling components written in any of the languages commonly used in scientific applications: C, C++, Fortran, 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 on Rails for web applications: start with something that works, and evolve it to the user's purpose.

Keywords: Code generation, software development environment, interface definition language, SIDL

## 1. Introduction

Software complexity has many measures. It can be naively quantified in terms of code measurements (e.g., lines of code) or in the number of instructions needed to perform a defined task. Programmers today, however, rarely create monolithic stand-alone programs; more often they find themselves contributing to a larger framework or runtime system. A better complexity measure in this case is the learning curve before the programmer can use the framework and contribute to it. Though hard to quantify, the intellectual cost of learning a new framework can be greater than the value of a programmer's contributed code. Here we describe a tool called Bocca whose main goal is to ease the complexity of developing high-performance software using the Common Component Architecture and automating many difficult and error-prone development tasks.

[*] Corresponding author: B.A. Allan, Sandia National Laboratories, 7011 East Ave., MS 9158, Livermore, CA 94551, USA. E-mail: baallan@sandia.gov.

### *1.1. The common component architecture*

The Common Component Architecture (CCA) is a component model that admits and supports a model for parallel computing [2,3,14]. As a component model CCA must mechanize linking together components at runtime without sacrificing performance. Among other things, this means that CCA must allow languages that are associated more with performance than with ease of use – C, C++ and Fortran – and yet admit languages more familiar to component computing, such as Java and Python. Indeed, unlike most other component models, CCA eschews language details in its component model, furnishing a connection mechanism based on *providing* and *using* language-independent interfaces (see Section 1.2). Two components can be linked when one component advertises that it can provide an interface of a particular type for which another component advertises a need. Interfaces that are donated or imported in this way are called *ports*. A CCA-compliant framework keeps track of uses and provides ports and takes care of transferring the port from providers to users (for more details see [2,7,10]).

## 1.2. Scientific interface definition language and Babel

In addition to defining a design pattern for composition, pragmatically a component model must be concerned about the mechanics of connecting the software together. Where performance is not an issue, connections can be made through a network line protocol (e.g., CORBA Component Model [16]). Another approach is to require that the components be written in a particular language (e.g., Enterprise JavaBeans [8]). Neither of these options is open to the scientific computing developers of the Common Component Architecture, requiring that it provide a language interoperability layer. This layer, called the Scientific Interface Definition Language (SIDL) [15], is a specification for defining interfaces that can be used to generate interoperable code in a variety of languages. Babel [4] is the SIDL interpreter that the CCA uses currently supporting Fortran 9X, Fortran 77, C, C++, Python and Java. Given a valid SIDL interface definition, Babel will generate the skeletons (implementation side bindings) and stubs (calling side bindings) for any of the supported languages invoking any other. An important difference between Babel and other language interoperability tools (e.g. Simplified Wrapper and Interface Generator, or SWIG [5]) is that Babel provides two-way invocation: each supported language can call or be called by every other. Special care has been taken by the developers of Babel to reduce the performance overhead of using Babel to the level of a few function calls. Developers are thus free to create their components in the language with which they are most comfortable and be certain that their work will be usable by other developers working in any of these supported languages.

## 1.3. Bocca: A CCA component generator

The best description of Bocca is that it is an *application generator* or as described in this paper, a *CCA component generator*. Armed with only the names of component and port types, users can create a complete skeleton application out of components in minutes. What Bocca does for the Common Component Architecture is similar to what Rails does for Ruby-enabled web applications (for more on this see Section 6). Bocca creates an entire application shell that runs out of the box and is ready for the programmer to insert implementation code. It adopts the philosophy of extreme programming [6] that all working code comes from other working code. Bocca gives the programmer an entire application that is "close" to what is wanted. From there the code can evolve to what the programmer has in mind. Because Bocca provides the component-oriented glue code and attendant build system, the programmer needs to learn only that which directly impacts the application, and nothing else. The value of Bocca is measured in the lines of code that the programmer did not have to write and, more important, understand. As we shall see in the following example, the savings, even for the simplest componentized applications, are considerable.

The utility of Bocca can be illustrated by using a simple example that generates the basic skeleton of a functioning application composed of two CCA-compliant components.

Figure 1 gives a quick demonstration of the power and simplicity of Bocca. Line 4 creates a new project called *myProject* by constructing a new directory by the same name and importing the build system into it. Every command related to this project must take place inside this project directory or one of its antecedents. In line 8 we create a CCA port type called `myPort`, which at present has no methods but is nonetheless a valid port. In lines 10 and 12 we create two components: *Worker*, which provides a `myPort`, and *Driver*, which uses it. The *Driver* component also provides a *GoPort* that provides a standard *GoPort*, which is the application entry point (similar to a traditional `main` program). In addition to the port type `myPort` each component needs an instance name for the port to disambiguate multiple ports of the same type on the same component. Here *Worker* names its `myPort` "`xport`" because it is providing a `myPort` and Driver names the port it will use "`wport`". We configure, build, and test the project with the `configure`, `make`, and `make test` commands. At this point we have an empty "shell" components and ports that have no methods or corresponding implementations but nevertheless can be linked together and "run" as a complete application (see Fig. 2).

