A Hands-On Guide to the Common Component Architecture

The Common Component Architecture Forum Tutorial Working Group

A Hands-On Guide to the Common Component Architecture

by The Common Component Architecture Forum Tutorial Working Group

Published 2007-09-24 21:43:13-04:00 (time this instance was generated)

Licensing Information

Requested Attribution. CCA Forum Tutorial Working Group, *A Hands-On Guide to the Common Component Architecture, version 0.5.2_rc1*, 2007, http://www.cca-forum.org/tutorials/.

Or in BibTeX format:

```
@Manual{hog-cca:0.5.2_rc1,
  title =   {A Hands-On Guide to the Common Component Architecture},
  author =  {The Common Component Architecture Forum Tutorial
             Working Group},
  edition = {0.5.2_rc1},
  year =    2007,
  note =    {http://www.cca-forum.org/tutorials/}
}
```

Table of Contents

Preface

$Revision: 1.19 $
$Date: 2007/09/24 14:33:10 $

The Common Component Architecture (CCA) is an environment for component-based software engineering (CBSE) specifically designed to meet the needs of high-performance scientific computing. It has been developed by members of the Common Component Architecture Forum [http://www.cca-forum.org].

This document is intended to guide the reader through a series of increasingly complex tasks starting from composing and running a simple scientific application using pre-installed CCA components and tools, to writing (simple) components of your own. It was originally designed and used to guide the "hands-on" portion of the CCA tutorial, but we hope that it will be useful for self-study as well.

We assume that you've had an introduction to the terminology and concepts of CBSE and the CCA in particular. If not, we recommend you peruse a recent version of the CCA tutorial presentations [http://www.cca-forum.org/tutorials/] before undertaking to complete the tasks in this Guide.

1. Help us Improve this Guide

If you find errors in this document, or have trouble understanding any portion of it, please let us know so that we can improve the next release. Email us at `<help@cca-forum.org>` with your comments and questions.

2. Finding the Latest Version of the CCA Hands-On Exercises

The hands-on exercises and this Guide are evolving and improving. We will maintain links to the current releases of this Guide, the tutorial code, and accompanying tools at http://www.cca-forum.org/tutorials/#sources. If you want older versions or intermediate "release candidates", follow the links there to the parent download directories to see the full list of available files.

3. Typographic Conventions

- `This font` is used for file and directory names.
- **This font** is used for commands.
- **`This font`** is used for input the user is expected to enter.
- *`This font`* is used for "replaceable" text or variables. Replaceable text is text that describes something you're supposed to type, like a *`filename`*, in which the word "filename" is a placeholder for the actual filename.
- The following fonts are used to denote various programming constructs: `class names` (CCA "components"), `interface names` (CCA "ports"), and `method names`. Also `variable names` and `environment variables` are marked up with special fonts.
- URLs [http://www.cca-forum.org/] are presented in square brackets after the name of the resource they describe in the print version of this Guide.
- Sometime we must break lines in computer output or program listings to fit the line widths available.

In these cases, the break will be marked by a “\” character. In real computer output, you see a long continuous line rather than a broken one. For program listings, unless otherwise indicated, you can join up the broken lines. In shell commands, you can use the “\” and break the input over multiple lines.

4. File and Directory Naming Conventions

Throughout this Guide, we refer to various files and directories, the precise location of which depends on how and where things were built and installed. All such references will be based on a few key directory locations, which will be determined when you build and install the software (Appendix B, *Building the CCA Tools and TAU and Setting Up Your Environment* and Appendix C, *Building the Tutorial Code Tree*). Wherever appropriate, we will write these as environment variables, so that the text in the Guide can simply be pasted into your shell session (assuming your login environment is setup as suggested in Section B.4, “Setting Up Your Login Environment”).

Warning

Note that tools such as the Ccaffeine framework do not expand environment variables. In these cases, you'll need to type in the complete path, substituting the placeholder (i.e. “*`TUTORIAL_SRC`*”) with the actual path.

If you're participating in an organized tutorial, you will be given information separately about the particular paths corresponding to these locations.

`CCA_TOOLS_ROOT` (`$CCA_TOOLS_ROOT`)
: The installation location of the CCA tools. (See Section B.1, “The CCA Tools”.)

`TAU_ROOT` (`$TAU_ROOT`)
: The installation location of the TAU Portable Profiling package. (See Section B.3, “Downloading and Installing TAU”.)

`TAU_CMPT_ROOT` (`$TAU_CMPT_ROOT`)
: The installation location of the TAU performance component. (See Section B.3, “Downloading and Installing TAU”.)

`TUTORIAL_SRC` (`$TUTORIAL_SRC`)
: The location that the `tutorial-src-`*`version`*`.tar.gz` file was unpacked and built. (See Appendix C, *Building the Tutorial Code Tree*.)

`WORKDIR` (`$WORKDIR`)
: This is the location of a working directory, in which you can carry out all of the exercises in this Guide. The basic requirements are that you have write access and sufficient disk space for the work (perhaps 100 MB), and if you're working through the tutorial independently, you can usually choose the *`WORKDIR`* based on your knowledge of the system you're using. However in some cases there may be other criteria as well. For example, organized tutorials are often conducted on a cluster, where all home directories are mounted from a single NFS file server. The load of many tutorial attendees simultaneously trying to build code on an NFS file system, even if they are using different nodes, has been known to kill servers from time to time. In cases like this, it might be better to work in a directory on a disk that is local to node, such as a scratch disk. If you're part of an organized tutorial *please use the `WORKDIR` you are assigned* to help reduce load problems.

5. Acknowledgments

There are quite a few people active in the Tutorial Working Group who have contributed to the general development of the CCA tutorial and this Guide in particular:

People
: Benjamin A. Allan, Rob Armstrong, David E. Bernholdt (chair), Randy Bramley, Tamara L. Dahlgren, Lori Freitag Diachin, Wael Elwasif, Tom Epperly, Madhusudhan Govindaraju, Ragib Hasan, Dan Katz, Jim Kohl, Gary Kumfert, Lois Curfman McInnes, Alan Morris, Boyana Norris, Craig Rasmussen, Jaideep Ray, Sameer Shende, Torsten Wilde, Shujia Zhou

Institutions
: Argonne National Laboratory, Binghamton University - State University of New York, Indiana University, Jet Propulsion Laboratory, Los Alamos National Laboratory, Lawrence Livermore National Laboratory, NASA/Goddard, University of Illinois, Oak Ridge National Laboratory, Sandia National Laboratories, University of Oregon

Computer facilities for the hands-on exercises in this tutorial have been provided by the Computer Science Department and University Information Technology Services of Indiana University, supported in part by NSF Grants CDA-9601632 and EIA-0202048.

Finally, we must acknowledge the efforts of the numerous additional people who have worked very hard to make the Common Component Architecture what it is today. Without them, we wouldn't have anything to present tutorials about!

Chapter 1. Introduction

$Revision: 1.26 $
$Date: 2007/09/25 01:35:35 $

In this Guide, we will take you step by step through a series of hands-on tasks with CCA components in the CCA software environment. The initial set of exercises are based on an example that's intentionally chosen to be very simple from a scientific viewpoint, numerical integration in one dimension, so that we can focus on the issues of the component environment. It may look like overkill to have broken down such a simple task into multiple components, but once you have a basic understanding of how to use and create components, you should be able to extend the concepts to components that are scientifically interesting to you and far more complex.

The exercises are laid out as follows:

- In Chapter 2, *Assembling and Running a CCA Application*, you will use pre-built components to assemble and run several different numerical integration applications.
- In Chapter 3, *Using Bocca: An Application Generator for CCA*, you will construct your own components for the numerical integration example, using the **bocca** tool.

You are strongly advised to at least read and understand Chapter 2, *Assembling and Running a CCA Application* before going on to later exercises. You'll need to use the techniques of Chapter 2, *Assembling and Running a CCA Application* to test the components you write later.

In Chapter 2, *Assembling and Running a CCA Application*, you'll be working with a complete version, pre-built of the tutorial code tree. Then in Chapter 3, *Using Bocca: An Application Generator for CCA* you'll start from scratch to create components on your own, replicating those in Chapter 2, *Assembling and Running a CCA Application*. In this way, the separate complete tutorial code tree can always serve as a reference if you run into problems. Of course if you're working through this Guide as part of an organized tutorial, there should be instructors around who can help you. And if you're working on your own, you can email us for help at `<help@cca-forum.org>`.

1.1. The CCA Software Environment

The CCA is, at its heart, just a specification. There are several realizations of the CCA as a software environment. In this Guide, we use the following tools to provide that software environment, which are currently the most widely used for high-performance (as opposed to distributed) computing using the CCA:

Ccaffeine
: A CCA framework which emphasizes local and parallel high-performance computing, and currently the predominate CCA framework in real applications. For more information, see http://www.cca-forum.org/ccafe/.

Babel
: A tool for language interoperability. It allows components written in different languages to be connected together. The Scientific Interface Definition Language (SIDL) is associated with Babel. For more information, see http://www.llnl.gov/CASC/components/babel.html. Babel uses Chasm for Fortran 90 array support. For more information, see http://chasm-interop.sourceforge.net [http://chasm-interop.sourceforge.net;].

bocca
: A tool for generating and manipulating the skeleton code for components. Bocca is designed to simplify some of the more tedious and mechanical aspects of creating components. (Before bocca, this Guide was a lot longer because we had to take you step by step through writing all of this "boilerplate" code for yourself.)

Many of the commands you will type are specific to the fact that you're using these tools as your CCA software environment. But the components you will use and create are independent of the particular tools being used.

1.2. Where to Go from Here

Before starting the exercises, you'll need to do a little bit of work to set things up. Depending on whether you're working through the Guide on your own or participating in an organized tutorial, this may include getting logged in to a remote system, preparing the CCA environment, and building the tutorial code needed for Chapter 2, *Assembling and Running a CCA Application*.

1. **Getting Connected**

   a. **Organized Tutorial Participant**

      If you're participating in an organized tutorial, you'll probably be using a remote system that's already setup with nearly all of the software you need. You'll be given details for your account, your machine assignment, etc. by the tutorial instructors. That info, together with the notes in Appendix A, *Remote Access for the CCA Environment* should give you sufficient information to get logged in to the remote machine. If you have any problems, ask the tutorial instructors.

   b. **Self-Study User**

      If you're working through the Guide on your own, you may choose to work locally or remotely, depending on the resources you have available. If you're working remotely, you may want to refer to the notes on using the CCA tools remotely in Appendix A, *Remote Access for the CCA Environment*.

2. **Preparing the CCA Environment**

   a. **Organized Tutorial Participant**

      In this case, the CCA tools (Ccaffeine, Babel, and bocca) will already have been built in a common area. You will have to do is insure that your login environment is properly setup to access those tools. This generally involves adding some directories to your `PATH` and setting some other environment variables. Instructions will be included with your account information. Some general notes can be found in Section B.4, “Setting Up Your Login Environment”. If you wish to use the Ccaffeine GUI, you will also need to download it and set it up on your local system. Instructions can be found in Section B.2, “The Ccaffeine GUI”.

   b. **Self-Study User**

      In this case, you will need to download and install the CCA tools (Ccaffeine, Babel, and bocca) and configure your login environment to use them. Instructions can be found in Appendix B, *Building the CCA Tools and TAU and Setting Up Your Environment*. If you wish to use the Ccaffeine GUI and you are working on a remote machine, you will need to download the GUI and set it up on your local system. Instructions can be found in Section B.2, “The Ccaffeine GUI”.

3. **Building the Tutorial Code**

a. **Organized Tutorial Participant**

   Once again, the tutorial code will already have been built in a central location.

b. **Self-Study User**

   You'll also need to download and build the tutorial code tree. Instructions can be found in Appendix C, *Building the Tutorial Code Tree*.

Once you've setup everything as outlined above, you should be ready to proceed to Chapter 2, *Assembling and Running a CCA Application*.

Chapter 2. Assembling and Running a CCA Application

$Revision: 1.10 $
$Date: 2007/09/24 16:14:09 $

In this exercise, you will work with pre-built components from the integrator example to compose several CCA-based applications and execute them. The integrator application is a simple example, designed to illustrate the basics of creating, building, and running component-based applications without scientific complexities a more realistic application would also present. The purpose of this application is to numerically integrate a one-dimensional function. Several different integrators and functions are available, in the form of components. A “driver” component controls the calculation, and for the Monte Carlo integrator, a random number generator is also required. The specific components available are:

Drivers:	`drivers.CXXDriver`, `drivers.F90Driver`, `drivers.PYDriver`
Integrators:	`integrators.MonteCarlo`, `integrators.Simpson`, `integrators.Trapezoid`
Functions:	`functions.CosFunction` (cos(x), which integrates to sin(1), or roughly 0.841), `functions.CubeFunction` (x^3, which integrates to 0.25), `functions.LinearFunction` (x, which integrates to 0.5), `functions.PiFunction` ($4/(1+x^2)$, which integrates to pi), `functions.QuinticFunction` (x^5-4x^4, which integrates to 1/6 - 4/5, or roughly -0.633), `functions.SquareFunction` (x^2, which integrates to 1/3)
Random Number Generators:	`randomgens.RandNumGenerator` (required by `integrators.MonteCarlo`)

The Ccaffeine framework provide three different ways for users to interact with it in order to assemble and run CCA applications. You can type commands in yourself at the framework's prompt, execute a script containing those same commands, or use a graphical user interface. The graphical approach is the easiest for most people to get a feel for how components work, so we will start with that (Section 2.1, “Using the GUI Front-End to Ccaffeine”) and later discuss how actions in the GUI map onto instructions in a script (see Section 2.2, “ Running Ccaffeine Using an `rc` File ”).

In practice, most users set the GUI interface aside after they become more comfortable with the CCA environment in favor of the scripting approach. That's especially true once they've developed a bunch of components and want to run simulations with them in batch jobs, where GUIs tend not to be so convenient. Of course it is entirely up to you which approach you use in the long run.

Note

This exercise uses the `tutorial-src` code tree. If you are participating in an organized tutorial, the tree will have been built for you in advance, and the location will be noted on your account information handout. If you're working through this exercise on your own, you'll need to build the code tree, following the instructions in Appendix C, *Building the Tutorial Code Tree*.

Tip

These exercises can involve a fair amount of typing. You may find it convenient to use the online HTML version of this Guide (at http://www.cca-forum.org/tutorials/#sources to cut and paste the necessary inputs. Note, however, that not everything can be cut-and-based directly. Take particular care with lines that had to be broken for purposes of documentation, and for placeholder values such as "*TUTORIAL_SRC*".

2.1. Using the GUI Front-End to Ccaffeine

There is a graphical front-end for Ccaffeine (known as ccafe-gui, or "the GUI") which provides a fairly simple visual programming metaphor for the assembly of applications using CCA components. The current GUI is a Java tool, making it quite portable. It can also be used over network connections, so that you can run it on your local machine to create and run applications on a computer somewhere else. In this exercise, we'll use the Ccaffeine GUI to assemble and run several different "applications" using the components already available in the `tutorial-src` tree.

Ccaffeine and its GUI are run as two separate processes, possibly on two different machines. Depending on the specific circumstances, there are a variety of ways to invoke the GUI and the Ccaffeine framework. Bocca generates two helper scripts in the `utils` subdirectory of a project, which will serve most purposes. Which to use depends on whether the graphical display you're using (the "*GUI host*") is directly attached to the machine on which you're running the framework (the "*Ccaffeine host*"), or whether they're separated by a network link.

2.1.1. Tools to Use when *GUI host* and *Ccaffeine host* are Identical

When you're working on a display that is directly attached to the *Ccaffeine host*, the bocca-generated **utils/run-gui.sh** script is the simplest one to use. It requires no arguments, and it automatically initializes the framework with a *palette* consisting of all of the components in the bocca project.

Note

While the GUI can be run remotely, using the X11 protocol to display on your local X11 server, this is generally unacceptably slow because of the way Java handles graphics in X11. You will probably get more satisfactory performance if you can run the GUI on your local system and allow it to connect over the network to the remote host where you're running Ccaffeine. Tunneling such a connection through an ssh session provides a straightfoward way to deal with intervening firewalls.

In this exercise, you will need to execute **`$TUTORIAL_SRC`/utils/run-gui.sh** in order to launch the front-end GUI and back-end framework with the pre-built components. In later exercises, you should be sure to invoke the **utils/run-gui.sh** script that corresponds to the bocca project you're working on.

2.1.2. Tools to Use when *GUI host* and *Ccaffeine host* are Separate

When working over the network, it is more effective to launch the GUI locally (since it is Java, it will work on Windows platforms as well as Mac, Linux, and unix) and simply transmit text commands over the network. This is the approach we generally use for organized tutorials, with the *Ccaffeine host* on a remote cluster, and your own laptop or another machine serving as the *GUI host*. Obviously this mode will require the use of two separate commands, one to launch the GUI and the other to launch the Ccaffeine framework. The bocca-generated **utils/bocca-gui-backend.sh** script can be used to launch the framework, while the **simple-gui.sh** (**simple-gui.bat** in Windows) script in the CCA tools installation launches the GUI.

The framework should be launched first, and must be told what port to listen on for the GUI connection: **utils/bocca-gui-backend.sh --port *`port_num`***. Typically, it can be any port number between 1025 and 65535 *that doesn't conflict with another application (CCA or any other) wanting to use the same port*. In an organized tutorial, the likelihood of collisions is fairly high, so you will be assigned a port number in that case. The script automatically initializes the framework with a *palette* consisting of all of the components in the bocca project.

The **simple-gui.sh** command (**simple-gui.bat** for Windows) are used to launch the GUI. Though they are provided as part of the CCA tools installation, you must have a copy of them, and the GUI's `jar` file on your display machine, following the directions in Section B.2, “The Ccaffeine GUI”. To make the connection, the script must be told both the hostname and the port number to connect to the framework you just launched: **simple-gui.sh --port *`port_num`* --host *`backend_host`*** (equivalently for Windows).