Componentizing applications is a way, possibly the best way, to manage complexity and allows a large project to scale both in size and number of developers. The component model itself, however, introduces a different type of complexity that does not exist in noncomponent software. Part of the motivation for Bocca was to overcome this learning curve by creating a runnable template for a componentized application into which users can easily insert code. In this way users can learn the component model incrementally and in steps that are most relevant to their immediate goals.

```
#!/bin/sh
# Project Creation

bocca create project myProject

cd myProject

bocca create port myPort

bocca create component Worker --provides=myPort:xport

bocca create component Driver --uses=myPort:wport --go=GO

./configure

make

# Test new project skeleton
make check
```

Fig. 1. Simple Bocca project creation script.

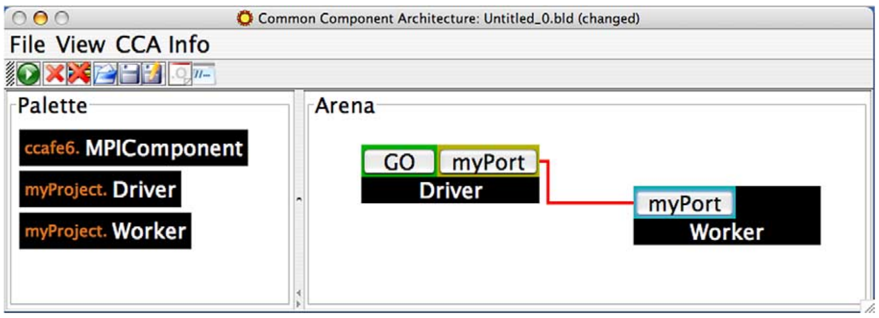

Fig. 2. An empty "shell" application resulting from the commands given in Fig. 1 shown in the CCA GUI.

## 2. Attacking complexity

Scientific software development and maintenance processes have several kinds of superlinear social and technical complexities that make them generally unscalable in costs for very large projects. We aim, by providing Bocca and CCA, to directly reduce several complexities of software development processes by enabling the following approaches:

- scientific language interoperability,
- software componentization,
- configuration management by default,
- source code layout by default,
- portability by default.

The complexity reductions our approach affords are easily expressed with the following numbers: $N_{\mathrm{L}}$ (number of programming languages), $N_{\mathrm{TLOC}}$ (total lines of code), $N_{\mathrm{SF}}$ (source file count), $N_{\mathrm{IL}}$ (interface lines of code), $N_{\mathrm{DEV}}$ (number of developers), $N_{\mathrm{PL}}$ (number of platforms), $N_{\mathrm{AP}}$ (number of applications), $N_{\mathrm{C}}$ (number of existing code libraries), $N_{\mathrm{SH}}$ (number of shells) and $N_{\mathrm{CM}}$ (number of configuration management tools).

Combining codes from among the languages frequently encountered in scientific computing toolkits (C, C++, Fortran 77, Fortran 90, Fortran 2003, Python and Java) is an arduous task if done manually. Many (more than $N_{\mathrm{L}}^2$) incompatible, point-to-point tools exist to handle particular pairs of languages. Our use of SIDL and Babel reduces interoperability complexity from $N^2$ to 1: The owner of a code writes the multilanguage wrapper for that code in the same language after specifying the wrapper with SIDL.

Software componentization reduces requirements on mental information processing in three areas by strictly enforcing conventions for public interfaces and private implementations of all functionality. We obtain this often promised but seldom delivered result because a runtime framework is responsible for all interactions across component boundaries and it is difficult for a component developer to understand private implementation of another component which is not in the same language.

The first area of complexity reduced by Bocca is manual code inspection. The quadratic potentially buggy (or low performance) interactions that must be managed decrease from $N^2_{\mathrm{TLOC}}$ to $N^2_{\mathrm{IL}}$ when interface discipline is enforced.

The second area reduced is design time spent negotiating interactions between existing codes when being integrated into larger projects. The specification of smaller, language-neutral interfaces in SIDL focuses discussions away from $N_{\mathrm{DEV}} * N_{\mathrm{L}} * N_{\mathrm{C}}$ shortcuts and toward $N_{\mathrm{IL}}$ lines of SIDL representing a maintainable, portable solution.

The third area reduced is coding custom initialization and launch processes on parallel platforms. $N_{\mathrm{PL}} * N_{\mathrm{AP}}$ such efforts are simply eliminated once each code is built of components and the problem of launching a generic component framework has been solved for a given platform.

Configuration management is a necessary part of scientific computing; since no operating system or compiler vendor meets the evolving needs of all libraries used in a portable application code. Manually managing $N_{\mathrm{PL}} * N_{\mathrm{SH}} * N_{\mathrm{CM}}$ compilation scripts with source code variations is too much for scientists and developers, who often do not even have access to one or more of the platform and compiler combinations on which the libraries they contribute must run. Through Bocca we reduce the build clutter by automatically providing autoconf and gnu make scripts for these tasks, reducing $N_{\mathrm{PL}} * N_{\mathrm{SH}} * N_{\mathrm{CM}}$ complexity to O(1) complexity. A key part of the provided build support is that Bocca-built software can be automatically installed for production use following common standards. These standards make detecting the installed packages much easier for downstream software builds.