Tip

If you invoke the **simple-gui.sh** (**simple-gui.bat**) script without arguments, the GUI will pop up a dialog box asking you to specify the hostname and port number to connect to. Filling in these dialogs quickly gets tedious, so you're better off using the command line. (In Windows, launch a **Command Prompt** window, and change directories to wherever you put **simple-gui.bat** and the GUI `jar` file.) In both Windows and most Linux/unix shells, you can simply use the **Up Arrow** key to recall the previous command to be executed again.

Tip

We have on occasion observed problems with the ccafe-gui interface hanging (most often while populating the *palette* as the GUI starts up). This seems to happen less often with Java version 1.4 than with more recent versions. If you're experiencing such problems, you might try switching to the latest Java 1.4 release.

Note

Connections between the GUI and the framework can be tunneled through an ssh connection in order to pass through firewalls between the *GUI host* and the *Ccaffeine host*. For more information, see Appendix A, *Remote Access for the CCA Environment* and in particular Section A.3, “Tunneling other Connections through SSH”.

In this exercise, you will need to execute **`$TUTORIAL_SRC`/utils/bocca-gui-backend.sh --port *`port_num`*** in order to launch the back-end framework with the pre-built components. In later exercises, you should be sure to invoke the **utils/bocca-gui-backend.sh** script that corresponds to the bocca project you're working on. The **simple-gui.sh --port *`port_num`* --host *`backend_host`*** invocations will remain the same throughout.

Other Ways to Launch the GUI and Ccaffeine

As your usage of the CCA becomes more sophisticated, you're likely to encounter situations where the bocca-generated helper scripts don't do exactly what you want. For example, you may need to use a different `rc` file to initialize the framework. Is is therefore worth mentioning a couple of the underlying tools, which are part of the CCA tools distribution.

gui-backend.sh	This command underlies **utils/bocca-gui-backend.sh**. The difference is that **gui-backend.sh** requires an additional argument to specify the `rc` file to initialize the framework, `--ccafe-rc` *`rc_file`*.
gui.sh	This command is equivalent to **simple-gui.sh**, but can be used on a machine with the CCA tools installed without needing to worry about where the GUI's `jar` file is.

2.1.3. Assembling and Running an Application Using the GUI

For the purposes of this exercise, we will assume that you are working in and environment in which *GUI host* and *Ccaffeine host* are separate machines. If they are the same, you can use **utils/run-gui.sh** as described in Section 2.1.1, “Tools to Use when *GUI host* and *Ccaffeine host* are Identical” instead of the first two steps, below.

1. Run **`$TUTORIAL_SRC`/utils/bocca-gui-backend.sh --port *`port_num`*** on the *Ccaffeine host* using the appropriate port.

   In the *Ccaffeine host* terminal window, you will see something like:

   ```
   (Ccaffeine host)
   my rank: -1, my pid: 9625
   Type: Server
   ```

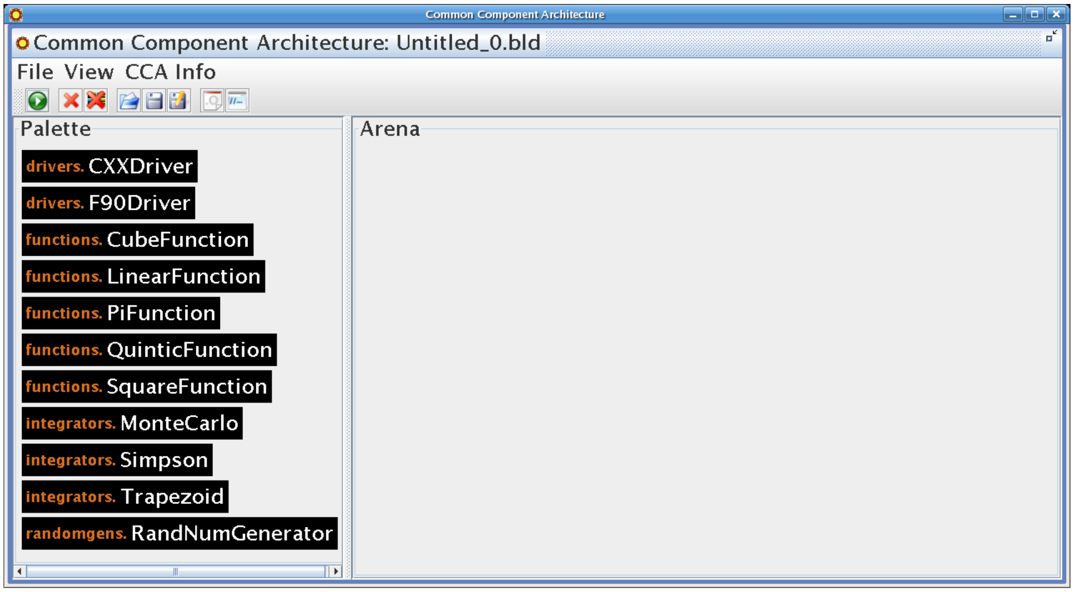
2. Run **simple-gui.sh --port *`port_num`* --host *`backend_host`*** (**simple-gui.bat** on Windows) on the *GUI host*.

   Once the GUI connects to Ccaffeine, Ccaffeine begins running the `rc` file it was invoked with. In the *GUI host* terminal window, you first see some startup messages from the GUI itself, followed by a series of messages as Ccaffeine processes the `rc` file and the GUI displays the results. These are debugging messages and can largely be ignored.

   In the *Ccaffeine host* terminal, you should see some additional messages as Ccaffeine processes the `rc` file, like:

   ```
   (Ccaffeine host)
   CCAFFEINE configured with spec (0.8.2) and babel (1.0.4).

   CCAFFEINE configured with classic (0.5.7).

   CCAFFEINE configured without neo and neo components.
   CmdLineClient parsing ...

   CmdContextCCA::initRC: Found components/tests/test_gui_rc.
   # There are allegedly 11 classes in the component path
   ```

   Finally, in the *GUI host* window, you should see some output associated with the GUI's initialization process, and the GUI itself should have appeared on your display, looking something like this:



Tip

The default layout has the *palette* area fairly narrow. You can click-and-drag on the bar separating the *palette* and the *arena* to adjust the width.

Note

You may see additional components in your *palette*, as we try to expand the variety of examples we provide in the `tutorial-src`.

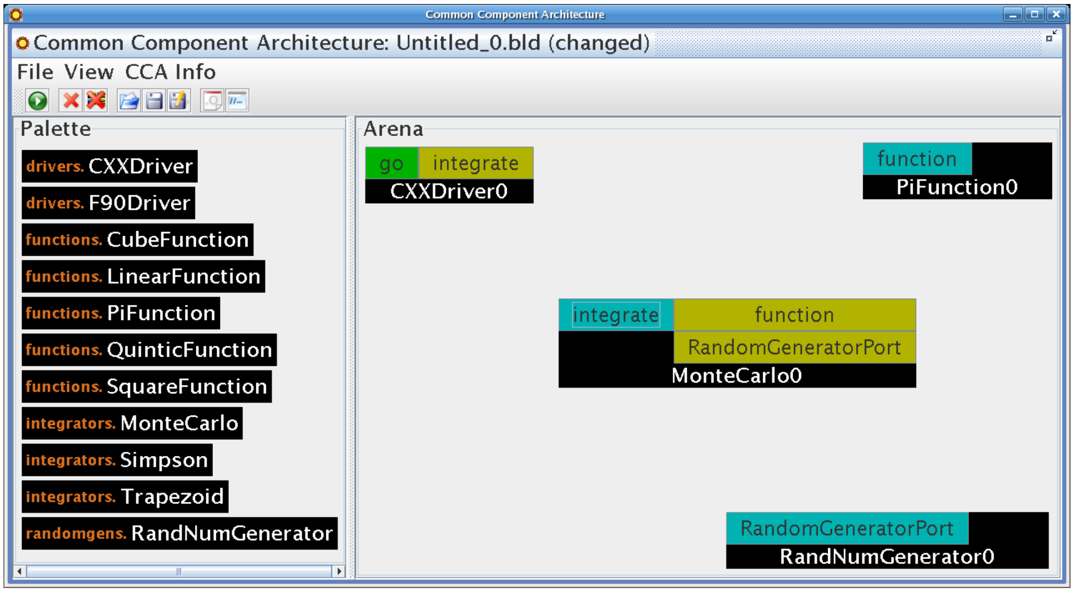
As mentioned above, the `test_gui_rc` sets up the **path** and loads the framework's *palette* with a set of available components. `rc` files are explained in detail in Section 2.2, “ Running Ccaffeine Using an `rc` File ”.

3. We will begin by instantiating a `drivers.CXXDriver` component. Click-and-drag the component you want from the *palette* to the *arena*. When you release the mouse button in the *arena*, a dialog box will pop up prompting you to name this instance of the component. The default will be the last part of the component's class name (i.e. `CXXDriver` for `drivers.CXXDriver`) with a numerical suffix to insure the name is unique. The suffix starts at 0 and simply counts up according to the number of instances of that component you've created in that session. You can, of course, enter any instance name you like, as long as it is unique across all components in the *arena*, but for simplicity, we will always accept the default value in this Guide.

4. For the first application, follow the same procedure to instantiate:

   - `drivers.CXXDriver`,
   - `functions.PiFunction`,
   - `integrators.MonteCarlo`,
   - `randomgens.RandNumGenerator`,

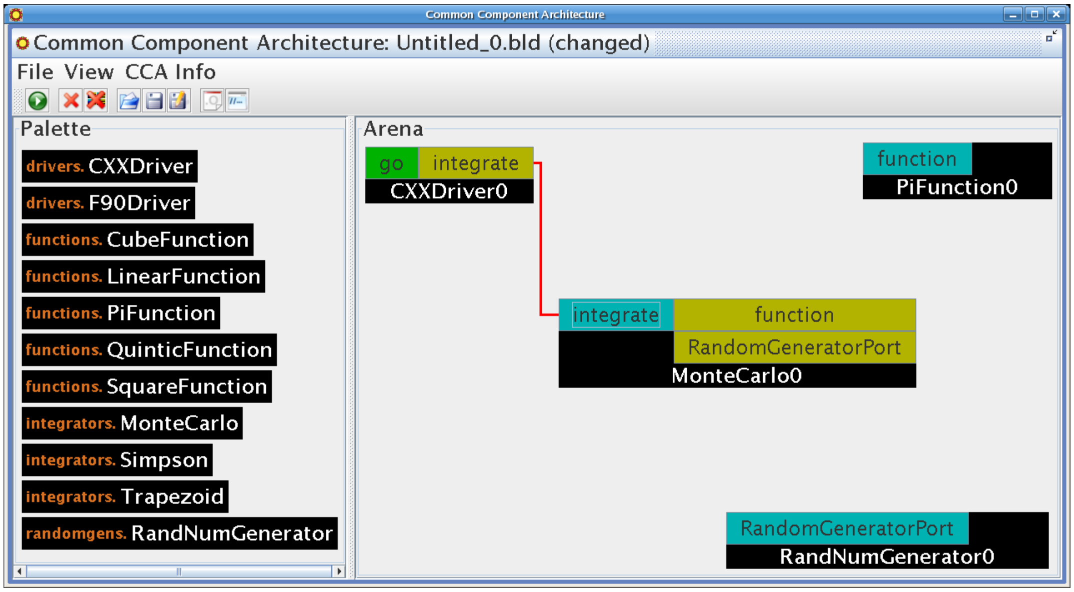
   (you may notice some debugging messages in the *GUI host* terminal window as you do this), and your GUI should look something like this:



Tip

You can drag components around the *arena* to arrange them as suits you -- just click on the black area of the compoment and drag it to the new location. The positions have no bearing on the operation of the GUI or your application.

5. The next step is to begin making connections between the ports of your components. Click-and-release `CXXDriver0`'s `integrate` *uses* port, then click-and-release `MonteCarlo0`'s `integrate` *provides* port and a red line should be drawn between the two:



Tip

If you hover the cursor over a particular port on a component, a “tool tip” box will pop up with the port's name and type based on the arguments to the `addProvidesPort` or `registerUsesPort` calls in the component's `setServices` method. This can be useful for double checking to make sure you're connecting matching ports.

Also notice that when you hover over a particular port (either *uses* or *provides*), matching ports of the opposite type (either *provides* or *uses*) will be highlighted.

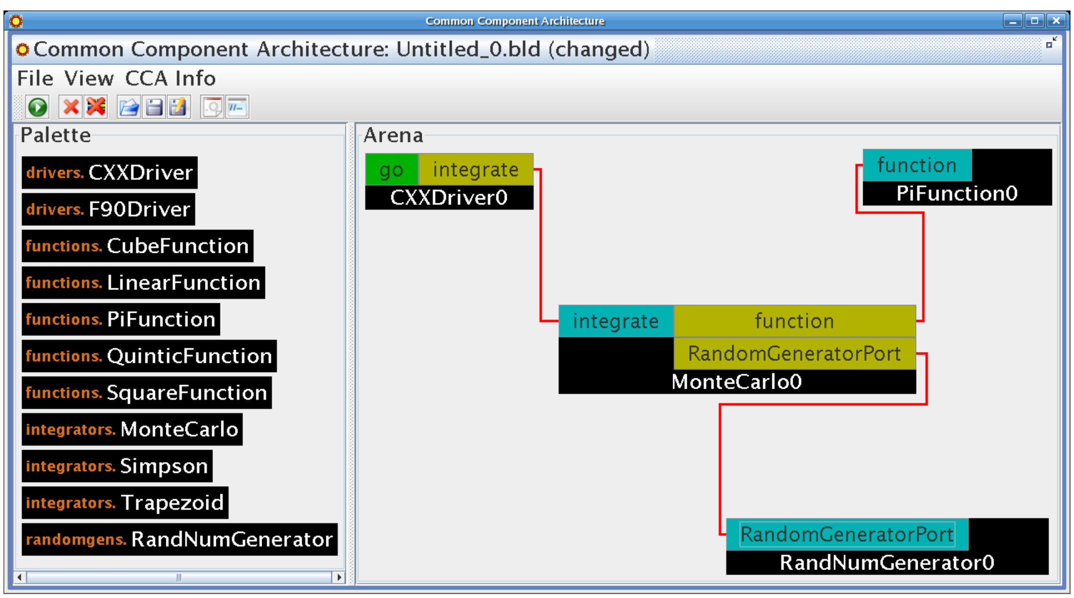
Note

You can move components around even after their ports are connected -- the connections will automatically rearrange. There is no harm in connections crossing each other, nor in connections passing behind other components (though of course they may make it harder to interpret the “wiring diagram” correctly).

6. Complete the first application by making the following connections:

   - `CXXDriver0`'s `integrate` to `MonteCarlo0`'s `integrate`
   - `MonteCarlo0`'s `function` to `PiFunction0`'s `function`
   - `MonteCarlo0`'s `RandomGeneratorPort` to `RandNumGenerator0`'s `RandomGeneratorPort`.

   At this point, your GUI should look something like:



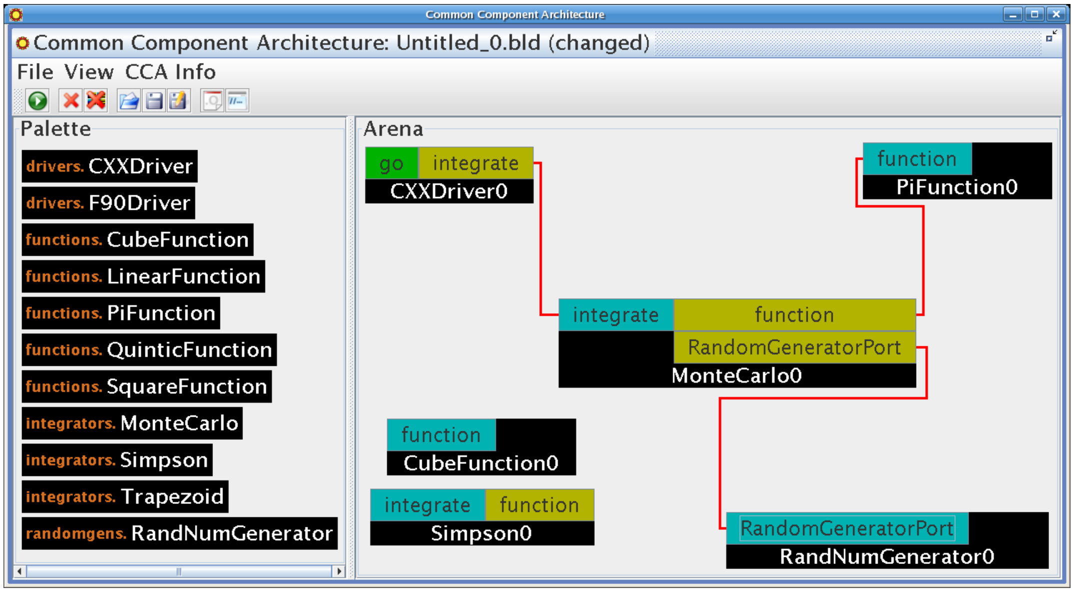
7. The application is now fully assembled and is ready to run. If you click-and-release the `go` button on the `CXXDriver0` component, you should see the result appear in the *Ccaffeine host* terminal, “`Value = 3.139160`” (since Monte Carlo integration is based on random sampling, you will not get exactly the same result every time you run it, but for this example, it should always be reasonbly close to pi) and the message “`IN: ##specific go command successful`” in the

GUI host terminal.

8. Next, we're going to use some of the other components to assemble a different application using the

   - `integrators.Simpson` and
   - `functions.CubeFunction`

   components. Since they're already in the *palette*, you can instantiate them in the same way as Step 3.