Creating and editing files require social effort on large projects. Development effort can be lost if conflicting changes are made in the same files. In the general case, there may be $N_{\mathrm{DEV}} * N_{\mathrm{SF}}$ interactions. Bocca prescribes that separate components are kept in separate directories. This approach provides strong hints about intellectual file ownership and editing responsibility, reducing the likelihood of accidental losses from conflicting edits of the same file. Also, this directory layout allows those who wish to exchange code via tar files instead of the formality of version control to easily export just that piece.

## 3. Enabling rapid development

Our aim is to decrease to near-zero the time required to define, build, and maintain the component glue aspects of CCA applications. We designed Bocca to capture knowledge about component enabled high-performance computing (HPC) applications 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. An application: a list of component instances, port connections, runtime input parameters, and component installations.
2. The SIDL entities and the uses/provides design pattern that defines a given component. This includes any external dependencies (typically libraries) required to build the implementation.
3. The choice of build tools.

Our software design reflects the different types of data that must be managed (see Fig. 3). Developers can leverage their favorite existing tools to perform all the functions that are generic to component software de-

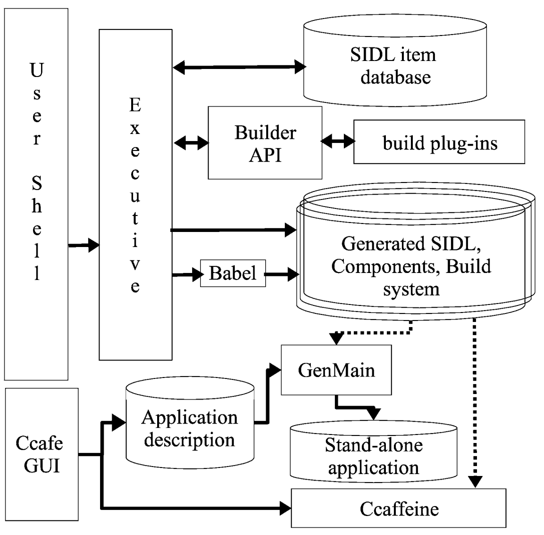

Fig. 3. Bocca design overview.

velopment: compilation, source code editing and version management, and execution of the completed applications in some framework. Packagers and users of the resulting software package need have no knowledge nor installation of Bocca at all.

Bocca is constrained by the unique key requirements of HPC software development, 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. Be inexpensive. Complex, unusual, or proprietary software and expensive training prerequisites are not allowable.
3. Have low abandonment costs. The user 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.
4. Remain fully functional in the spartan development environments typical for high-performance architectures. Production HPC environments frequently lack particular graphical tools, access to remote desktops, and the very latest versions of common open-source tools.
5. Be queryable for syntax help and examples as well as the current project state.

Our approach is to design a command-line tool, similar in look and feel to many version control systems. The tool performs various actions (e.g., `create`, `remove`, `rename`) on a set of SIDL and CCA subjects (e.g., component, port, class, interface). All the 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. Specifics of these data structures are discussed in Section 5. A small language for describing application [1,3] construction from Bocca-generated components is discussed in Section 4.3.

For both maintenance and rapid prototyping, Bocca has sufficient project data to handle renaming or removing any SIDL entity and then automatically update both the user-customized sources within the project and the noneditable Bocca-generated sources (see Section 5).

## 4. Code lifecycle management

The simple example presented in Section 1.3 illustrates how Bocca can be used to rapidly build a functioning prototype of a component-based application. The true utility of Bocca lies in the ability to manage the complete life cycle of complex component-based applications. Such applications comprise the full range of SIDL entities interconnected by nontrivial inheritance and dependency relationships. Figure 4 shows the different Bocca commands used to manage different phases in the lifecycle of managed code.

Bocca uses the `project` abstraction as the container for all code artifacts. As can be seen in Fig. 4, upon creation of an initial set of artifacts that reflect the intended structure of the target application, the user then goes into an update-query-refine cycle to fine tune the application and add implementation details to the various code skeletons generated by Bocca.

### 4.1. Code creation

While the full description of Bocca command syntax and various options is beyond the scope of this paper, we nonetheless outline some major themes that underline Bocca's utility as a rapid development environment and code maintenance and management tool. Details can be found in the CCA Tutorial Hands-On Guide [9].

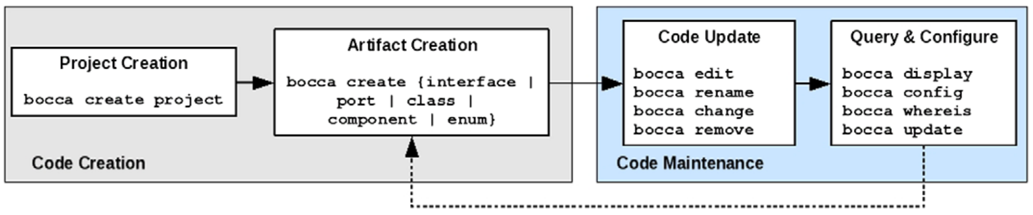

Fig. 4. Code life cycle management using Bocca.

**Extensive use of defaults:** Bocca uses default values for many aspects that define created code artifacts. For example, the programming language specified at project creation time becomes the default language for classes and components belonging to this project. If a default language is never specified, then the default language is C++. The use of defaults, especially for component-related aspects of the code, expedites the process of creating a working application skeleton, while maintaining the user's ability to refine the design afterwards.

**Dependency management:** Bocca uses dependency information explicitly provided by the user, as well as those implicit in the created artifacts, to automatically generate code that reflects this dependency in all supported languages. For instance, when a port is used by a component, Bocca automatically generates a SIDL method that incorporates the used port symbol as an argument. This in turn causes the Babel compiler to generate the correct code that refers to any header files or modules needed to access code for the used port. This dependency management is particularly important as it is used in the Bocca-generated build system to streamline the build process.

**Interaction with external code:** Bocca supports the use of code that is *external* to the current project using one of two approaches. The first approach incorporates such code into newly developed project artifacts (e.g. the `--import-sidl` and `--import-impl` command line option). This approach is particularly useful to migrate legacy code into Bocca-managed projects. The alternative uses external SIDL symbols *in-place*, caching their definition in a local directory for use by the build system. This latter approach allows for parallel development of inter-dependent projects, without the need to totally subsume one project into the other.

## 4.2. Code maintenance

Bocca also plays an essential part in evolving and fine tuning an initial application prototype as the developer gains a better understanding of the developed code. At the core of Bocca's ability to carry out application maintenance lies the detailed SIDL and CCA component dependency information maintained by Bocca, which covers all managed code artifacts.

Commands that can be used to evolve a component-based application include `bocca rename`, which changes the name of a code artifact; `bocca edit`, which streamlines the editing process of Bocca-generated code; and `bocca display`, which queries the Bocca-maintained project database for information related to project entities. Such commands also include `bocca change`, which captures changes applied to the structure of the code entities themselves and their inter-dependencies. Such a change may include adding (or removing) a *port* used (or provided) by a component, changing the inheritance attributes of an interface or a class, or capturing the dependency on a symbol in a method argument to properly structure the build system. Code maintenance commands also include `bocca remove`, to delete a code entity from the managed project.

## 4.3. Application generator

*Appgen* is an application generator that interprets directives for manipulating components and creates a `main()` program in a language of the user's choice. While not strictly part of Bocca, it works with Bocca-generated components. The generated code can be used as is or incorporated as part of a larger application. Appgen takes advantage of the fact that the CCA specification requires that compliant frameworks may also be used passively like a library, instantiating components and linking them together on behalf of an driving application. This embedding of CCA component networks allows even `main()` programs to appear in a CCA framework as a component itself [3].

Figure 5 is an example of Appgen input source that creates a standalone application that runs the example from Section 1.3. The `path` command in line 2 tells the framework where to find components in the filesystem. Lines 5 and 6 define how the symbols for the particular component class are to be loaded in the case of a shared object library. If the components are statically linked in, such as might be the case in a high-performance setting, then these statements have no effect. `get-global` means load all of the symbols globally so they are resolved at the top level for the entire application. Lines 10 and 11 instantiate the components by their classname (first argument) giving each a unique instance name (second argument). The instance name will be used to refer to these components for configuration and invocation. Next, in line 14, we connect the `myPort` uses port on the `Driver` to the provides port of the same type on the `Worker` compo-

```
1  # identify where the appgen tool can find the components:
2  path set /usr/share/cca
3
4  # retrieve and load the component symbols:
5  repository get-global myproject.Driver
6  repository get-global myproject.Worker
7
8  # instantiate the components and give them
9  # the names ''driver'' and ''worker'':
10 create myProject.Driver driver
11 create myProject.Worker worker
12
13 # connect the components:
14 connect driver myportname worker myportname
15
16 # run the driver component by invoking its ''go'' port:
17 go driver
```

Fig. 5. Input for the Appgen tool which will create a `main()` program in the language of the user's choice, assembling and running the components as an application.

nent. Finally, on line 17, we invoke the `GoPort` on the `Driver`. This yields the same application shown from the GUI view (see Fig. 2) but as an executable generated in any of the supported programming languages.

## 5. Implementation

Bocca is implemented in Python by using an extensible architecture of interfaces and modules corresponding to the different parts of the system.

### 5.1. Project graph representation

Unlike graphical IDEs, Bocca cannot maintain elaborate project state in memory and must thus rely on an efficient persistent representation of all project information. We have chosen a graph representation for expressing the relationships between project entities, based on a Python graph package developed by Dick and Gaitanis [12]. The vertices of the graph generally correspond to SIDL entities, such as package, interface, port, class, component and enum. Dependencies between entities are represented by edges between the corresponding vertices. There are several types of dependencies between vertices:

- "extends" dependence between an interface that extends another interface, or a class that extends another class;
- "implements" dependence between a class that implements an interface;
- "uses" and "provides" dependencies between ports and components;
- "contains" dependence, which is used to express the relationships between packages and the SIDL entities they contain, as well as noninheritance-related dependencies between any two SIDL entities, for example, a SIDL interface used as an argument type in a class method.