Tip

As we've mentioned, wiring diagrams can become hard to interpret when they become cluttered, as is the case with the screen shot above. To help interpret the diagram, remember the following:

- “Wires” only connect to the *sides* of ports -- on the left side of *provides* ports (on the left side of the component), or on the right side of *uses* ports. Connections are never made to the top or bottom of a component.
- The GUI's wire-drawing algorithm is aware only of the two components that are being connected. It will make no attempt to avoid other components or other wires. So wires can pass behind components without connecting to any of their ports, and wires may overlap.
- If you're still uncertain how to interpret the connections try rearranging the components slightly. Connections attached to the component will follow as you drag it around, but others not associated with that component will remain unchanged.

9. Next, we break the port connections we don't need so we can reconnect to the new components. Right-click on the `integrate` (either the *user* or the *provider*) and a dialog box will pop up asking you to confirm that you want to break the connection. (A bug in the GUI causes this dialog box

to appear twice sometimes. Just respond appropriately both times.) You will need to break the following connections:

- `CXXDriver0`'s `integrate` to `MonteCarlo0`'s `integrate`
- `MonteCarlo0`'s `function` to `PiFunction0`'s `function`

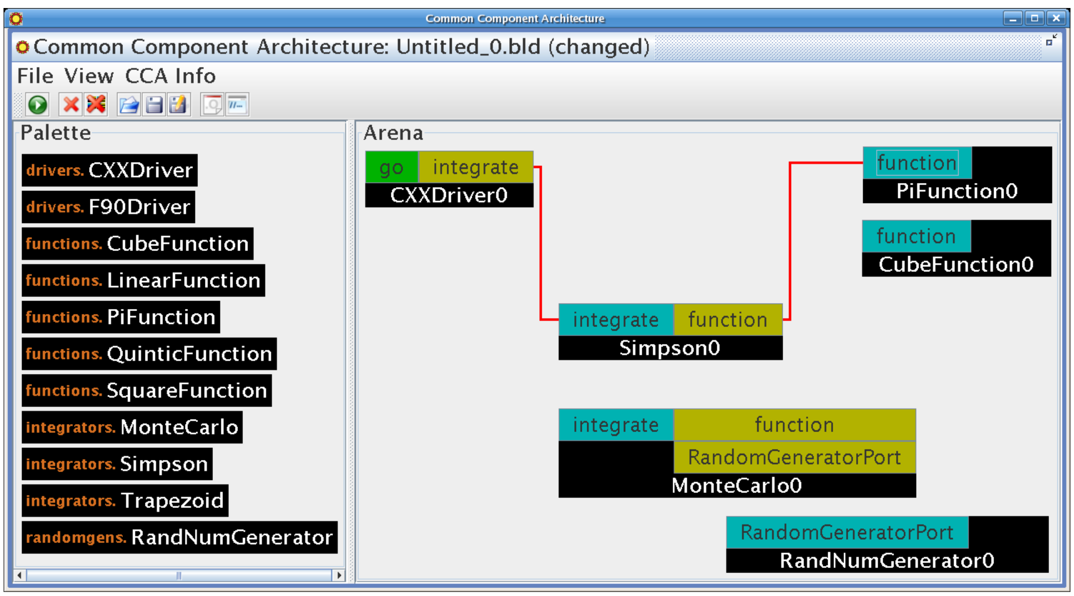
Whether or not `MonteCarlo0` remains connected to `RandNumGenerator0` is immaterial because neither component is connected to any other component in the *arena* and so will no be involved when a disjoint assembly of components is executed.

Note

Step 8 and Step 9 could have been done in either order.

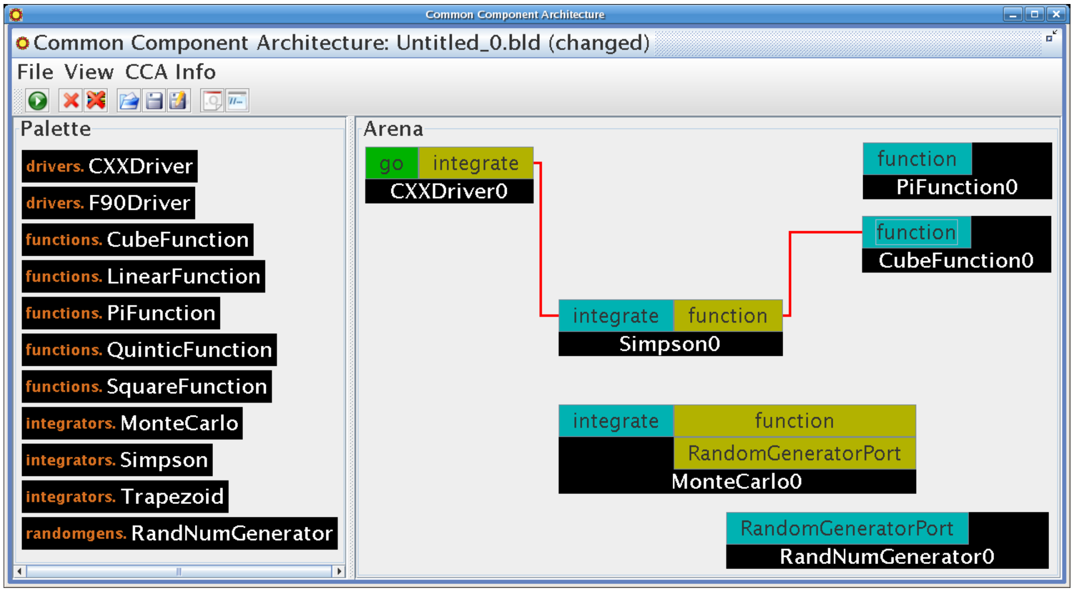
10. Assemble the new application by making the following connections:

    - `CXXDriver0`'s `integrate` to `Simpson0`'s `integrate`
    - `Simpson0`'s `function` to `PiFunction0`'s `function`



Click-and-release the `go` button on the `CXXDriver0` component, you should see the result appear in the *Ccaffeine host* terminal, “`Value = 3.141593`” and the message “`IN: ##specific go command successful`” in the *GUI host* terminal.

11. Finally, create a third application by replacing `PiFunction0` with `CubeFunction0`. When you click on the `go` you should get “`Value = 0.250000`” in the *Ccaffeine host* terminal (with a deterministic integrator, the result should be repeatable).



12. At this point, you should understand how to instantiate components, how to connect and disconnect their ports, and how to execute the application with the `go` port. Feel free to use any and all of the components available in the *palette* to experiment with other integration applications.

Note

Observe that as a user of CCA components, you have no idea what language each component is implemented in. (Admittedly, the names of the drivers are suggestive of the implementation language, but those names were chosen at the convenience of the component developer, and they provide no guarantees regarding the components' implementations.) The language interoperability features of Babel allow components to be hooked together regardless of implementation language with complete transparency.

13. To politely exit the GUI, select File → Quit. This will terminate both the GUI and the backend **ccafe-client** sessions.

Tip

If you've used the GUI to setup and start a long-running simulation, and you don't want to leave the GUI running continuously, you can use the File → Detach option to close the GUI but leave the backend running. *However it is currently impossible to reattach to a running session.*

2.2. Running Ccaffeine Using an `rc` File

In practice, most people don't use the GUI all the time. And even die-hard GUI users will sometimes need to modify the `rc` file that does the initialization. Ccaffeine will also accept commands interactively or in the form of a script (the `rc` file). This capability is very useful when you simply want to run CCA-based applications that you already know how to assemble. In this section, we will examine in detail an `rc` file that does everything you did in the GUI in the previous section.

When we're not using the GUI, the Ccaffeine invocation is much simpler, and there is no need for the helper scripts we used before (**utils/bocca-gui-backend.sh** or **gui-backend.sh**). For direct use, Ccaffeine can be invoked as **`ccafe-single`** or **`ccafe-batch`**, depending on whether you're using it in a single-process (i.e. sequential) interactive situation, or in non-interactive situations, including parallel jobs.

1. Change directories to your *`WORKDIR`* or another place write in, so that we can capture the output of running the `$TUTORIAL_SRC/components/tests/task0.rc` `rc` file.

   Execute the command

   ```
   ccafe-single --ccafe-rc $TUTORIAL_SRC/components/tests/task0.rc \
                >& task0.out
   ```

   (assuming you're using the **csh** or **tcsh** shells; if you're using the **sh** or **bash** shells, replace the output redirection “**`>& task0.out`**” with “**`> task0.out 2>&1`**”).

   The `rc` file is a simple script interpreted by the Ccaffeine framework. It allows components to be instantiated and destroyed, and for ports to be connected and disconnected. The **utils/bocca-run-guibackend.sh** (or **utils/run-gui.sh**) script you used in the previous procedure to launch the framework automatically included a simpler `rc` file (`$TUTORIAL_SRC/components/tests/guitest.gen.rc`) that merely sets the component search path and makes the project's components available on the the *palette*, leaving you to actually instantiate and connect up components in the GUI.

   View the `task0.out` file satisfy yourself that the script ran. (Of course you can also view the script itself if you want.) Below we'll work our way through each section of the script and the corresponding output, but it may help you to see the input and output in their entirety. The step numbers should correspond to the steps in the preceeding GUI procedure.

2. The beginning of the `task0.rc` script looks like this:

   ```
   #!ccaffeine bootstrap file.
   # ------- don't change anything ABOVE this line.-------------

   # Step 2

   path
   path set /home/csm/bernhold/proj/cca/tutorial/tutorial/src-acts07/components/li
   path

   palette
   repository get-global drivers.CXXDriver
   repository get-global drivers.F90Driver
   repository get-global functions.CubeFunction
   repository get-global functions.LinearFunction
   repository get-global functions.QuinticFunction
   repository get-global functions.SquareFunction
   repository get-global integrators.MonteCarlo
   repository get-global integrators.Simpson
   repository get-global integrators.Trapezoid
   repository get-global randomgens.RandNumGenerator
   palette
   ```

The `rc` file begins with a “magic” line (a structured comment) indicating that the script is meant to be processed by Ccaffeine. Ccaffeine expect to find such a line at the beginning of all `rc` files.

Ccaffeine uses a “path” to determine where it should look for CCA components (specifically the `.cca` files, which internally point to the actual libraries that comprise the component). The `rc` file prints the path before and after setting the path for pedagogical reasons. In “real” scripts, you might want to print the path out for debugging or documentation purposes.

Path-related commands in Ccaffeine include:

path Prints the current path.

path append Adds a directory to the end of the current path.

path init Sets the path from the value of the `$CCA_COMPONENT_PATH` environment variable.

path prepend Adds a directory to the beginning of the current path.

path set Sets the path to the value provided.

Ccaffeine also has the concept of a *palette* of components from which applications can be assembled. Unlike a typical unix shell, where putting an executable into your path means you can use it directly, Ccaffeine has a two step process. Components in the path can be added to the *palette* using the command **repository get-global *`class_name`***, where *`class_name`* is the component's class name. This two step approach gives you a little more control when there are large numbers of components in your path. However in this case, we've simply loaded all of the components in the `tutorial-src` tree.

The **palette** commands before and after the block of **repository** commands is simply meant to illustrate that the framework's *palette* starts empty, and ends up with the components you requested. They aren't needed in a “real” script.

The output from these commands should look something like this:

```
CCAFFEINE configured with spec (0.8.2) and babel (1.0.4).

CCAFFEINE configured with classic (0.5.7).

CCAFFEINE configured without neo and neo components.
my rank: -1, my pid: 27566
Type: One Processor Interactive


CmdContextCCA::initRC: Found task0_rc.



pathBegin
pathEnd! empty path.

# There are allegedly 11 classes in the component path

pathBegin
pathElement /home/csm/bernhold/proj/cca/tutorial/tutorial/src-acts07/components
```

```
pathEnd

Components available:

Loaded drivers.CXXDriver NOW  GLOBAL .

Loaded drivers.F90Driver NOW  GLOBAL .

Loaded functions.CubeFunction NOW  GLOBAL .

Loaded functions.LinearFunction NOW  GLOBAL .

Loaded functions.QuinticFunction NOW  GLOBAL .

Loaded functions.SquareFunction NOW  GLOBAL .

Loaded integrators.MonteCarlo NOW  GLOBAL .

Loaded integrators.Simpson NOW  GLOBAL .

Loaded integrators.Trapezoid NOW  GLOBAL .

Loaded randomgens.RandNumGenerator NOW  GLOBAL .

Components available:
drivers.CXXDriver
drivers.F90Driver
functions.CubeFunction
functions.LinearFunction
functions.QuinticFunction
functions.SquareFunction
integrators.MonteCarlo
integrators.Simpson
integrators.Trapezoid
randomgens.RandNumGenerator
```

Note

`rc` files used to initialize the GUI should contain *only* the magic line, **path** and **repository get-global** commands. You can view `$TUTORIAL_SRC/components/tests/gui-setup.rc` as an example.

3. Next we instantiate the components we're going to use to assemble our first application, to place them in the *arena*:

```
# Steps 3-4

instances
instantiate drivers.CXXDriver CXXDriver0
instantiate functions.PiFunction PiFunction0
instantiate integrators.MonteCarlo MonteCarlo0
instantiate randomgens.RandNumGenerator RandNumGenerator0
instances
```

The command syntax is **instantiate *`class_name instance_name`***. (The plain **instantiate** commands before and after are, once again, for pedagogical purposes, to list the contents of the *arena*.) The component's *`class_name`* is set in the SIDL file where it is defined, and is also used in the **repository get-global** command. The *`instance_name`* is chosen by the user, and must simply be unique within the *arena*. You may remember that the GUI suggests a default *`instance_name`* when prompting you for it, but that's a feature of the GUI, not the framework. Here you have to enter it yourself. It happens that we've used the same thing that the GUI would suggest.

The output from these commands should look something like this:

```
FRAMEWORK of type Ccaffeine-Support

CXXDriver0 of type drivers.CXXDriver
successfully instantiated

PiFunction0 of type functions.PiFunction
successfully instantiated

MonteCarlo0 of type integrators.MonteCarlo
successfully instantiated

RandNumGenerator0 of type randomgens.RandNumGenerator
successfully instantiated

CXXDriver0 of type drivers.CXXDriver
FRAMEWORK of type Ccaffeine-Support
MonteCarlo0 of type integrators.MonteCarlo
PiFunction0 of type functions.PiFunction
RandNumGenerator0 of type randomgens.RandNumGenerator
```

4. Now we need to connect up the ports on the components we've instantiated in order to assemble the application:

```
# Steps 5-6

display chain
display component MonteCarlo0
connect CXXDriver0 integrate MonteCarlo0 integrate
connect MonteCarlo0 function PiFunction0 function
connect MonteCarlo0 RandomGeneratorPort RandNumGenerator0 RandomGeneratorPort
display chain
```

The command syntax is **connect *`user_component user_port provider_component provider_port`***.

The **display** command provides various kinds of information about the *arena* and components therein. **display chain** details the connections between components. **display component *`component_instance`*** lists the *uses* and *provides* ports the component has registered.

The output from these commands should look something like this:

```
Component CXXDriver0 of type drivers.CXXDriver
Component FRAMEWORK of type Ccaffeine-Support
Component MonteCarlo0 of type integrators.MonteCarlo
Component PiFunction0 of type functions.PiFunction
Component RandNumGenerator0 of type randomgens.RandNumGenerator

------------------------------------
```

```
Instance name: MonteCarlo0
Class name: integrators.MonteCarlo
------------------------------------
UsesPorts registered for MonteCarlo0

0. Instance Name: function Class Name: function.FunctionPort
1. Instance Name: RandomGeneratorPort Class Name: randomgen.RandomGeneratorPort
------------------------------------
ProvidesPorts registered for MonteCarlo0

Instance Name: integrate Class Name: integrator.IntegratorPort
------------------------------------

CXXDriver0))))integrate---->integrate((((MonteCarlo0
connection made successfully

MonteCarlo0))))function---->function((((PiFunction0
connection made successfully

MonteCarlo0))))RandomGeneratorPort---->RandomGeneratorPort((((RandNumGenerator0
connection made successfully

Component CXXDriver0 of type drivers.CXXDriver
 is using integrate connected to Port: integrate provided by component MonteCar
Component FRAMEWORK of type Ccaffeine-Support
Component MonteCarlo0 of type integrators.MonteCarlo
 is using function connected to Port: function provided by component PiFunction
 is using RandomGeneratorPort connected to Port: RandomGeneratorPort provided b
Component PiFunction0 of type functions.PiFunction
Component RandNumGenerator0 of type randomgens.RandNumGenerator
```

5. Now that we have a complete application, we can start it by invoking the driver's `go`:

   ```
   # Step 7

   go CXXDriver0 go
   ```

   The command syntax is **go** ***`component_instance`*** ***`port_name`***.