The project graph representation is serialized and stored in ASCII form. Initially Bocca used Python's *cPickle* serialization, but it proved more cumbersome to implement and maintain the special handling of graph entities (due to potential circular references) than to implement a slightly less general custom serialization for the graph. Storing the graph in a binary format is possible, as well, but does not offer significant performance advantages. In addition, the persistent project state file is human-readable, aiding in debugging. A high-level visual representation of the project is also available via a Graphviz [13] dot file, which is automatically generated by the `bocca display` command.

Figure 6 shows a visualization of the graph representation of the simple Bocca project presented in Section 1.3, which consists of a package containing a port (*myProject.myPort*) and two components: *myProject.Component1*, which uses the port, and *myProject.Component2*, which provides the port.

### 5.2. Command dispatching

Each Bocca *action* (e.g., `create`, `change`, `display`) corresponds to a method in the interface that all

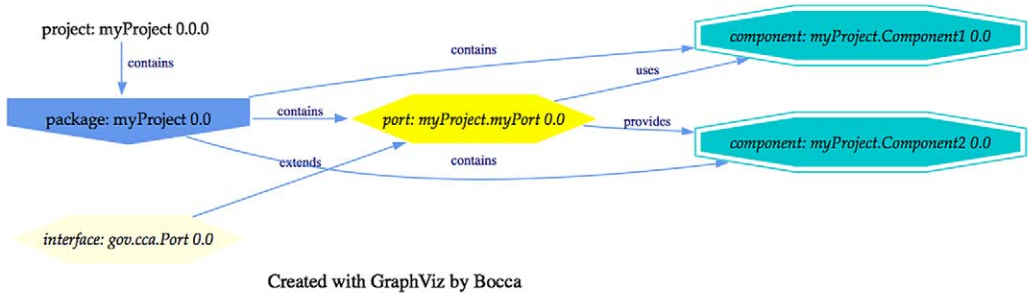
project: myProject 0.0.0
contains
package: myProject 0.0
contains
contains
port: myProject.myPort 0.0
uses
provides
component: myProject.Component1 0.0
component: myProject.Component2 0.0
extends
contains
interface: gov.cca.Port 0.0
Created with GraphViz by Bocca

Fig. 6. Graph project representation illustration for a simple Bocca project.

pertinent SIDL entities implement. Each Bocca command also has a *subject* (e.g., project, port, component), which is the entity to which the action applies. Sometimes the subject is implicit, as in `bocca display`, where the subject is `project`, or when a SIDL name for an entity that exists in the project is used, as in `bocca display MyPort`, where the subject is `port`. In the graph project representation described in Section 5.1, the *BVertex* interface contains methods corresponding to all command-line actions. Each *subject* is a class descended from *BVertex*.

A `dispatcher` module is responsible for determining what the action and subject are on the command line. The only option the dispatcher recognizes is `--help/-h`. All other options are parsed and handled by the class corresponding to each subject. For each Bocca command, the dispatcher performs the following steps:

1. Determine and validate the action and subject.
2. Load the project graph and attempt to determine the subject if it was not given on the command line.
3. Load the Python module containing the class implementing the subject (e.g., Interface, Port, Class, Component).
4. When the `create` action is present, create a new instance of the subject class.
5. For refactoring actions (e.g., `change`, `rename`), retrieve the vertex from the project graph corresponding to the specified subject.
6. Invoke the method corresponding to the action on the instance corresponding to the subject.
7. After completion of the method, perform cleanup and, if necessary, save the project state.

The dispatcher also performs some special help action handling (to support the help command outside of a Bocca project) and allows some flexibility in the command line syntax, such as not specifying the subject explicitly.

## 5.3. *Code generators*

Bocca automatically generates SIDL definitions and boilerplate component code in the implementation for all Babel-supported languages. Babel parses SIDL and generates code in C, C++, Fortran, Java or Python, using `splicers`, or special structured comments, to designate portions of the code, referred to as *splicer blocks*, that can be edited by the user. The first concern of both Bocca and Babel code generators is never to lose anything a user writes by hand. A splicer 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 splicer block ends with a line *key*`.end(symbol)`. Any text contained between the `begin`/`end` splicer comments is not modified by Babel, while any code outside of a splicer block is regenerated when Babel is applied to the SIDL file. Bocca generates SIDL files containing splicer blocks for user comments, methods (in interfaces and classes), and enumerators (in enums).

After Babel generates the implementation code in the target language, Bocca inserts component and other boilerplate code within the Babel splicers. Two types of code are generated by Bocca:

- Bocca default code: boilerplate code for getting ports, handling exceptions and other common operations associated with CCA components or SIDL interfaces. This code is generated once and can be modified or removed entirely by the developer.
- Protected code: the contents of protected blocks are essential for compilation and correctness and should not be modified or removed by the user.