   The output from these commands should look something like this:

   ```
   Value = 3.140205
   ##specific go command successful
   ```

6. Now we use commands we already know to complete the rest of the operations that we previously performed using the GUI:

   ```
   # Step 8

   instantiate integrators.Simpson Simpson0
   instantiate functions.CubeFunction CubeFunction0

   # Step 9

   disconnect CXXDriver0 integrate MonteCarlo0 integrate
   ```

```
disconnect MonteCarlo0 function PiFunction0 function

# Step 10

connect CXXDriver0 integrate Simpson0 integrate
connect Simpson0 function PiFunction0 function
display chain
go CXXDriver0 go

# Step 11

disconnect Simpson0 function PiFunction0 function
connect Simpson0 function CubeFunction0 function
display chain
go CXXDriver0 go
```

The output from these commands should look something like this:

```
Simpson0 of type integrators.Simpson
successfully instantiated

CubeFunction0 of type functions.CubeFunction
successfully instantiated



CXXDriver0))))integrate-\ \-integrate((((MonteCarlo0
connection broken successfully

MonteCarlo0))))function-\ \-function((((PiFunction0
connection broken successfully



CXXDriver0))))integrate---->integrate((((Simpson0
connection made successfully

Simpson0))))function---->function((((PiFunction0
connection made successfully

Component CXXDriver0 of type drivers.CXXDriver
 is using integrate connected to Port: integrate provided by component Simpson0
Component CubeFunction0 of type functions.CubeFunction
Component FRAMEWORK of type Ccaffeine-Support
Component MonteCarlo0 of type integrators.MonteCarlo
 is using RandomGeneratorPort connected to Port: RandomGeneratorPort provided b
Component PiFunction0 of type functions.PiFunction
Component RandNumGenerator0 of type randomgens.RandNumGenerator
Component Simpson0 of type integrators.Simpson
 is using function connected to Port: function provided by component PiFunction

Value = 3.141593
##specific go command successful



Simpson0))))function-\ \-function((((PiFunction0
```

```
connection broken successfully

Simpson0))))function---->function((((CubeFunction0
connection made successfully

Component CXXDriver0 of type drivers.CXXDriver
 is using integrate connected to Port: integrate provided by component Simpson0
Component CubeFunction0 of type functions.CubeFunction
Component FRAMEWORK of type Ccaffeine-Support
Component MonteCarlo0 of type integrators.MonteCarlo
 is using RandomGeneratorPort connected to Port: RandomGeneratorPort provided b
Component PiFunction0 of type functions.PiFunction
Component RandNumGenerator0 of type randomgens.RandNumGenerator
Component Simpson0 of type integrators.Simpson
 is using function connected to Port: function provided by component CubeFuncti

Value = 0.250000
##specific go command successful
```

7. At the end of the `rc` files, it is important to remember to terminate the framework.

```
# Step 13

quit
```

The output from these commands should look something like this:

```
bye!
exit
```

Warning

If your `rc` file ends without a **quit** command, Ccaffeine will leave you in interactive mode rather than terminating and returning you to the shell prompt. If you make this mistake a **Control-c** will interrupt Ccaffeine and return you to the shell prompt.

Feel free to copy `$TUTORIAL_SRC/components/tests/task0.rc` to your workspace, modify it, and run it yourself.

2.3. Notes on More Advanced Usage of the GUI

There are a couple of other features of the GUI and its interaction with the Ccaffeine backend that are worth mentioning.

- The `rc` file used in conjunction with a GUI session need not be limited to **path** and **repository get-global** commands -- it is possible to include all Ccaffeine commands, such as in the script of Section 2.2, “ Running Ccaffeine Using an `rc` File ”. The GUI will display all instantiated components, and all connections between their ports. However, the GUI has no mechanism to *place* the components intelligently in the *arena*, so it just puts them all on top of each other. You can, of course, drag them into more reasonable positions.

- It is possible to save the visual state of the GUI in a "`.bld`" file using the Save or Save As... button. The `.bld` file can be loaded into the GUI and replayed by launching it with the `--buildFile` *`file.bld`* option.

  The syntax of the `.bld` file is similar to that of the `rc` file, but they are *not* interchangeable. The `.bld` file can contain commands to instantiate and destroy components and to connect and disconnect ports, as well as commands to move components within the *arena*, and it can only be interpreted by the GUI. The **path** and **repository get-global** commands must always be in the `rc` file, which is interpreted only by the Ccaffeine backend. Also, Ccaffeine itself does not understand the movement commands of the `.bld` file.

Chapter 3. Using Bocca: An Application Generator for CCA

$Revision: 1.21 $
$Date: 2007/09/25 00:35:35 $

While the CCA specification allows you to create components "by hand", it is much quicker to use an application generator that provides templated code for a components and a build system. Naturally bocca cannot create your implementation for you, but all of the glue code that embodies the CCA is created in a few of commands. The advantage of the this approach is that a lot of build and component defaults have been chosen for you. The downside is that you don't get to pick these defaults for yourself.

3.1. Creating a Bocca Project

If your CCA environment is setup properly (Appendix B, *Building the CCA Tools and TAU and Setting Up Your Environment*) then the **bocca** command is already in your command path and you are ready to go. Find a safe place to begin your bocca project, such as your *`WORKDIR`*:

```
$  cd $WORKDIR

$
```

The first thing to do is to create a project directory within which all of your components and ports will reside. Normally you would choose a relevant project name but for now we will just call it **`myProject`**. Create the project directory now:

```
$  bocca create project myProject --language=LANG

The project was created successfully in /var/tmp/user/myProject
$
```

Here *`LANG`* is the implementation language that your components will default to. Just choose the one of **`c`**, **`cxx`**, or **`f90`** with which you are most comfortable. (The default language can actually be any language that bocca and Babel are configured to support, but currently the Guide includes detailed instructions for only C, C++, and F90.) Now that the project is created, we see that bocca has created a lot of build scaffolding to support the componentized application we will write. The first thing you notice is that bocca has created a directory:

```
$  ls
```

```
myProject
$
```

Feel free to poke around a bit:

```
$  ls myProject/
```

```
BOCCA          README       components    ports      utils
Makefile  __init__.py  configure     project.make
Makefile.custom  buildutils   configure.in  project.make.in
$
```

3.2. Creating Ports and Components

Let's create a component. First make sure that your current working directory is inside the project directory:

```
$  cd myProject; pwd
```

```
/usr/tmp/user/myProject
$
```

It is important to be in the project directory when you invoke **bocca** because it picks up all of the context for your project from there (similar to CVS or Subversion). Go ahead and create the component now:

```
$   bocca create component emptyComponent
```

```
Updating the cxx implementation of component myProject.emptyComponent ...
Updating makefiles...
$
```

You will notice that this will take a little time and that bocca has selected `myProject` as the default package name for `emptyComponent`. Bocca will always default to the project name as the package name for both ports and components. Note we have named our component `emptyComponent` because it has no uses nor provides ports and thus is rather uninteresting. Nonetheless all of the necessary make system scaffolding and code has been generated for the component, including the `setServices` call. Here we use as an example the case where *LANG* is **cxx**:

```
$  ls components/myProject.emptyComponent/
```

```
glue                                    myProject_emptyComponent_Impl.hxx
myProject_emptyComponent_Impl.cxx       myProject_emptyComponent_Impl.hxx.rej
myProject_emptyComponent_Impl.cxx.rej
$
```

Fortran, C, and Python will contain similar files. Here in the `components` directory a new directory, `myProject.emptyComponent` has been created to hold your component. And inside there is the code already generated for the component (again continuing with *LANG* = **cxx**) in the files: `myProject_emptyComponent_Impl.cxx`, `myProject_emptyComponent_Impl.hxx` with some Babel glue code in the `glue` subdirectory.

An Empty Component in Ccaffeine

Although the component you've created can't actually *do* anything useful at this point, it is a valid components, and you can build it and instantiate it in Ccaffeine if you like:

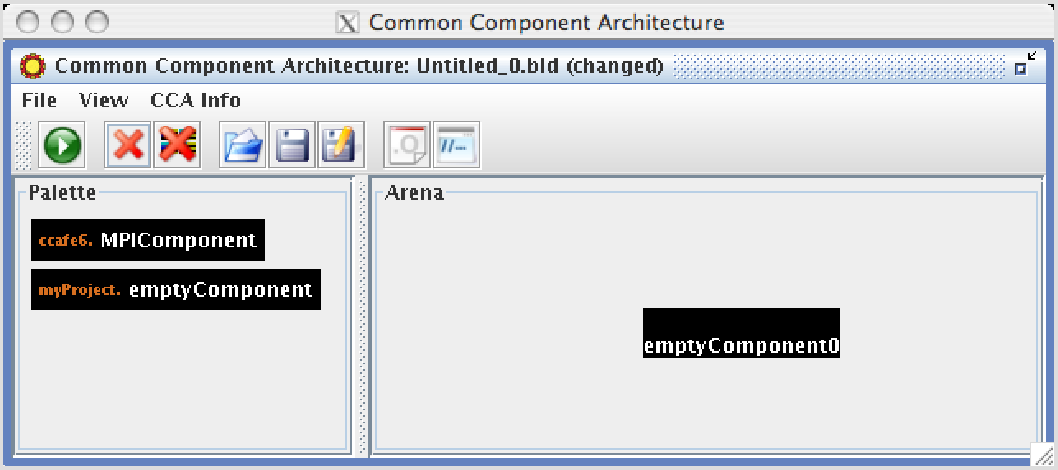
```
$  configure;make
```

```
checking for bash... /bin/sh
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
checking for openpty in -lutil... yes
checking for bocca... /home/rob/cca/bocca/cca-tools-0.6.4_rc8/bin/bocca
configure: Configuring with languages: c cxx f90 python java
configure: Project source dir apparently /var/tmp/user/myProject
configure: Using 1 processe(s) in calls to make.
checking whether make sets $(MAKE)... yes
configure: creating ./config.status
```

```
config.status: creating project.make
config.status: creating buildutils/common_vars.make
config.status: creating utils/run-gui.sh
config.status: creating utils/bocca-gui-backend.sh
config.status: executing outmsg commands

# ==========================================================================
# Building in components/. Available languages: c cxx f90 python java
# ==========================================================================
make[1]: Entering directory `/var/tmp/user/myProject/components'

### Building component myProject.emptyComponent:
- using Babel to generate cxx implementation...  compiling sources...
- creating component library: libmyProject.emptyComponent.la ...
- finished libtooling: components/lib/libmyProject.emptyComponent.la ...
- creating Ccaffeine test script (components/tests/instantiation.gen.rc)...

Build summary:
SUCCESS building myProject.emptyComponent

### To test instantiation of successfully built components, run 'make check' ###
make[1]: Leaving directory `/var/tmp/user/myProject/components'

################ Finished building everything #################
####### You can run some simple tests with 'make check' #######

$
```

(Your output should be substantially similar, but will at least have different paths.)

Now, you can run Ccaffeine and the GUI following the same procedure you used in Section 2.1, “Using the GUI Front-End to Ccaffeine” and you should see something like this:



You can now instantiaate the `emptyComponent`. Of course it lacks any *uses* or *provides* ports and thus cannot be used for anything, but it *is* a full fledged CCA component.

In order to have some exportable or importable functionality in a component we have to have some *uses* and *provides* ports. Bocca will also create the scaffolding and code for ports. Just as in the pre-built application of Chapter 2, *Assembling and Running a CCA Application* we will want to create a

, a `Integrator`, and a `Driver`. Before we can do that we will have to create some ports for these components to *use* and *provide*. Similarly we wish to create a `FunctionPort`, and an `IntegratorPort`:

```
$  bocca create port IntegratorPort
```

```
Updating makefiles...
$
```

```
$  bocca create port FunctionPort
```

```
Updating makefiles...
$
```

Notice that we have opted for the default package `myProject` that is created for us transparently for all components and ports unless otherwise specified. Now, create a set of components similar to those that you used in Chapter 2, *Assembling and Running a CCA Application*, specifying that they will *provide* or *use* the appropriate ports:

```
$  bocca create component Function --provides=FunctionPort:thisFunction
```

```
Updating the cxx implementation of component myProject.Function ...
Updating makefiles...
$
```

```
$  bocca create component Integrator --provides=IntegratorPort:integrate \
    --uses=FunctionPort:integrateThis
```

```
Updating makefiles...
$
```

```
$  bocca create component Driver --uses=IntegratorPort:integrate \
    --provides=gov.cca.ports.GoPort:run
```

```
Updating makefiles...
$
```

This last **`bocca create`** decorates our component with a CCA specified `GoPort` that is not specified as part of this project. Since `gov.cca.ports.GoPort` is a part of the CCA specification, it will be found in the **configure; make** operation. Speaking of which since we have changed a number of things, **configure; make** would be a good thing to do now:

```
$  configure;make
```

```
checking for bash... /bin/sh
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
checking for openpty in -lutil... yes
checking for bocca... /home/rob/cca/bocca/cca-tools-0.6.4_rc8/bin/bocca
configure: Configuring with languages: c cxx f90 python java
configure: Project source dir apparently /var/tmp/user/myProject
configure: Using 1 processe(s) in calls to make.
checking whether make sets $(MAKE)... yes
configure: creating ./config.status
config.status: creating project.make
config.status: creating buildutils/common_vars.make
config.status: creating utils/run-gui.sh
config.status: creating utils/bocca-gui-backend.sh
config.status: executing outmsg commands
make: Warning: File `Makefile.custom' has modification time 4.5e+04 s in the futur

# ==========================================================================
# Building in components/. Available languages: c cxx f90 python java
# ==========================================================================
make[1]: Entering directory `/var/tmp/user/myProject/components'
make[1]: Warning: File `user.make' has modification time 4.5e+04 s in the future
make[2]: Warning: File `/var/tmp/user/myProject/buildutils/common_rules.make' has

### Building component myProject.Driver:
- updating cxx interface and port libraries: myProject.IntegratorPortmake[3]: Warn
make[3]: warning:  Clock skew detected.  Your build may be incomplete.

- using Babel to generate cxx implementation...  compiling sources...
- creating component library: libmyProject.Driver.la ...
- finished libtooling: components/lib/libmyProject.Driver.la ...
- creating Ccaffeine test script (components/tests/instantiation.gen.rc)...
make[2]: warning:  Clock skew detected.  Your build may be incomplete.
```

```
### Building component myProject.Function:
- using Babel to generate cxx implementation...  compiling sources...
- creating component library: libmyProject.Function.la ...
- finished libtooling: components/lib/libmyProject.Function.la ...
- creating Ccaffeine test script (components/tests/instantiation.gen.rc)...

### Building component myProject.Integrator:
- using Babel to generate cxx implementation...  compiling sources...
- creating component library: libmyProject.Integrator.la ...
- finished libtooling: components/lib/libmyProject.Integrator.la ...
- creating Ccaffeine test script (components/tests/instantiation.gen.rc)...

### Building component myProject.emptyComponent: doing nothing -- library is up-to

Build summary:
SUCCESS building myProject.Driver
SUCCESS building myProject.Function
SUCCESS building myProject.Integrator

### To test instantiation of successfully built components, run 'make check' ###
make[1]: warning:  Clock skew detected.  Your build may be incomplete.
make[1]: Leaving directory `/var/tmp/user/myProject/components'

################ Finished building everything #################
####### You can run some simple tests with 'make check' #######

make: warning:  Clock skew detected.  Your build may be incomplete.
$
```

Note that this operation is usually very time-consuming.

Running **make check** will test that the components you've created can be instantiated successfully in the Ccaffeine framework:

```
$  make check
```

```
make -C components check
make[1]: Entering directory `/var/tmp/user/myProject/components'

### Test library load and instantiation for the following languages: c cxx f90 pyt
Running instantiation tests only
###
LDPATH=/home/rob/cca/bocca/cca-tools-0.6.4_rc8/lib:/home/rob/cca/bocca/cca-tools-0
###
CLASSPATH=/home/rob/cca/bocca/cca-tools-0.6.4_rc8/lib/sidl-1.0.6.jar:/home/rob/cca
###
Test script: /var/tmp/user/myProject/components/tests/instantiation.gen.rc
==> Instantiation tests passed for all built components (see /var/tmp/user/myProje
make[1]: Leaving directory `/var/tmp/user/myProject/components'
$
```

If you were to run the GUI (Section 2.1, "Using the GUI Front-End to Ccaffeine") or do the command line equivalent in Ccaffeine (Section 2.2, " Running Ccaffeine Using an `rc` File "), you would find that the components are decorated with the ports you expect, and they can even be connected (an operation of the framework, not of the components or the ports being connected). But of course they have not yet been implemented, so attempting to run an application with these components would cause it to crash.

3.3. Inserting Implementations into Bocca-Generated Components

So far, with very little work, we have generated what appears to be an application but is really just the componentized shell of an application. In order to cause it to do something useful we have to add the implementation. There are two places that we have to change things to make that happen: add methods to the interface definition, or `.sidl` file and then put the implementation code into the components in the language chosen in Section 3.1, " Creating a Bocca Project ". The `.sidl` files are located in the directory `myProject/ports/sidl` and the implementations for the components will be found in the directories `myProject/components/myProject.Driver`, `myProject/components/myProject.Integrator`, and `myProject/components/myProject.Function`.

Where Bocca Puts Your Files and How to Find Out Where They Are

It is not necessarily intuitive that the Driver component code will be found in `myProject/components/myProject.Driver` or that the SIDL for the `IntegratorPort` will be found in `myProject/ports/sidl/myProject.IntegratorPort.sidl`. You can use the **bocca whereis** command to find this information:

```
$  bocca whereis IntegratorPort

./myProject/ports/sidl/myProject.IntegratorPort.sidl
$
```

giving the location of `IntegratorPort`'s SIDL file, and

```
$  bocca whereis -i Driver

myProject/components/myProject.Driver/myProject_Driver_Impl.cxx
$

$  bocca whereis -m Driver

./myProject/components/myProject.Driver/myProject_Driver_Impl.hxx
$
```

Again we are assuming a selection of C++ as the default language for the original project. The `-i` option means return the implementation file path for the component, and the `-m` means to return the implementation header path in C++, C or module path in the case of Fortran. These are the files that component writers would most likely want to edit.

Bocca also has an automated facility for directly editing these files using the `$EDITOR` environment variable:

```
$  bocca edit Driver
```

Will launch the editor indicated in `$EDITOR` and *update the state of the entire bocca project after the edit is complete*. This last bit of automation makes sure that the state of the project is always up to date, however it can cause some problems if you are in the habit of having more than one instance of the editor working on different files. Note that simply editing a file by hand without this **bocca edit** command is possible, for example:

```
$  vi `bocca whereis -i Driver`
```

requires that you type **./configure; make** to bring all of the dependencies back into sync when you are done.

3.3.1. Adding Methods to Ports

First modify the SIDL files to create the `gov.cca.Port`s that are needed to import/export functionality from/to the components, look at the file `myProject/ports/sidl/myProject.IntegratorPort.sidl`.

```
// DO-NOT-DELETE bocca.splicer.begin(myProject.comment)
// Insert-code-here {myProject.comment} (Insert your package comments here)

// DO-NOT-DELETE bocca.splicer.end(myProject.comment)
package myProject version 0.0 {

    // DO-NOT-DELETE bocca.splicer.begin(myProject.IntegratorPort.comment)
    // Insert-code-here {myProject.IntegratorPort.comment} (Insert your port comme

    // DO-NOT-DELETE bocca.splicer.end(myProject.IntegratorPort.comment)
    interface IntegratorPort extends gov.cca.Port
    {
        // DO-NOT-DELETE bocca.splicer.begin(myProject.IntegratorPort.methods)
        // Insert-code-here {myProject.IntegratorPort.methods} (Insert your port m

        // DO-NOT-DELETE bocca.splicer.end(myProject.IntegratorPort.methods)
    }
}
```

Edit `ports/sidl/myProject.IntegratorPort.sidl` to insert the `integrate` method signature:

```
// DO-NOT-DELETE bocca.splicer.begin(myProject.comment)
// Insert-code-here {myProject.comment} (Insert your package comments here)

// DO-NOT-DELETE bocca.splicer.end(myProject.comment)
package myProject version 0.0 {

    // DO-NOT-DELETE bocca.splicer.begin(myProject.IntegratorPort.comment)
    // Insert-code-here {myProject.IntegratorPort.comment} (Insert your port comme

    // DO-NOT-DELETE bocca.splicer.end(myProject.IntegratorPort.comment)
    interface IntegratorPort extends gov.cca.Port
    {
        // DO-NOT-DELETE bocca.splicer.begin(myProject.IntegratorPort.methods)


    double integrate(in double lowBound, in double upBound, in int count);


        // DO-NOT-DELETE bocca.splicer.end(myProject.IntegratorPort.methods)
    }
}
```

Again edit the file `myProject/ports/sidl/myProject.FunctionPort.sidl`:

```
// DO-NOT-DELETE bocca.splicer.begin(myProject.comment)
// Insert-code-here {myProject.comment} (Insert your package comments here)

// DO-NOT-DELETE bocca.splicer.end(myProject.comment)
package myProject version 0.0 {

    // DO-NOT-DELETE bocca.splicer.begin(myProject.FunctionPort.comment)
    // Insert-code-here {myProject.FunctionPort.comment} (Insert your port comment

    // DO-NOT-DELETE bocca.splicer.end(myProject.FunctionPort.comment)
    interface FunctionPort extends gov.cca.Port
    {
        // DO-NOT-DELETE bocca.splicer.begin(myProject.FunctionPort.methods)
        // Insert-code-here {myProject.FunctionPort.methods} (Insert your port met

        // DO-NOT-DELETE bocca.splicer.end(myProject.FunctionPort.methods)
    }
}
```

to add two methods `init` and `evaluate` so that the file looks like this:

```
// DO-NOT-DELETE bocca.splicer.begin(myProject.comment)
// Insert-code-here {myProject.comment} (Insert your package comments here)

// DO-NOT-DELETE bocca.splicer.end(myProject.comment)
package myProject version 0.0 {

    // DO-NOT-DELETE bocca.splicer.begin(myProject.FunctionPort.comment)
    // Insert-code-here {myProject.FunctionPort.comment} (Insert your port comment

    // DO-NOT-DELETE bocca.splicer.end(myProject.FunctionPort.comment)
    interface FunctionPort extends gov.cca.Port
    {
        // DO-NOT-DELETE bocca.splicer.begin(myProject.FunctionPort.methods)



    void   init(in array<double,1> params);
    double evaluate(in double x);



        // DO-NOT-DELETE bocca.splicer.end(myProject.FunctionPort.methods)
    }
}
```

What we have done is place methods into the SIDL files in a language-independent way. When you type:

configure;make

```
checking for bash... /bin/sh
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
checking for openpty in -lutil... yes
checking for bocca... /home/rob/cca/bocca/cca-tools-0.6.4_rc8/bin/bocca
configure: Configuring with languages: c cxx f90 python java
configure: Project source dir apparently /var/tmp/user/myProject
configure: Using 1 processe(s) in calls to make.
checking whether make sets $(MAKE)... yes
configure: creating ./config.status
config.status: creating project.make
config.status: creating buildutils/common_vars.make
config.status: creating utils/run-gui.sh
config.status: creating utils/bocca-gui-backend.sh
config.status: executing outmsg commands

# ========================================================================
# Building in components/. Available languages: c cxx f90 python java
# ========================================================================
```

```
make[1]: Entering directory `/var/tmp/user/myProject/components'

### Building component myProject.Driver:
- updating cxx interface and port libraries: myProject.IntegratorPort
- using Babel to generate cxx implementation...  compiling sources...
- creating component library: libmyProject.Driver.la ...
- finished libtooling: components/lib/libmyProject.Driver.la ...
- creating Ccaffeine test script (components/tests/instantiation.gen.rc)...

### Building component myProject.Function:
- using Babel to generate cxx implementation...  compiling sources...
- creating component library: libmyProject.Function.la ...
- finished libtooling: components/lib/libmyProject.Function.la ...
- creating Ccaffeine test script (components/tests/instantiation.gen.rc)...

### Building component myProject.Integrator:
- using Babel to generate cxx implementation...  compiling sources...
- creating component library: libmyProject.Integrator.la ...
- finished libtooling: components/lib/libmyProject.Integrator.la ...
- creating Ccaffeine test script (components/tests/instantiation.gen.rc)...

### Building component myProject.emptyComponent: doing nothing -- library is up-to

Build summary:
SUCCESS building myProject.Driver
SUCCESS building myProject.Function
SUCCESS building myProject.Integrator

### To test instantiation of successfully built components, run 'make check' ###
make[1]: Leaving directory `/var/tmp/user/myProject/components'

################ Finished building everything #################
####### You can run some simple tests with 'make check' #######

$
```

the methods will be placed into your already generated components in the language you chose when you created the project in Section 3.1, “ Creating a Bocca Project ”. At this point we are ready to insert the implementation into the components. You will now want to jump to the particular language implementation you chose in Section 3.1, “ Creating a Bocca Project ”. There is a section below for each language choice available in bocca. While it is hardly remarkable that we have to resort to a specific programming language to create a component implementation, it is rather surprising that we have acheived all of the foregoing without it.

3.3.2. Language Specific Implementations of the Function, Integrator and Driver Components

3.3.2.1. C++ Implementation

Assumes you created the project with **`bocca create project myProject --language=cxx`**

Look at the file: `myProject/components/myProject.Function/myProject_Function_Impl.cxx` that bocca has generated for you. Find the `evaluate_Impl` method generated from the SIDL definition of `FunctionPort`:

```
/**
 * Method:  evaluate[]
 */
double
myProject::Function_impl::evaluate_impl (
  /* in */double x )
{
  // DO-NOT-DELETE splicer.begin(myProject.Function.evaluate)
  // Insert-Code-Here {myProject.Function.evaluate} (evaluate method)

    // DO-DELETE-WHEN-IMPLEMENTING exception.begin()
    /*
     * This method has not been implemented
     */
    ::sidl::NotImplementedException ex = ::sidl::NotImplementedException::_create(
    ex.setNote("This method has not been implemented");
    ex.add(__FILE__, __LINE__, "evaluate");
    throw ex;
    // DO-DELETE-WHEN-IMPLEMENTING exception.end()

  // DO-NOT-DELETE splicer.end(myProject.Function.evaluate)
}
```

As the comment suggests this method is "not implemented" but some code has been inserted by Babel to make sure an exception is thrown to inform the user if this is called by mistake. Remove this boilerplate so and substitute an implementation for the `PiFunction` (i.e. the integral from 0 to 1 of 4/(1 + x^2>) is pi).

```
/**
 * Method:  evaluate[]
 */
double
myProject::Function_impl::evaluate_impl (
  /* in */double x )
{
  // DO-NOT-DELETE splicer.begin(myProject.Function.evaluate)


  return 4.0 / (1.0 + x * x);


  // DO-NOT-DELETE splicer.end(myProject.Function.evaluate)
}
```

Now in the same file find the second method for `FunctionPort`: `init`:

```
/**
```

```
 * Method:  init[]
 */
void
myProject::Function_impl::init_impl (
  /* in array<double> */::sidl::array<double> params )
{
  // DO-NOT-DELETE splicer.begin(myProject.Function.init)


  // Do nothing.


  // DO-NOT-DELETE splicer.end(myProject.Function.init)
}
```

Here we don't have any initialization so we just eliminate the code that throws the exception for running the method.

Similarly look at the `Integrator` in `myProject/components/myProject.Integrator/myProject_Integrator_Impl.cxx`:

```
/**
 * Method:  integrate[]
 */
double
myProject::Integrator_impl::integrate_impl (
  /* in */double lowBound,
  /* in */double upBound,
  /* in */int32_t count )
{
  // DO-NOT-DELETE splicer.begin(myProject.Integrator.integrate)
  // Insert-Code-Here {myProject.Integrator.integrate} (integrate method)

    // DO-DELETE-WHEN-IMPLEMENTING exception.begin()
    /*
     * This method has not been implemented
     */
    ::sidl::NotImplementedException ex = ::sidl::NotImplementedException::_create(
    ex.setNote("This method has not been implemented");
    ex.add(__FILE__, __LINE__, "integrate");
    throw ex;
    // DO-DELETE-WHEN-IMPLEMENTING exception.end()

  // DO-NOT-DELETE splicer.end(myProject.Integrator.integrate)
}
```

Again remove this boilerplate code and insert an implementation of the Trapezoid rule for integration that *uses* the `FunctionPort`:

```
/**
 * Method:  integrate[]
 */
double
myProject::Integrator_impl::integrate_impl (
  /* in */double lowBound,
  /* in */double upBound,
  /* in */int32_t count )
{
  // DO-NOT-DELETE splicer.begin(myProject.Integrator.integrate)


   myProject::FunctionPort  myFunPort;
   gov::cca::Port          generalPort;

   generalPort = d_services.getPort("FunctionPort");
   if (generalPort._is_nil()){
       fprintf(stderr, "Error:: %s:%d: generalPort is nil - \
             maybe I'm not connected to any port!!\n",
             __FILE__, __LINE__);
       exit(1);
    }

    myFunPort = ::babel_cast< myProject::FunctionPort >(generalPort);
    if (myFunPort._is_nil()){
       fprintf(stderr, "Error:: %s:%d: myFunPort is nil - \
             maybe I'm connected to the wrong \"Provides\" port!!\n",
             __FILE__, __LINE__);
       exit(1);
    }

    double h = (upBound - lowBound) / count;
    double retval = 0.0;
    double sum = 0.0;
    for (int i = 1; i <= count; i++){
       sum += myFunPort.evaluate(lowBound + (i - 1) * h) +
              myFunPort.evaluate(lowBound + i * h);
    }
    retval = h/2.0 * sum;
    d_services.releasePort("FunctionPort");
    return retval;


  // DO-NOT-DELETE splicer.end(myProject.Integrator.integrate)
}
```

Finally for the `Driver` component in `myProject/components/myProject.Driver/myProject_Driver_Impl.cxx` we have to implement the `GoPort` to get things going. The generated (empty) method looks like:

```
/**
```

```
 *
 * 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)
  // Insert-Code-Here {myProject.Driver.go} (go method)

    // DO-DELETE-WHEN-IMPLEMENTING exception.begin()
    /*
     * This method has not been implemented
     */
    ::sidl::NotImplementedException ex = ::sidl::NotImplementedException::_create(
    ex.setNote("This method has not been implemented");
    ex.add(__FILE__, __LINE__, "go");
    throw ex;
    // DO-DELETE-WHEN-IMPLEMENTING exception.end()

  // DO-NOT-DELETE splicer.end(myProject.Driver.go)
}
```

Again delete the protective boilerplate code between the `splicer` directives and insert an implementation of the `GoPort` method `go`. The `go` function will be called by the framework when the component's "go" button is pushed in the GUI. We will implement that method to get a reference to the `IntegratorPort` that we have been connected to and use it to compute the integral:

```
/**
 *
 * 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)



  double value;
  int count = 100000;
  double lowerBound = 0.0, upperBound = 1.0;

  ::myProject::IntegratorPort integrator;

  // get the port ...
  gov::cca::Port port = d_services.getPort("IntegratorPort");
  integrator = babel_cast< ::myProject::IntegratorPort >(port);
```

```
  if(integrator._is_nil()) {
    fprintf(stdout, "drivers.CXXDriver not connected\n");
    d_services.releasePort("IntegratorPort");
    return -1;
  }
    // operate on the port
    value = integrator.integrate (lowerBound, upperBound, count);

    fprintf(stdout,"Value = %lf\n", value);
    fflush(stdout);

  // release the port.
  d_services.releasePort("IntegratorPort");
  return 0;


  // DO-NOT-DELETE splicer.end(myProject.Driver.go)
}
```

Now remake your project tree to finish the components:

```
$  configure;make
```

```
checking for bash... /bin/sh
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
checking for openpty in -lutil... yes
checking for bocca... /home/rob/cca/bocca/cca-tools-0.6.4_rc8/bin/bocca
configure: Configuring with languages: c cxx f90 python java
configure: Project source dir apparently /var/tmp/user/myProject
configure: Using 1 processe(s) in calls to make.
checking whether make sets $(MAKE)... yes
configure: creating ./config.status
config.status: creating project.make
config.status: creating buildutils/common_vars.make
config.status: creating utils/run-gui.sh
config.status: creating utils/bocca-gui-backend.sh
config.status: executing outmsg commands

# ==========================================================================
# Building in components/. Available languages: c cxx f90 python java
# ==========================================================================
make[1]: Entering directory `/var/tmp/user/myProject/components'
```

```
### Building component myProject.Driver:
- creating component library: libmyProject.Driver.la ...
- finished libtooling: components/lib/libmyProject.Driver.la ...
- creating Ccaffeine test script (components/tests/instantiation.gen.rc)...

### Building component myProject.Function:
- creating component library: libmyProject.Function.la ...
- finished libtooling: components/lib/libmyProject.Function.la ...
- creating Ccaffeine test script (components/tests/instantiation.gen.rc)...

### Building component myProject.Integrator:
- creating component library: libmyProject.Integrator.la ...
- finished libtooling: components/lib/libmyProject.Integrator.la ...
- creating Ccaffeine test script (components/tests/instantiation.gen.rc)...

### Building component myProject.emptyComponent: doing nothing -- library is up-to

Build summary:
SUCCESS building myProject.Driver
SUCCESS building myProject.Function
SUCCESS building myProject.Integrator

### To test instantiation of successfully built components, run 'make check' ###
make[1]: Leaving directory `/var/tmp/user/myProject/components'

################ Finished building everything ################
####### You can run some simple tests with 'make check' #######

$
```

You should now be able to instantiate these components, assemble them into an application, and run the application, following the same procedures as in Chapter 2, *Assembling and Running a CCA Application*, and get a result that's reasonably close to pi.

3.3.2.2. Fortran9X Implementation

Assumes you created the project with **bocca create project myProject --language=f90**

Look at the file: `myProject/components/myProject.Function/myProject_Function_Impl.F90` that bocca has generated for you. Find the `evaluate_Impl` method generated from the SIDL definition of `FunctionPort` method `evaluate`:

```
!
! Method:  evaluate[]
!

recursive subroutine myProject_Function_evaluate_mi(self, x, retval,          &
  exception)
  use sidl
  use sidl_NotImplementedException
  use sidl_BaseInterface
  use sidl_RuntimeException
  use myProject_Function
  use myProject_Function_impl
  ! DO-NOT-DELETE splicer.begin(myProject.Function.evaluate.use)
  ! Insert-Code-Here {myProject.Function.evaluate.use} (use statements)
  ! DO-NOT-DELETE splicer.end(myProject.Function.evaluate.use)
```

```
  implicit none
  type(myProject_Function_t) :: self ! in
  real (kind=sidl_double) :: x ! in
  real (kind=sidl_double) :: retval ! out
  type(sidl_BaseInterface_t) :: exception ! out

! DO-NOT-DELETE splicer.begin(myProject.Function.evaluate)
! Insert-Code-Here {myProject.Function.evaluate} (evaluate method)

! DO-DELETE-WHEN-IMPLEMENTING exception.begin()
!
! This method has not been implemented
!
  type(sidl_BaseInterface_t) :: throwaway
  type(sidl_NotImplementedException_t) :: notImpl
  call new(notImpl, exception)
  call setNote(notImpl, 'Not Implemented', exception)
  call cast(notImpl, exception,throwaway)
  call deleteRef(notImpl,throwaway)
  return
! DO-DELETE-WHEN-IMPLEMENTING exception.end()

! DO-NOT-DELETE splicer.end(myProject.Function.evaluate)
end subroutine myProject_Function_evaluate_mi
```

As the comment suggests this method is "not implemented" but some code has been inserted by Babel to make sure an exception is thrown to inform the user if this is called by mistake. Remove this boilerplate code and substitute an implementation for the `SquareFunction`: x^2:

```
!
! Method:  evaluate[]
!

recursive subroutine SquareFun_evaluatetu_xix96dh_mi(self, x, retval,          &
  exception)
  use sidl
  use sidl_NotImplementedException
  use sidl_BaseInterface
  use sidl_RuntimeException
  use functions_SquareFunction
  use functions_SquareFunction_impl
  ! DO-NOT-DELETE splicer.begin(functions.SquareFunction.evaluate.use)
  ! Insert-Code-Here {functions.SquareFunction.evaluate.use} (use statements)
  ! DO-NOT-DELETE splicer.end(functions.SquareFunction.evaluate.use)
  implicit none
  type(functions_SquareFunction_t) :: self ! in
  real (kind=sidl_double) :: x ! in
  real (kind=sidl_double) :: retval ! out
  type(sidl_BaseInterface_t) :: exception ! out

! DO-NOT-DELETE splicer.begin(functions.SquareFunction.evaluate)

        retval = x*x

! DO-NOT-DELETE splicer.end(functions.SquareFunction.evaluate)
end subroutine SquareFun_evaluatetu_xix96dh_mi
```

Now in the same file find the second method for `FunctionPort`: `init`:

```
recursive subroutine myProject_Function_init_mi(self, params, exception)
  use sidl
  use sidl_NotImplementedException
  use sidl_BaseInterface
  use sidl_RuntimeException
  use myProject_Function
  use sidl_double_array
  use myProject_Function_impl
  ! DO-NOT-DELETE splicer.begin(myProject.Function.init.use)
  ! Insert-Code-Here {myProject.Function.init.use} (use statements)
  ! DO-NOT-DELETE splicer.end(myProject.Function.init.use)
  implicit none
  type(myProject_Function_t) :: self ! in
  type(sidl_double_1d) :: params ! in
  type(sidl_BaseInterface_t) :: exception ! out

! DO-NOT-DELETE splicer.begin(myProject.Function.init)
! Insert-Code-Here {myProject.Function.init} (init method)

! Do nothing.

! DO-NOT-DELETE splicer.end(myProject.Function.init)
end subroutine myProject_Function_init_mi
```

Here we don't have any initialization so we just eliminate the code that throws the exception for running the method.