For example, Fig. 7 contains a code excerpt from the implementation of the “Driver” component generated from the example in Section 1.3 and illustrated

```
/**
 *
 * Execute some encapsulated functionality on the component.
 * Return 0 if ok, -1 if internal error but component may be
 * used further, and -2 if error so severe that component cannot
 * be further used safely.
 */
int32_t myProject::Driver_impl::go_impl ()
{
  // DO-NOT-DELETE splicer.begin(myProject.Driver.go)
// User editable portion is in the middle at the next Insert-UserCode-Here line.

// Bocca generated code. bocca.protected.begin(myProject.Driver.go:boccaGoProlog)
  int bocca_status = 0;
  // The user's code should set bocca_status 0 if computation proceeded ok.
  // The user's code should set bocca_status -1 if computation failed but might
  // succeed on another call to go(), e.g. when a required port is not yet
  // connected.
...
  gov::cca::Port port;

  // nil if not fetched and cast successfully:
  myProject::myPort myPort;
  // True when releasePort is needed (even if cast fails):
  bool myPort_fetched = false;
  // Use a myProject.myPort port with port name myPort
  try{
    port = this->d_services.getPort("myPort");
  } catch ( ::gov::cca::CCAException ex ) {
    // we will continue with port nil (never successfully assigned) and
    // set a flag.
...
  }
  if ( port._not_nil() ) {
    // even if the next cast fails, must release.
    myPort_fetched = true;
    myPort = ::babel_cast< myProject::myPort >(port);
    if (myPort._is_nil()) {

#ifdef _BOCCA_STDERR
      std::cerr << "myProject.Driver:␣Error␣casting␣gov::cca::Port␣"
                << "myPort␣to␣type␣"
                << "myProject::myPort" << std::endl;
#endif //_BOCCA_STDERR

      goto BOCCAEXIT; // we cannot correctly continue. clean up and leave.
    }
  }

// Bocca generated code. bocca.protected.end(myProject.Driver.go:boccaGoProlog)

  // When this try/catch block is rewritten by the user, we will not change it.
  try {
    // All port instances should be rechecked for ._not_nil before calling
    // in user code. Not all ports need be connected in arbitrary use.
    // The uses ports appear as local variables here named exactly as on the
    // bocca commandline.

    // Insert-UserCode-Here {myProject.Driver.go}
...
  // DO-NOT-DELETE splicer.end(myProject.Driver.go)
}
```

Fig. 7. Excerpt of the Bocca-generated code for Driver.GoPort from the example in Fig. 1, line 12.

by Fig. 1. Specifically, this code is generated for the "GO" method on the Driver component in C++. Recall that this method is called by a CCA-compliant framework to start execution. Line 10 in Fig. 7 is the start of the splicer block that has been generated by Babel to protect user code from changes by Babel. Within this block are special "protected" blocks generated by Bocca of the form `bocca.protected.begin(`*symbol*`)` closed by `bocca.protected.end(`*symbol*`)` (line 13 closed by line 50 in Fig. 7; note that this is only a partial listing because of space considerations). This block will be parsed and changed by Bocca and serves as a warning to developers that alterations may break the code or will be lost. All user modifications outside of `bocca.protected` blocks and inside `splicer` blocks will be preserved.

### 5.4. *Project refactoring*

In addition to project creation, Bocca automates many project refactoring and maintenance tasks. Before Bocca, some common operations, such as removing or renaming an interface or a class, were time-consuming and error prone, requiring manual updates to SIDL files, header and implementation files in various languages and makefiles. Bocca relies on the dependence information in the project graph representation to propagate all changes correctly and automatically updates affected files, as well as the project graph.

#### 5.4.1. *Importing existing code*

In many cases, the developer is not starting from a blank slate and may wish to leverage existing SIDL and implementation code when creating or modifying a Bocca project. To serve this common need, Bocca provides import functionality, both for SIDL and for implementation code in the languages supported by Babel. To support SIDL imports, Bocca contains a SIDL parser, implemented using Ply [11], which is a pure-Python implementation of the popular compiler construction tools lex and yacc. As with other Bocca commands, the `--import-sidl` option can be applied to different SIDL entities. An entire SIDL package can be imported into a Bocca project or package, or individual interfaces, ports, classes, components, or enums can be imported into an existing Bocca package.

In addition to importing SIDL, Bocca can import implementation code from existing Babel-based software. The import implementation relies on the Babel splicers in the code and thus does not need to parse each of the supported languages. Usually a few manual changes are necessary after importing implementation source code.

#### 5.4.2. *Managing the project state*

As described in Section 5.2, the persistent graph project representation is loaded and stored before and after each Bocca command. When a Bocca command fails, the persistent state is not modified.

Many Bocca commands result in the creation of new files or modifications to existing files. If a command fails halfway through its execution, after modifying certain files, simply exiting with an error code would leave the project in an inconsistent state. To provide graceful recovery from errors, Bocca uses a "smart" file manager that is able to undo any operation on a file performed during the execution of a command. Additionally, the file manager automatically makes backups for all destructive actions, e.g., `rename` and `remove`.

Users can control many aspects of Bocca through the use of a projectwide configuration file. For example, one can specify different levels of validation after user input into SIDL files: from no validation, to running a syntax check, up through regeneration of all affected implementation code.