Similarly look at the `Integrator` in `myProject/components/myProject.Integrator/myProject_Integrator_Impl.F90` specifically the `integrate` method:

```
!
! Method:  integrate[]
!

recursive subroutine Integrat_integrate27jlju5gbk_mi(self, lowBound, upBound,  &
  count, retval, exception)
  use sidl
  use sidl_NotImplementedException
  use sidl_BaseInterface
  use sidl_RuntimeException
  use myProject_Integrator
  use myProject_Integrator_impl
  ! DO-NOT-DELETE splicer.begin(myProject.Integrator.integrate.use)
  ! Insert-Code-Here {myProject.Integrator.integrate.use} (use statements)
  ! DO-NOT-DELETE splicer.end(myProject.Integrator.integrate.use)
  implicit none
  type(myProject_Integrator_t) :: self ! in
  real (kind=sidl_double) :: lowBound ! in
  real (kind=sidl_double) :: upBound ! in
  integer (kind=sidl_int) :: count ! in
  real (kind=sidl_double) :: retval ! out
  type(sidl_BaseInterface_t) :: exception ! out

! DO-NOT-DELETE splicer.begin(myProject.Integrator.integrate)
! Insert-Code-Here {myProject.Integrator.integrate} (integrate method)
```

```
! DO-DELETE-WHEN-IMPLEMENTING exception.begin()
!
! This method has not been implemented
!
  type(sidl_BaseInterface_t) :: throwaway
  type(sidl_NotImplementedException_t) :: notImpl
  call new(notImpl, exception)
  call setNote(notImpl, 'Not Implemented', exception)
  call cast(notImpl, exception,throwaway)
  call deleteRef(notImpl,throwaway)
  return
! DO-DELETE-WHEN-IMPLEMENTING exception.end()

! DO-NOT-DELETE splicer.end(myProject.Integrator.integrate)
end subroutine Integrat_integrate27jlju5gbk_mi
```

Again remove this boilerplate code and insert an implementation that *uses* the `FunctionPort` and a Monte Carlo method for integration (to run this we will need to hook up the RandNumGenerator component from the pre-built *TUTORIAL_SRC* tree):

```
!
! Method:  integrate[]
!

recursive subroutine MonteCar_integrateudcgc9x69z_mi(self, lowBound, upBound,  &
  count, retval, exception)
  use sidl
  use sidl_NotImplementedException
  use sidl_BaseInterface
  use sidl_RuntimeException
  use integrators_MonteCarlo
  use integrators_MonteCarlo_impl
  ! DO-NOT-DELETE splicer.begin(integrators.MonteCarlo.integrate.use)
  ! Insert use statements here...

  use function_FunctionPort
  use randomgen_RandomGeneratorPort
  use gov_cca_Services
  use gov_cca_Port
  use sidl_BaseInterface

  ! DO-NOT-DELETE splicer.end(integrators.MonteCarlo.integrate.use)
  implicit none
  type(integrators_MonteCarlo_t) :: self ! in
  real (kind=sidl_double) :: lowBound ! in
  real (kind=sidl_double) :: upBound ! in
  integer (kind=sidl_int) :: count ! in
  real (kind=sidl_double) :: retval ! out
  type(sidl_BaseInterface_t) :: exception ! out

! DO-NOT-DELETE splicer.begin(integrators.MonteCarlo.integrate)


! Insert the implementation here...


  type(gov_cca_Port_t) :: generalPort, generalPort2
  type(function_FunctionPort_t) :: functionPort
  type(randomgen_RandomGeneratorPort_t) :: randomPort
  type(SIDL_BaseInterface_t) :: excpt, funcExcpt, randomExcpt, throwaway
```

```
! Private data reference
type(integrators_MonteCarlo_wrap) :: dp

real (selected_real_kind(15, 307)) :: sum, width, x, func
integer (selected_int_kind(9)) :: i

! Access private data
call integrators_MonteCarlo__get_data_m(self, dp)
retval = -1

if (not_null(dp%d_private_data%d_services)) then
   ! Obtain a handle to a FunctionPort
   call getPort(dp%d_private_data%d_services, &
        'function', generalPort, funcExcpt)

   if (is_null(funcExcpt)) then

      call cast(generalPort, functionPort, throwaway)
      if (not_null(functionPort)) then

         ! Obtain a handle to a RandomGeneratorPort
         call getPort(dp%d_private_data%d_services, &
              'RandomGeneratorPort', generalPort2, randomExcpt)

         if (is_null(randomExcpt)) then
            call cast(generalPort2, randomPort, throwaway)
            if (not_null(randomPort)) then
               ! Compute integral
               sum = 0
               width = upBound - lowBound
               do i = 1, count
                  call getRandomNumber(randomPort, x, excpt)
                  call checkExceptionMC(excpt, &
                    'getRandomNumber(randomPort, x, excpt)')
                  x = lowBound + width*x
                  call evaluate(functionPort, x, func, excpt)
                  call checkExceptionMC(excpt, &
                  'evaluate(functionPort, x, func, excpt)')
                  sum = sum + func
               enddo
               retval = width*sum/count
            else  ! randomPort is null
               write(*,*) 'Exception: MonteCarlo: incompatible RandomGeneratorPo
            endif
            call deleteRef(generalPort2, throwaway)

         else  ! randomExcpt is not null

            call checkExceptionMC(randomExcpt, 'getPort(''RandomPort'')')
         endif
      else    ! functionPort is null
         write(*,*) 'Exception: MonteCarlo: incompatible FunctionPort'
      endif
      call deleteRef(generalPort, throwaway)

      ! Free ports
      call releasePort(dp%d_private_data%d_services, &
           'RandomGeneratorPort', excpt)
      call checkExceptionMC(excpt, 'releasePort(''RandomGeneratorPort'')')

      call releasePort(dp%d_private_data%d_services, &
           'function', excpt)
      call checkExceptionMC(excpt, 'releasePort(''function'')')

   else      ! funcExcpt is not null
```

```
        call checkExceptionMC(funcExcpt, 'getPort(''function'')')
     endif
  else
     write(*,*) 'Error: MonteCarlo: integrate called before setServices'
  endif

! DO-NOT-DELETE splicer.end(integrators.MonteCarlo.integrate)
end subroutine MonteCar_integrateudcgc9x69z_mi
```

Finally for the `Driver` component in file `myProject/components/myProject.Driver/myProject_Driver_Impl.F90` we have to implement the `GoPort` to get things going. The generated (empty) method looks like:

```
!
! Method:  go[]
!
! 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.
!

recursive subroutine myProject_Driver_go_mi(self, retval, exception)
  use sidl
  use sidl_NotImplementedException
  use sidl_BaseInterface
  use sidl_RuntimeException
  use myProject_Driver
  use myProject_Driver_impl
  ! DO-NOT-DELETE splicer.begin(myProject.Driver.go.use)
  ! Insert-Code-Here {myProject.Driver.go.use} (use statements)
  ! DO-NOT-DELETE splicer.end(myProject.Driver.go.use)
  implicit none
  type(myProject_Driver_t) :: self ! in
  integer (kind=sidl_int) :: retval ! out
  type(sidl_BaseInterface_t) :: exception ! out

! DO-NOT-DELETE splicer.begin(myProject.Driver.go)
! Insert-Code-Here {myProject.Driver.go} (go method)

! DO-DELETE-WHEN-IMPLEMENTING exception.begin()
!
! This method has not been implemented
!
  type(sidl_BaseInterface_t) :: throwaway
  type(sidl_NotImplementedException_t) :: notImpl
  call new(notImpl, exception)
  call setNote(notImpl, 'Not Implemented', exception)
  call cast(notImpl, exception,throwaway)
  call deleteRef(notImpl,throwaway)
  return
! DO-DELETE-WHEN-IMPLEMENTING exception.end()

! DO-NOT-DELETE splicer.end(myProject.Driver.go)
end subroutine myProject_Driver_go_mi
```

Again delete the protective boilerplate code between the `splicer` directives and insert an implementation of the GoPort method go. The go subroutine will be called by the framework when the component's "go" button is pushed in the GUI. We will implement that method to get a reference to the IntegratorPort that we have been connected to and use it to compute the integral:

```
!
! Method:  go[]
!
! 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.
!

recursive subroutine drivers_F90Driver_go_mi(self, retval, exception)
  use sidl
  use sidl_NotImplementedException
  use sidl_BaseInterface
  use sidl_RuntimeException
  use drivers_F90Driver
  use drivers_F90Driver_impl
  ! DO-NOT-DELETE splicer.begin(drivers.F90Driver.go.use)
  ! Insert use statements here...

  use sidl_BaseInterface
  use gov_cca_Port
  use integrator_IntegratorPort

  ! DO-NOT-DELETE splicer.end(drivers.F90Driver.go.use)
  implicit none
  type(drivers_F90Driver_t) :: self ! in
  integer (kind=sidl_int) :: retval ! out
  type(sidl_BaseInterface_t) :: exception ! out

! DO-NOT-DELETE splicer.begin(drivers.F90Driver.go)


! Insert the implementation here...


  type(gov_cca_Port_t) :: generalPort
  type(SIDL_BaseInterface_t) :: excpt
  type(integrator_IntegratorPort_t) :: integratorPort

  ! Private data reference
  type(drivers_F90Driver_wrap) :: dp

  ! local variables for integration
  real (kind=sidl_double) :: lowBound
  real (kind=sidl_double) :: upBound
  integer (kind=sidl_int) :: count
  real (kind=sidl_double) :: value

  ! Initialize local variables
  count = 100000
  lowBound = 0.0
  upBound = 1.0

  ! Access private data
  call drivers_F90Driver__get_data_m(self, dp)
  retval = -1

  ! get the port ...
  call getPort(dp%d_private_data%d_services, &
```

```
          'integrate', generalPort, excpt)
     call checkExceptionDriver(excpt, 'getPort(''integrate'')')
     if(is_null(generalPort)) then
        write(*,*) 'drivers.F90Driver not connected'
        return
     endif

     ! Get an IntegratorPort reference from the general port one
     call cast(generalPort, integratorPort, excpt)
     call checkExceptionDriver(excpt, 'cast(generalPort, integratorPort, excpt)')

     if (not_null(integratorPort)) then
        value = -1.0 ! nonsense number to confirm it is set

        ! operate on the port
        call integrate(integratorPort, lowBound, upBound, count, value, excpt)
        call checkExceptionDriver(excpt, &
           'integrate(integratorPort, lowBound, upBound, count, value, excpt)')

        write(*,*) 'Value = ', value
     else   ! integratorPort is null
        write(*,*) 'DriverF90: incompatible IntegratorPort'
     endif

     ! release the port
     call releasePort(dp%d_private_data%d_services, &
          'integrate', excpt)
     call checkExceptionDriver(excpt, 'releasePort(''integrate'')')

     retval = 0
     return

! DO-NOT-DELETE splicer.end(drivers.F90Driver.go)
end subroutine drivers_F90Driver_go_mi
```

Now remake your project tree to finish the components:

```
$  configure;make
```

You should now be able to instantiate these components, assemble them into an application, and run the application, following the same procedures as in Chapter 2, *Assembling and Running a CCA Application*, and get a result that's reasonably close to 1/3.

3.3.2.3. C Implementation

Assumes you created the project with **bocca create project myProject --language=c**

Look at the file: `myProject/components/myProject.Function/myProject_Function_Impl.c` that bocca has generated for you. Find the `evaluate_Impl` method generated from the SIDL definition of `FunctionPort`:

```
/*
 * Method:  evaluate[]
```

```
 */

#undef __FUNC__
#define __FUNC__ "impl_myProject_Function_evaluate"

#ifdef __cplusplus
extern "C"
#endif
double
impl_myProject_Function_evaluate(
  /* in */ myProject_Function self,
  /* in */ double x,
  /* out */ sidl_BaseInterface *_ex)
{
  *_ex = 0;
  {
    /* DO-NOT-DELETE splicer.begin(myProject.Function.evaluate) */
    /* Insert-Code-Here {myProject.Function.evaluate} (evaluate method) */

    /* DO-DELETE-WHEN-IMPLEMENTING exception.begin() */
    /*
     * This method has not been implemented.
     */
    SIDL_THROW(*_ex, sidl_NotImplementedException,     "This method has not been i
  EXIT:;
    /* DO-DELETE-WHEN-IMPLEMENTING exception.end() */

    /* DO-NOT-DELETE splicer.end(myProject.Function.evaluate) */
  }
}
```

As the comment suggests this method is "not implemented" but some code has been inserted by Babel to make sure an exception is thrown to inform the user if this is called by mistake. Remove this boilerplate so and substitute an implementation for the `LinearFunction`:

```
/*
 * Method:  evaluate[]
 */

#undef __FUNC__
#define __FUNC__ "impl_functions_LinearFunction_evaluate"

#ifdef __cplusplus
extern "C"
#endif
double
impl_functions_LinearFunction_evaluate(
  /* in */ functions_LinearFunction self,
  /* in */ double x,
  /* out */ sidl_BaseInterface *_ex)
{
  *_ex = 0;
  {
    /* DO-NOT-DELETE splicer.begin(functions.LinearFunction.evaluate) */


  /* Insert the implementation of the evaluate method here... */

  return 12.0 * x + 3.2;

    /* DO-NOT-DELETE splicer.end(functions.LinearFunction.evaluate) */
  }
```

Now in the same file find the second method for `FunctionPort`: `init`:

```
/*
 * Method:  init[]
 */

#undef __FUNC__
#define __FUNC__ "impl_functions_LinearFunction_init"

#ifdef __cplusplus
extern "C"
#endif
void
impl_functions_LinearFunction_init(
  /* in */ functions_LinearFunction self,
  /* in array<double> */ struct sidl_double__array* params,
  /* out */ sidl_BaseInterface *_ex)
{
  *_ex = 0;
  {
    /* DO-NOT-DELETE splicer.begin(functions.LinearFunction.init) */

  /* Insert the implementation of the init method here... */

/* Do Nothing. */

    /* DO-NOT-DELETE splicer.end(functions.LinearFunction.init) */
  }
}
```

Here we don't have any initialization so we just eliminate the code that throws the exception for running the method.

To incorporate this component into an application you will have to link it up with `Driver` and `Integrator` components from the pre-built source tree. To compile and link this component remake your project:

```
$  configure;make
```

Connect up and run your components in the GUI if you like:

```
$  ./utils/run-gui.sh
```

Chapter 4. Understanding arrays and component state

$Revision: 1.11 $
$Date: 2007/09/24 18:10:42 $

In this exercise, you will develop a component that uses Babel arrays as arguments in the ports that the component provides. Specifically, this exercise will introduce and use the following concepts and artifacts

- Creating, changing and accessing “normal” SIDL arrays.
- Using “raw” SIDL arrays.
- Using object (component) state to store arbitrary data types (including arrays).

Note

This exercise is self-contained. Components and ports explained and developed here do not rely on components and/or ports used in the numerical integration exercises.

4.1. Introduction

In the first part of this exercise, we present the details of two components that work together to evaluate a series of simple linear matrix operations. One component serves as the driver, while the other *provides* the `LinearOp` port. The specification of this port is found in the file `$TUTORIAL_SRC/ports/sidl/arrayop.LinearOp.sidl`, partially reproduced here for easy reference

```
   ...

 /** This port can be used to evaluate a matrix operation of the form
   * of the form
   *  R = Sum[i=1, N] {Alpha_i A_i v_i} + Sum[j=1, N] {Beta_j v_j}}
   * Where:
   *    alpha_i, Beta_j   Double scalar
   *    A_i               Double array of size [m, n]
   *    v_i, v_j          Vector of size [n]
   *    A_i  v_j          Matrix vector multiplication
   */
     interface LinearOp extends gov.cca.Port
     {
 /** Initialize (or Re-Initialize) internal state in preparation
   * for accumulation.
   */
        void  init();

 /** Evaluate Acc = Acc + alpha A x,  where
   *   Acc     The internal accumulator maintained by implementors
   *           of this interface
   * return the result  in vector y (of size m)
   */
        int mulMatVec (in double               alpha,
                       in rarray<double, 2>    A(m, n),
```

```
                        in rarray<double, 1>    x(n),
                        inout rarray<double, 1> y(m),
                        in int                  m,
                        in int                  n);

/** Evaluate Acc = Acc + beta v,  where
  *   Acc     The internal accumulator maintained by implementors
  *           of this interface
  * return the result  in vector y (of size m)
  */
        int addVec ( in double            beta,
                     in array<double, 1>  v,
                     out array<double, 1> r);

/** Get result of linear operators
  *
        int getResult (inout rarray<double, 1>  r(m),
                        in    int                  m);

     }

  ...
```

Note

- The port methods `mulMatVec` and `getResult` use SIDL raw arrays (also referred to as r-arrays), which are designed to simplify implementation in Fortran dialects (especially Fortran77). Raw arrays are assumed to adhere to column-major memory layout, with zero-based indexing. Further details of raw SIDL arrays can be found in the Babel User Guide [http://www.llnl.gov/CASC/components/software.html].