#### 5.4.3. *Help system*

The `help` action and `--help/-h` options are an important source of documentation for Bocca and are implemented using the Python docstrings embedded in the method implementations. 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 and will not be out of sync with the implementation.

### 5.5. *Build system*

Bocca does not incorporate a specific build paradigm. The definition of multiple different build systems is made possible through a pair of abstract interfaces, *LocationManager* and *Builder*. These interfaces allow different build systems to be plugged into Bocca-managed projects using an option in the project configuration file. As part of the Bocca distribution, a GNU Make [17] plugin is provided. This plugin generates makefile segments that automate the build of all Bocca-managed entities. Users can extend and override make variables and rules using special user-controlled makefile segments that are then included in the automatically generated build files. Users can define rules that are executed before and after each stage of the build: code generation, compilation, linking and installation. In addition to the predefined makefile variables provided by the builder plugin (e.g., installation locations, compilers, compiler and other flags), developers can define new makefile variables that are used in predefined and/or user-defined rules.

## 6. Related work

It is tempting to consider Bocca, albeit commandline driven, just another IDE. Although Boccafulfills many functions traditionally associated with IDEs, it does not provide the language-syntax-enabled editors, visual programming interfaces, and so on that are inseparable from traditional IDEs. It is difficult to draw distinctions between the typical IDE and Bocca, but certainly the IDE is more comprehensive. By taking advantage of language and application domain particulars, the *intention* of an IDE is to speed the development process generically. Traditional IDEs are involved in every aspect of the development. Hence, special-purpose IDE application is *necessary* for all aspects of code creation, testing and deployment. Unique peculiarities in these languages and bindings are exploited to flatten the learning curve and otherwise speed code construction. Bocca, on the other hand, is meant to create a CCA shell application or import existing code into a componentized application and is limited to that purpose. The CCA application developer relies on Bocca only to define and maintain CCA components and SIDL interfaces; the rest of the development process is left to other tools. Pragmatically, a typical IDE takes many more man-years of effort in order to be viable, an effort that is beyond the resources of the Bocca team.

Bocca borrows from the extreme programming philosophy that all working code comes from other working code, and it is easier to evolve a working application than to start out with a blank slate. There is no requirement for Bocca itself to build or edit the application after it has been invoked except to augment or change aspects of its componentization. While typical IDEs are concerned with every aspect of software development, Bocca is concerned only with the CCA componentization. Possibly the closest analogue in commercial software is Ruby on Rails [18]. Given a database schema and a few names, Rails creates a database-backed web application shell that users can modify to suit their needs. Similar to Bocca, does not seek to speed the traditional process of development but to change the process itself by giving the user a working web application. Also similarly, Rails' scope is limited to the web application domain and is not involved in editing, building, or running of the application itself. Users employ their own tools to insert specific implementation code.

Depending on its design, the IDE is a full partner in a developer's code creation means that once undertaken, the IDE is forever the basis of the developer's code base. Traditional IDEs are involved in every aspect of the development meaning that special-purpose IDE application is *necessary* for all aspects of code creation, testing, and deployment. Unless Boccais invoked specifically to alter or augment some component aspect, the application is otherwise edited, built, and debugged without Bocca intervention. Thus, Bocca's "cost of abandonment" is low. There is little penalty for getting started with Bocca, and in the event that Boccabecomes too restrictive, continuing development without it.

## 7. Quantifying Bocca's impact

Quantitative studies of Bocca's impact on developer productivity have not been performed because of the rapid and iterative nature of Bocca 's development. We can, however, provide measures of the complexities managed by counting the artifacts Bocca automatically manages. The statistics in Table 1 refer to the previous example of Section 1.3 choosing an implementation language of C++. These are rough measures of Bocca's effectiveness in reducing the software engineering complexity the scientist must deal with in developing portable component software. The only significant dependence of these results on the choice of implementation language is that the lines of helper code (CCA source) generated are substantially higher to manage object type handling and exceptions in languages lacking those concepts, such as C and Fortran.

We show in Table 1 the quantity and sizes of files and code generated by Bocca use, by execution of the builder plugin, and by installation of the resulting product. For the two components and one port in the example of Fig. 2, the set of customizable files is small, while the glue code is both lengthy and widespread. The majority of the glue code is due to the CCA requirement that components be language agnostic.

Portable HPC software build systems are expensive to create, debug, and maintain. The CCA specification and the Babel tool prescribe no standards for

Table 1
Project entities managed with Bocca

| Files | Lines | Code type |
|---|---|---|
| User customizable | | |
| 3 | 116 | SIDL Files |
| 4 | 1354 | Babel Impl source |
| 16 | 481 | Build system |
| Uneditable | | |
| n/a | 498 | CCA source (inserted into Impl) |
| 94 | 9732 | Babel glue source code |
| 798 | 157,709 | Build process intermediates |
| Installed product | | |
| 26 | n/a | Interface libraries |
| 2 | n/a | Component libraries |
| 70 | 22,513 | Headers or scripts |
| 201 | 4 | Package metadata (xml) |

build processes or installation practices. Bocca fills this void using the pattern set by the builder-plugin of the user's choice, completely hiding the size and complexity of the build infrastructure required to assemble the different makefile fragments into a complete build process.