- The port method `addVec` uses the “normal” SIDL array class. This class allows access to arrays through accessor functions. There are also provisions that allow access to the underlying array memory for more efficient operations. Refer to the Babel User Guide [http://www.llnl.gov/CASC/components/software.html] for more details on normal SIDL arrays.

The tutorial source contains fully implemented three components that *provide* the `LinearOp` port. The components `F90ArrayOp`, `F77ArrayOp`, and `CArrayOp` can be found at `$TUTORIAL_SRC/components/{arrayOps.F90ArrayOp, arrayOps.F77ArrayOp, arrayOps.CArrayOp`. In addition, a driver component that *uses* the `LinearOp` port can be found at `$TUTORIAL_SRC/components/arrayDrivers.CDriver`.

In the following sections, we will present some of the aspects of using SIDL arrays, using the code in the driver and the three `arrayOps` components as examples. You will then be asked to implement a component that *provides* a `NonLinearOp` port and a driver, using the aforementioned four components as a template.

4.2. The `CDriver` Component

The SIDL specification for the `CDriver` component can be found in the file `$TUTORIAL_SRC/components/sidl/arrayDrivers.CDriver.sidl`. The implementation of this component (in the C programming language) can be found at `$TUTORIAL_SRC/components/arrayDrivers.CDriver/` in the two files `arrayDrivers_CDriver_Impl.c` and arrayDrivers_CDriver_Impl.h. Component implementation details include details of component/framework interaction that should be now familiar, and will not be discussed further in this exercise. We will focus on the handling of different types of SIDL arrays in the go method.

4.2.1. Using SIDL Raw Arrays

Raw arrays (and vectors) are used as arguments in the call to mulMatVec. Note that multidimensional SIDL raw arrays are *always* assumed to use column-major storage. This requirement necessitates special treatment when calling methods that use SIDL raw arrays as arguments from languages that follow a default raw-major array storage order (C and C++). The caller may choose to alter the memory layout of the array argument throughout its entire lifetime, or alternatively perform a matrix transpose operation on “native” arrays before and after every call to a SIDL method that uses raw arrays. In the example presented here, we have chosen to adopt column-major storage throughout the lifetime of the raw array argument A, as shown in the initialization code shown below

```
/*
 *      _          _          _   _          _   _
 *     | 1.0  4.0 |          | 1.0 |        | 3.0 |
 * A = | 2.0  5.0 |     v1 = | 2.0 |  sda1 = | 4.0 |
 *     | 3.0  6.0 |           -   -         | 5.0 |
 *      -          -                         -   -
 *
 * Note that A needs to be stored in column-major order to make
 * the call using SIDL raw arrays
 */
value = 0.0;
for (i = 0; i <= m; i++){
  for (j = 0; j <= n; j++){
    A[i*n+j] = (value += 1.0);
  }
}
```

When making a call to a SIDL method that has SIDL raw arrays arguments, the dimensions of those arrays must be explicitly included in the argument list in the SIDL specification. No special “wrapping” of native arrays is needed to make a call using SIDL raw arrays arguments. This can be seen in the call to the mulMatVec method.

```
retval = arrayop_LinearOp_mulMatVec(linopPort, alpha, A, v1, y, m , n, &throwaw
if (retval != 0){
  fprintf(stderr, "Error:: %s:%d: Error in call to mulMatVec() \n",
          __FILE__, __LINE__);
   return(-1);
}
```

The requirement to use column-major memory layout is one of the restrictions imposed by Babel to allow for the use of raw arrays. See the Babel User Guide [http://www.llnl.gov/CASC/components/software.html] for the complete list.

4.2.2. Using SIDL Normal Arrays

SIDL “normal” arrays are implemented in the Babel runtime, with bindings in all Babel supported languages. SIDL normal arrays provided a more flexible array representation, with the ability to directly access the underlying array memory in languages that support this capability (C, C++, F90, and F77). In Python, there are situations where arrays must be copied when passing in and out, but direct access is used wherever the Numerical Python package will allow. In Java, arrays are accessed using the Java Native Interface. More information on SIDL normal arrays can be found in the Babel User Guide [http://www.llnl.gov/CASC/components/software.html].

In this exercise, the method addVec uses SIDL normal arrays (sda1, and sda2). The SIDL specification of the addVec method designates sda1 as an input argument, therefore it needs to be created (more specifically, associated with memory) on the caller side before the call is made. The Babel runtime provides array manipulation bindings in Babel supported languages (except Python, which uses *Numeric Python* arrays). The one-dimensional, SIDL double array sda1 is created using the following code

```
sda1 = sidl_double__array_create1d(m);
if (!sda1){
   fprintf(stderr, "Error:: %s:%d: Error creating sda1.\n",
           __FILE__, __LINE__);
   return(-1);
}
```

The Babel runtime C binding contains macros that allow direct access to underlying SIDL array memory and properties (dimensions, strides, etc.), without having to go through the standard set() and get() methods. One such macro is used in this example to access the underlying memory of SIDL array sda1

```
sda1_data = sidlArrayAddr1(sda1, 0);
for (value =0.0, i = 0; i <= m; i++){
   sda1_data[i] = (double) i + 3.0 ;
}
```

Other macros are used in the loop that prints the result returned in the SIDL out array sda2, after the call to addVec.

```
printf("Result2 = ");
for ( i = sidlLower(sda2, 0); i <= sidlUpper(sda2, 0); i++){
   printf("%.2f  ", sidlArrayElem1(sda2,i));
}
printf("\n");
```

Direct access to underlying SIDL array memory is also available in the Babel SIDL array binding in F77, F90, and C++. Example of such use is available in the discussion in Section 4.3, “Linear Array Operations Components”.

4.3. Linear Array Operations Components

In this section, we present some of the implementation details of (non-driver) components that *provide* ports with SIDL arrays as arguments. The tutorial source contains implementation of three components, CArrayOp, F77ArrayOp, and F90ArrayOp, implemented in C, F77, and F90 respectively.

4.3.1. The CArrayOp Component

Code for the CArrayOp component can be found in the directory $TUTORIAL_SRC/components/arrayOps.CArrayOp/, in the two Impl files arrayOps_CArrayOp_Impl.c and arrayOps_CArrayOp_Impl.h. Private component state is represented by entries in the struct arrayOps_CArrayOp__data in the header file arrayOps_CArrayOp_Impl.h

```
struct arrayOps_CArrayOp__data {
   ...
   ...
   double            *myVector;
   int               myVecLen;
   /* DO-NOT-DELETE splicer.end(arrayOps.CArrayOp._data) */
```

```
};
```

Private component data is initialized and associated with the component instance in the bocca-generated component constructor method `impl_arrayOps_CArrayOp__ctor`

```
  struct arrayOps_CArrayOp__data *dptr =
       (struct arrayOps_CArrayOp__data*)malloc(sizeof(struct arrayOps_CArrayOp__da
   if (dptr) {
      memset(dptr, 0, sizeof(struct arrayOps_CArrayOp__data));
   }
   arrayOps_CArrayOp__set_data(self, dptr);
```

Note the use of the *built-in* method `arrayOps_CArrayOp__set_data` to associate the newly allocated `struct` with this component instance. A corresponding method, `arrayOps_CArrayOp__get_data` is used to access this private data.

The method `impl_arrayOps_CArrayOp_mulMatVec` uses SIDL raw arrays (array `A`, and vectors `x` and `y`). Multi-dimension SIDL raw arrays are assumed to be stored in column-major order, as shown in the code to multiply array `A` and vector `x`

```
   for (i= 0; i <= m; i++){
     y[i] = 0.0;
     for (j = 0 ; j <= n; j++){
       y[i] += alpha * A[j*m + i] * x[j];  /* Raw array A is column-major */
     }
     pd->myVector[i] += y[i];
     y[i] = pd->myVector[i];
   }
```

The method `impl_arrayOps_CArrayOp_addVec` uses the more flexible SIDL normal arrays. SIDL normal arrays are represented in C using a `struct sidl_`*`XXX`*`__array`, where *`XXX`* is the actual type of array elements. In this example, the SIDL `out` normal array `*r` is created (and underlying memory allocated) in the call

```
   *r = sidl_double__array_create1d(n);
```

Direct access to a SIDL normal array's underlying memory is acheived via the C macro `sidlArrayAddr1` (for 1-dimensional arrays `*r` and `v`).

Note

When implementing a method that has SIDL normal arrays as arguments, it should not be assumed that the array is contiguous in memory (stride=1). SIDL normal arrays allow for different strides in all dimensions. As such, the correct code for vector addition has the form

```
   vstride = sidlStride(v, 0);
   for ( i = 0; i <= n; i++){
      rdata[i] = pd->myVector[i] += beta * vdata[i*vstride];
   }
```

No stride is used when accessing the vector `r` since it is created inside the `addVec` routine with a stride=1 (implied in the call to `sidl_double__array_create1d`).

4.3.2. The F77ArrayOp Component

Code for the F77ArrayOp component can be found in the directory $TUTORIAL_SRC/components/arrayOps.F77ArrayOp/, in Impl file arrayOps_f77ArrayOp_Impl.f. Private component state is represented by entries an an array of SIDL opaque types. It is the responsibility of the programmer to ensure consistency of the treatment of entries in this array across method calls (this is similar to the way entries into common blocks are manipulated). Code for the creation and initialization of the private component state can be found in the component constructor method arrayOps_F77ArrayOp__ctor_fi.

```
        tmp = 0
        itmp = 0

        call sidl_int__array_create1d_f(1, intArray)
        if (intArray .ne. 0) then
           call sidl_opaque__array_set1_f(stateArray, 0, tmp)
           call sidl_int__array_set1_f(intArray, 0, itmp)
           call sidl_opaque__array_set1_f(stateArray, 1, intArray)
           call sidl_opaque__array_set1_f(stateArray, 2, tmp)
        else
     . . .
```

The SIDL *built-in* method arrayOps_F77ArrayOp__set_data_f is used to associate the newly created SIDL opaque array with this instance of the component. The method arrayOps_F77ArrayOp__get_data_f is used to retrieve this private data for further manipulation.

The method arrayOps_F77ArrayOp_mulMatVec_fi uses SIDL raw arrays arguments. In F77 implementation, SIDL raw arrays appear as regular F77 arrays, with zero-based indexing. The component uses the SIDL normal array accVector to store the running sum of the linear matrix operations. Note that this enables the dynamic sizing of this vector at runtime to match the dimensions of the array and vector arguments. Direct access to the underlying memory for SIDL normal arrays is done through the sidl_double__array_access_f method (for arrays of SIDL type double). This method computes uses a *reference array* (nativeVec) of size one, and computes the offset (refindex) that needs to be added to indices into nativeVec to access memory associated with SIDL normal array accVector.

```
       call sidl_double__array_access_f(accVector, nativeVec,
      $       lower, upper, stride, refindex)
       do i = 0, m-1
          y(i) = nativeVec(refindex + i)
          do j = 0, n-1
             y(i) = y(i) + alpha * A(i, j) * x(j)
          end do
          y(i) = y(i) + nativeVec(refindex + i)
          nativeVec(refindex + i) = y(i)
       end do
```

Accesssing entries in a normal SIDL array can also be done through *accessor* subroutine calls. In the case of arrays of SIDL type double, the accessor subroutines are sidl_opaque__array_set1_f and sidl_opaque__array_get1_f (for single dimensional arrays).

```
   if (accVector .eq. 0) then
      call sidl_double__array_create1d_f(m, accVector)
      call sidl_int__array_set1_f(intArray, 0, m)
      call sidl_opaque__array_set1_f(stateArray, 2, accVector)
      dblTmp = 0.0
      do i = 0, m-1
```

```
        call sidl_double__array_set1_f(accVector, i, dblTmp)
     end do
  else
     . . .
```

Note

When implementing a method that has SIDL normal arrays as arguments, it should not be assumed that the array is contiguous in memory (stride=1). SIDL normal arrays allow for different strides in all dimensions. As such, the correct code for vector addition in `addVec` has the form

```
      do i = 0, m-1
        nativeR(refindexR + i) = nativeVec(refindex + i) +
    $                            beta * nativeV(refindexV +i*strideV(1))
        nativeVec(refindex + i) = nativeR(refindexR + i)
      end do
```

No stride is used when accessing the array `r` since it is created inside the `addVec` routine with a stride=1 (implied in the call to `sidl_double__array_create1d_f`).

4.3.3. The `F90ArrayOp` Component

Code for the `F90ArrayOp` component can be found in the directory `$TUTORIAL_SRC/components/arrayOps.F90ArrayOp`, in the `Impl` files `arrayOps_F90ArrayOp_Impl.F90`and `arrayOps_F90ArrayOp_Mod.F90`. Private component state is represented by the `type arrayOps_F90ArrayOp_priv` in the file `arrayOps_F90ArrayOp_Mod.F90`

```
  type arrayOps_F90ArrayOp_priv
    sequence
! DO-NOT-DELETE splicer.begin(arrayOps.F90ArrayOp.private_data)

! Bocca generated code. bocca.protected.begin(arrayOps.F90ArrayOp.private_data)
  ! Handle to framework Services object
  type(gov_cca_Services_t) :: d_services
! Bocca generated code. bocca.protected.end(arrayOps.F90ArrayOp.private_data)

  real (selected_real_kind(15, 307)), dimension(:), pointer :: myVectorP
  integer (selected_int_kind(9)) :: myVecLen

! DO-NOT-DELETE splicer.end(arrayOps.F90ArrayOp.private_data)
  end type arrayOps_F90ArrayOp_priv
```

The constructor subroutine `arrayOps_F90ArrayOp__ctor_mi` contains the bocca-generated code for the allocation and initialization of the private data associated with this component instance

```
  type(arrayOps_F90ArrayOp_wrap) :: dp
  ! Allocate memory and initialize
  allocate(dp%d_private_data)
  call set_null(dp%d_private_data%d_services)
  dp%d_private_data%myVectorP => NULL()
  call arrayOps_F90ArrayOp__set_data_m(self, dp)
```

The call to the *built-in* method `arrayOps_F90ArrayOp__set_data_m` associates the newly created structure pointed to via `dp` with this instance of the component. The corresponding method `ar-`

`rayOps_F90ArrayOp__get_data_m` is used to retrieve this private data for further processing.

The subroutine that implements the `mulMatVec` method uses SIDL raw arrays (note that the name of this subroutine is altered by Babel to accomodate F90 identifier length restrictions). SIDL raw arrays manifest themselves in F90 implementations as regular F90 arrays that use zero-based indexing.

```
real (selected_real_kind(15, 307)), dimension(0:m-1, 0:n-1) :: A ! in
real (selected_real_kind(15, 307)), dimension(0:n-1) :: x ! in
real (selected_real_kind(15, 307)), dimension(0:m-1) :: y ! inout
```

The subroutine that implements the `addVec` method uses SIDl normal arrays. SIDL normal arrays are represented as user defined types, with a `pointer` data member (`d_data`that points to an F90 array built on top of the underlying SIDL array memory. While access to SIDL normal array entries can be achieved via accessor subroutines (`set` and `get` - defined for all native SIDL types and user defined classes and interfaces), it is more convenient (and efficient) to access those entries directly via the `d_data` pointer.

```
vdata => v%d_data
rdata => r%d_data
rdata = pd%myVectorP + beta * vdata
pd%myVectorP = rdata
```

Note

When implementing a method that has SIDL normal arrays as arguments, it should not be assumed that the array is contiguous in memory (stride=1). SIDL normal arrays allow for different strides in all dimensions. The Babel runtime build the correct F90 array descriptor (dope vector) that correctly reflects the strides used to create the SIDL array.

4.4. Assignment: `NonLinearOp` Component and Driver

In this section, you will use the `LinearOp` components and driver described earlier as a template to develop a driver and a component that *provides* the `NonLinearOp` port. The specification of this port can be found in the SIDL file `$TUTORIAL_SRC/ports/sidl/arrayop.NonLinearOp.sidl`, and is repeated here for convenience.

```
/** This port can be used to evaluate a linear matrix operation
  * of the form
  *  R = Sum[i=1, N] {Alpha_i log(A_i)} + Sum[j=1, N] {Beta_j A_j .* M_j}}
  * Where:
  *    alpha_i, Beta_j   Double scalar
  *    A_i, M_j          Double array of size [m, n]
  *    log(A_i)          Elementwise log (base 10) of matrix A_i
  *    A_j .* M_j        Elementwise multiplication of A_j and M_j
  */
  interface NonLinearOp extends gov.cca.Port
  {
/** Initialize (or Re-Initialize) internal state in preparation
  * for accumulation.
  */
     void  init();
```

```
/** Evaluate Acc = Acc + alpha log(A) where
  *   log(A)  Elementwise log (base 10) of array A
  *   Acc     The internal accumulator maintained by implementors
  *           of this interafce
  * return the result  in array R
  */
     int logMat (in double                alpha,
                 in rarray<double, 2>     A(m, n),
                 inout rarray<double, 2>  R(m, n),
                 in int                   m,
                 in int                   n);

/** Evaluate Acc = Acc + beta A .* M, where
  *   .*   denotes elementwise multiplications of arrays
  *   Acc  the internal accumulator maintained by implementors
  *        of this interafce
  * return the result  in array R
  */
     int mulMatMat ( in double           beta,
                     in array<double, 2>  A,
                     in array<double, 2>  M,
                     out array<, 2> R);

/** Get result of nonlinear operation accumulation.
  *
     int getResult (inout rarray<double, 2>  R(m, n),
                    in    int                m,
                    in    int                n);
  }
```

Note that you can continue to work in the project directory created earlier, or you can create a new project just for this exercise, since it does not rely on any of the components developed earlier in the tutorial.