Several Bocca users have adopted the more radical (and considerably cheaper) approach that the entire build system is only a generated and therefore disposable artifact. These users permanently manage only the files containing their value-added work: the SIDL definitions, the component implementations, the external software dependencies, and the scripts that regenerate the build system.

## 8. Conclusions and future work

The development environment for high-performance computing software presents unique requirements that are not addressed by any of the commonly available integrated development systems. Such requirements include the need to seamlessly support mixed language development, where the same project involves code written in any of the commonly used languages in high performance computing. Additionally, the need to support development and deployment on high-end computing platforms that may lack proper infrastructure for GUI-based development renders the command-line user interface essential for development tools that target such platforms. Furthermore, the fact that HPC code is typically ported to several platforms during its useful life time renders the use of any platform-specific development environment highly undesirable.

The Bocca development tool addresses the unique requirements for high-performance scientific computing software. Bocca facilitates the rapid prototyping of complex multilingual applications, while at the same time providing the required infrastructure for the continued refinement and update of the original application design. Bocca's infrastructure, developed entirely in the widely available Python programming language, guarantees ease of portability to any HPC platform. The default Bocca-generated build system uses standard, portable tools that guarantee maximum portability of Bocca projects across HPC platforms. Furthermore, Bocca generated projects are not dependent on Bocca itself for build and deployment. Thus, such projects can evolve independently of Bocca, should the need arise.

The underlying design philosophy of Bocca maximizes the reuse of widely accepted standard tools in the HPC community, while streamlining the development and deployment process. The pluggable build system back-end architecture of Bocca relies on the integration of widely available build systems and tools, rather than introducing a new build system that locks developers into a specific build methodology that may not fit their expertise or development requirements.

## Acknowledgments

This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, US Department of Energy, under Contract DE-AC02-06CH11357 (Argonne National Laboratory); Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the US Department of Energy under contract DE-AC05-00OR22725. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the US Department of Energy under contract DE-AC04-94AL85000.

## References

[1] 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/tutorial ModFramework.pdf, 2002.

[2] B.A. Allan, R. Armstrong, D.E. Bernholdt, F. Bertrand, K. Chiu, T.L. Dahlgren, K. Damevski, W.R. Elwasif, T.G.W. Epperly, M. Govindaraju, D.S. Katz, J.A. Kohl, M. Krishnan, G. Kumfert, J.W. Larson, S. Lefantzi, M.J. Lewis, A.D. Malony, L.C. McInnes, J. Nieplocha, B. Norris, S.G. Parker, J. Ray, S. Shende, T.L. Windus and S. Zhou, A component architecture for high-performance scientific computing, *Intl. J. High Perform. Comput. Appl.* **20**(2) (2006), 163–202.

[3] 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, *Concurr. Comput.: Pract. Exp.* **14**(2) (2002), 323–345.

[4] Babel homepage, http://www.llnl.gov/CASC/components/babel.html, 2008.

[5] D. Beazley, Simplified wrapper and interface generator (SWIG), http://www.swig.org, 2008.

[6] K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[7] 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, 2004, IEEE Computer Society, 2004.

[8] R.M.-H. Bill Burke, *Enterprise JavaBeans 3.0*, 5th edn, O'Reilly Media, Inc., Sebastopol, CA, 2006.

[9] CCA tutorial hands-on guide: A step-by-step walk-through for creating CCA components, http://www.cca-forum.org/download/tutorial/guide-html-0.5.3_rc1.

[10] CCA tutorials, http://www.cca-forum.org/tutorials/.

[11] Dabeaz LLC, Ply homepage, http://www.dabeaz.com/ply/index.html, 2008.

[12] R. Dick and K. Gaitanis, Graph – Directed and undirected graph data structures and algorithms, http://ziyang.ece.northwestern.edu/ dickrp/python/mods.html, 2005.

[13] J. Ellson, E. Gansner, L. Koutsofios, S.C. North and G. Woodhull, Graphviz – Open source graph drawing tools, in: *Graph Drawing*, LNCS, Vol. 2265, Springer, Berlin/Heidelberg, 2002, pp. 594–597.

[14] W. Elwasif, B. Norris, B. Allan and R. Armstrong, Bocca: A development environment for HPC components, in: *HPC-GECO/CompFrame 2007*, Montreal, Canada, October 2007.

[15] S.R. Kohn, G. Kumfert, J.F. Painter and C.J. Ribbens, Divorcing language dependencies from a scientific software library, in: *10th SIAM Conference on Parallel Processing*, Portsmouth, VA, 2001.

[16] G.T. Leavens and M. Sitaraman, *Foundations of Component-Based Systems*, Cambridge University Press, New York, NY, 2000.

[17] R. Mecklenburg, *Managing Projects with GNU Make*, O'Reilly Media, Inc., Sebastopol, CA, 2004.

[18] D. Thomas, D.H. Hansson, L. Breedt, M. Clark, J.D. Davidson, J. Gehtland and A. Schwarz, *Agile Web Development with Rails*, 2nd edn, Pragmatic Progammer, Lewisville, TX, 2006.