1. **Create `NonLinearOp` port**

   Use bocca to create your own version of the `NonLinearOp` port specification by importing the existing definition from `$TUTORIAL_SRC`. This can be done using the command

   ```
   bocca create port arrayop.NonLinearOp \
   --import-sidl=arrayop.NonLinearOp:$TUTORIAL_SRC/ports/sidl/arrayop.NonLinearOp.
   ```

2. **Create `arrayOps.NonLinearOp` component**

   Next you will create a component that provides the `NonLinearOp` port using the bocca command

   ```
   bocca create component arrayOps.NonLinearOp \
   --provides=arrayop.NonLinearOp:NonLinearPort --lang=LANG
   ```

   where LANG is your development language of choice from the list of languages supported by Babel.

3. **Create `arrayDrivers.NLinearDriver` component**

   In this step, you will use bocca to create a driver for the `arrayDrivers.NLinearDriver` component, using the command

```
bocca create component arrayDrivers.NLinearDriver \
--provides=gov.cca.ports.GoPort:Go \
--uses=arrayop.NonLinearOp:NonLinearPort --lang=LANG
```

where LANG is your development language of choice for the driver.

4. **Edit components implementation file(s)**

   Edit the newly generated `Impl` files to implement the methods in the newly created driver component (in the directory `components/arrayDrivers.NLinearDriver`) and the nonlinear matrix operation component (in the directory `components/arrayOps.NonLinearOp`). Build the new components (by running **`make`** in the top level directory of your project (this will also build the required port code for the languages you use).

5. **Running the New `NonLinearOp` Component Application**

   You can run the application using the technique you used in Chapter 2, *Assembling and Running a CCA Application*.

Appendix A. Remote Access for the CCA Environment

$Revision: 1.5 $
$Date: 2004/10/10 21:10:08 $

There is really nothing special about using the CCA environment on a remote system compared to any other tools routinely used in technical computing. But there are a few things you can do that might make it more convenient to work remotely. So here are some notes intended to point you to the appropriate places in the manuals for the software you're using.

A.1. Commandline Access

Everything associated with the CCA *can* be done using only commandline access to the remote system. The primary tool for this kind of access at present is the Secure Shell protocol, SSH. Both free and commercial implementations of ssh are widely available. Among the most common are OpenSSH [http://www.openssh.org] for Linux(-like) systems and PuTTY [http://www.chiark.greenend.org.uk/~sgtatham/putty/] for Windows. When we describe specifically how to do something with an SSH client, we will describe it for these two packages. However we won't be using any unusual capabilities of SSH, so most other implementions probably have an equivalent.

A.2. Graphical Access using X11

Your remote CCA environment will be on a Linux(-like) system (because at present, the CCA tools do not run directly on Windows), in which graphical tools (such as text editors, debuggers, performance tools, etc.) typically use the X11 environment. If you wish to use these graphical tools remotely, you'll need an X11 environment on your local system. This is standard on most Linux(-like) systems. On Windows, you will probably have to install an X11 server.

Warning

Running X11 tools remotely can be annoyingly slow, especially over a long-haul connection or a slow network. You may prefer to stick to commandline tools.

Most SSH clients are capable of *forwarding* X11 traffic through your SSH session. If this option is available to you, it is probably the most convenient and definitely the most secure way of running X11 tools remotely. (It is possible for the administrator of the remote system to configure the SSH server to prevent X11 forwarding, but we try to insure that this is not the case on the systems we use for organized tutorials.)

A.2.1. OpenSSH

In most cases, SSH will forward X11 traffic by default, so the simplest thing is to go ahead and try it. To explicitly enable X11 forwarding use the `-X` option to ssh. If you want to disable it for some reason (for instance, it is too slow for your taste and you have a tendency to inadvertently start up graphical tools instead of commandline ones), then use the `-x` option.

A.2.2. PuTTY

In PuTTY, there is a checkbox to Enable X11 forwarding on the Connection → SSH → Tunnels configuration page.

A.3. Tunneling other Connections through SSH

Similar to X11 forwarding, most SSH clients have the ability to *tunnel* other network connections through an SSH session, also known as *port forwarding*. Tunnels connect a port on your local system to a port on a remote system, so that you can make a connection to the port on your local system and, via the tunnel, it will be forwarded to the designated port of the remote system. (Other tunneling setups are possible, but we do not use them in this Guide.) The remote system could be the system you SSH into, or a system *reachable* from the system you SSH into. The two primary uses for tunnels in the context of the CCA are working on clusters where internal nodes don't have direct access to the external network, and making connections through firewalls, for example to run the GUI (of course the firewall must pass the SSH connection that carries the tunnel).

An important thing to note about tunneling is that the port numbers on both ends of the tunnel must be made explicit. Only one application at a time can listen on a port, so port numbers on both ends of the tunnel must be selected to avoid conflicts. Assuming you're the only user on your local system, you must select non-privileged port numbers (1025-65565) that don't conflict with each other, or with any servers or other applications that might already be using ports on your system. In the examples below, we use port 2022 on the `localhost` side of a tunnel for an SSH connection. The same rules apply to the ports on the remote system. If you're sharing the system on which you're running the exercises, you'll need to be sure to select ports not being used by other users. Though statistically, the chances of a collision are relatively small, we avoid such problems in organized tutorials by *assigning* each user a port number to use for the Ccaffeine GUI (in the examples below, we use port 3314). If you're working on your own and are encountering problems finding a free port, the **netstat** (**netstat -a -t -u** on Linux-like systems, or **netstat -a** at the Windows command prompt) can give you a list of the ports currently in use.

A.3.1. Tunneling with OpenSSH

The `-L` *`localPort`*`:`*`remoteHost`*`:`*`remotePort`* option to **ssh** is used to setup tunnels. The following are examples of some tunneling arrangements that might be useful in a CCA context:

- Establishing an SSH connection to the head node of a cluster which will forward SSH connections to an internal node. Then using the tunnel to make a direct connection to the internal node:

```
ssh -L 2022:clusterInternalNode:22 clusterHeadNode
ssh -p 2022 localhost
```

- Establishing an SSH connection to a firewalled machine which will forward connections from the Ccaffeine GUI running locally to the Ccaffeine framework backend running remotely:

```
ssh -L 3314:remoteHost:3314 remoteHost
java -classpath ccafe-gui.jar \
     gov.sandia.ccaffeine.dc.user_iface.BuilderClient \
        --builderPort 3314 --host localhost
```

> **Tip**
>
> Don't worry if you don't understand the details of the java command that invokes the GUI. It is described in more detail in Section 2.1, “Using the GUI Front-End to Ccaffeine”. The key features for this discussion are the `--builderPort 3314 --host localhost` arguments, which tell the GUI to connect to the *local* end of the tunnel.

- Establishing tunnels to an internal node of a cluster for both SSH and Ccaffeine GUI connections:

```
ssh -L 2022:clusterInternalNode:22 \
    -L 3314:clusterInternalNode:3314 clusterHeadNode
```

which can be used precisely as in the preceeding examples.

A.3.2. Tunneling with PuTTY

In PuTTY, tunnels are specified on the Connection → SSH → Tunnels configuration page. To configure a tunnel, you need to go to the Add new forwarded port section of the page. Source port is the port on your local system that you will connect to in order to use the tunnel. In the OpenSSH instructions above, it is labeled *`localPort`* and is the *first* part of the argument of the `-L` option. In PuTTY, the Destination field is *`remotHost`*`:`*`remotePort`*, or the second and third pieces of the OpenSSH `-L` argument. The Local button should always be checked (meaning that the tunnel will be setup to forward from your *local* system to the destination system).

Tip

You might want to take advantage of PuTTY's ability to save “sessions” to save and easily reuse complicated (or tedious) SSH configurations, particularly those including multiple tunnels.

In order to *use* a tunnel once it is setup, you simply enter give the application **`localhost`** and the appropriate port number to connect to. To initiate a tunneled SSH session with PuTTY, you would enter this information in the Session → Host Name and Session → Port fields. In the examples given earlier for OpenSSH (Section A.3.1, “Tunneling with OpenSSH”), a connection to **`localhost`** port **`2022`** would give you an ssh connection to directly to clusterInternalNode. And the Ccaffeine GUI would be invoked in the same way as above (modulo unix vs. Windows details in the command itself).

Appendix B. Building the CCA Tools and TAU and Setting Up Your Environment

$Revision: 1.23 $
$Date: 2007/09/24 03:45:05 $

The primary tools you'll be using are the Ccaffeine CCA framework [http://www.cca-forum.org/ccafe/] and the Babel language interoperability tool [http://www.llnl.gov/CASC/components/babel.html]. This section provides brief instructions on how to download and install a distribution of these tools (named, creatively enough, “cca-tools”) that has been tested for compatibility with the tutorial code.

Caution

These tools are still under development as we extend their capabilities. Consequently, it is possible to find numerous releases and snapshots of the individual tools, any given combination of which may not have been tested for compatibility. *Don't use* the individual tool distributions unless you've got a particular reason, usually based on direct conversations with their developers. The latest version of the “cca-tools” package is the recommended distribution for routine use and will provide you with a matched set of tools that will work together properly.

The TAU performance observation tools [http://www.cs.uoregon.edu/research/paracomp/tau/tautools/] can be used in conjunction with the CCA to provide simple instrumentation and monitoring at the level of component interfaces (and of course it can be used to instrument a component internally just like any other piece of code). If you wish to use TAU it will also be necessary for you to install it on your system.

B.1. The CCA Tools

B.1.1. System Requirements

Note

We strongly recommend using a Linux platform to work through these exercises, since this is currently the most extensively tested and most easily supported platform for the CCA tools. If this is not possible, or you have a specific need to use another platform while working through these exercises, please contact us at `<help@cca-forum.org>` to discuss the best way to proceed. We're also interested to hear what platforms you would like to run your CCA applications on in the longer term in order to help us focus our porting and testing efforts.

The requirements to build the CCA tools on Linux platforms are listed below. Requirements for other platforms will vary somewhat.

- gcc >= 3.2
- Java Software Development Kit >= 1.4. The **java** and **javac** commands must be in your execution path.

Note

We have on occasion observed problems with the ccafe-gui interface hanging (most often while populating the *palette* as the GUI starts up). This seems to happen less often with version 1.4 than with more recent versions.

- A connection to the internet. (A network connection is required both to download the code cca-tools package and during the build process.)
- Python >= 2.3 built with **--enable-shared** (on platforms that support shared libraries), and Numerical Python (NumPy). If you have multiple versions of Python installed and prefer to have a version in your execution path that does *not* meet the criteria above, you should set the `PYTHON` environment variable to point to a suitable version for the CCA tools prior to configuring them. You can check the python version with **python `-V`**.

Additional Optional Software. There are also a number of other packages which are not *required* in order to build the CCA tools, but can be used if present (and may be required in order to obtain certain functionality). If you want to use them, they should be installed before you begin to install the CCA tools.

- MPI: recent versions of MPICH are known to work. At present, the automatic configuration tools do not handle other MPI implementations, and Ccaffeine has not yet been extensively tested against other implementations.

Note

At present, there are no exercises that require MPI.

- Fortran 90: A variety of Fortran 90 compilers are supported. Because Babel needs to know about the format of the array descriptors used internally by the compiler, the CCA tools will have to be configured with both the path to the compiler and information about which compiler it is. Here is the list of currently supported compilers and the associated labels recognized by the CCA tools configuration script.

Compiler	CCA Tools “VENDOR” Label
Absoft	Absoft
HP Compaq Fortran	Alpha
Cray Fortran	Cray
GNU gFortran	GNU
IBM XL Fortran	IBMXL
Intel v8	Intel
Intel v7	Intel_7
Lahey	Lahey
NAG	NAG
SGI MIPS Pro	MIPSpro
SUN Solaris	SUNWspro

You should have the compiler in your execution path, and any relevant `.so` libraries in your

LD_LIBRARY_PATH. These are required to properly configure the CCA tools package.

- GNU autotools >= 2.59; >= 2.60 recommended. These are not required by the CCA tools themselves, but would be needed if your development activities require adding to the basic `configure` script generated by bocca.

B.1.2. Downloading and Building the CCA Tools Package

1. The latest version of the CCA Tools package can be found at http://www.cca-forum.org/tutorials/#sources with a filename of the form `cca-tools-`*`version`*`.tar.gz`.
2. Untar the cca-tools tar ball some place that is convenient to build and follow the instructions in the README to build it.

The CCA tools build procedure has been tested on a variety of systems with a range of different configuration options, and it works the majority of the time. However it is possible your platform or configuration requirements will confuse it, and it will not build properly for you. If this happens, please contact us at <help@cca-forum.org> with the output of your attempt to configure and build the package, and any pertinent information about your system. We want to help you get a working CCA environment and improve the packaging of the tools for future users.

B.2. The Ccaffeine GUI

The Ccaffeine front-end GUI is part of the CCA tools distribution you installed above. But if you're running the exercises on a remote system and want to use the GUI (it is *not* required to complete the exercises), you will need to download and setup the GUI on your local system before you can use it. (It will work over an X11 connection to the remote system, if you have one, but we tend to find performance of Java tools like the GUI unacceptable and generally recommend running it locally and connecting to the remote system via an SSH tunnel, as described in Section A.3, “Tunneling other Connections through SSH”.)

B.2.1. System Requirements

These requirements apply to both Linux-like and Windows systems.

- Java Software Development Kit >= 1.4. The **java** command must be in your execution path.

B.2.2. Downloading and Setting Up the GUI

1. To use the GUI on your local system, you will need to download the `ccafe-gui.jar` and the convenience script to run it. The script to download depends on which operating system you're local system is running. For Linux-like systems, it is `simple-gui.sh`, and for Windows systems, it is `simple-gui.bat`. The files could be copied (using **scp**) from the CCA tools installation on the remote system (in the `$CCA_TOOLS_ROOT/bin` subdirectory), or (probably more conveniently) downloaded from http://www.cca-forum.org/tutorials/#sources.

2. The scripts expect to be located in the same directory as the `jar` file. Instructions for using the scripts can be found in Section 2.1, "Using the GUI Front-End to Ccaffeine".

B.3. Downloading and Installing TAU

> **Note**
>
> Note that TAU is not currently used in any of the exercises (we're working on changing that). Everything in this Guide will work fine without it.

1. The latest version of the TAU Portable Profiling package can be found at http://www.cs.uoregon.edu/research/paracomp/tau/tautools/. Also needed for the CCA environment is the Performance component, available at http://www.cs.uoregon.edu/research/paracomp/proj/tau/cca/.

2. Untar the `tau_`*`version`*`.tar.gz` file in a convenient place.

3. Next, configure TAU with **`./configure`** ***`options`***. You can specify an installation prefix with the `-prefix=`*`TAU_ROOT`* option (the default is use the directory in which you *build* TAU). There are many other configuration options available (type **`./configure -help`** for a complete list).

   > **Note**
   >
   > In these exercises, MPI is not needed, but if you want to build TAU with it, you'll need to use the `-mpiinc` and `-mpilib` options. Also, for these exercises, TAU does *not* need to be compiled with Fortran support. Fortran support would be required to work with Fortran code you directly instrument. In these exercises, you will be using TAU via a the TAU performance component, which is written in C++.

4. Build TAU using **`make install`**

5. Untar the `performance-`*`version`*`.tar.gz` file someplace convenient to build.

6. Configure the performance component using **`./configure -ccafe=`*`CCA_TOOLS_ROOT`*` -taumakefile=`*`TAU_ROOT`*`/include/Makefile -without-classic -without-proxygen -ccatk=`*`TAU_CMPT_ROOT`***. *`CCA_TOOLS_ROOT`* and *`TAU_ROOT`* are the installation roots for the CCA tools and TAU that you specified in previous steps. *`TAU_CMPT_ROOT`* is the directory into which you want the performance component tools installed.

7. Build the performance component using **`make ; make install`**

B.4. Setting Up Your Login Environment

Once the CCA tools (and TAU, if needed) have been built, you will need to setup your login environment so that the appropriate commands are added to your execution path, and libraries are added to your `LD_LIBRARY_PATH`.

Wherever you installed the tools above, we will use the following notation in this section:

CCA_TOOLS_ROOT — The *fully qualified* path to where the CCA tools were installed (the `--prefix` directory, or the default `./local` expanded to be complete paths, rather than relative)

TAU_ROOT — The *fully qualified* path to TAU's install directory (the `-prefix` directory)

TAU_CMPT_ROOT — The *fully qualified* path to the TAU performance component (the `-ccatk` directory).

Then the following commands should work, depending on which shell you use:

csh, tcsh and Related Shells.

```
set path=(CCA_TOOLS_ROOT/bin TAU_ROOT \
          TAU_CMPT_ROOT $path)
setenv LD_LIBRARY_PATH CCA_TOOLS_ROOT/lib:$LD_LIBRARY_PATH
```

bash, ksh, sh and Related Shells.

```
export PATH=CCA_TOOLS_ROOT/bin:TAU_ROOT:TAU_CMPT_ROOT:$PATH
export LD_LIBRARY_PATH=CCA_TOOLS_ROOT/lib:$LD_LIBRARY_PATH
```

These commands could be added to your own login files (`$HOME/.cshrc` or `$HOME/.profile`), put in a file somewhere else and sourced in your login files (this is the approach we use in the organized tutorials), or, if appropriate, added to the system login setup by your system administrator.

Tip

If you're a participant in an organized tutorial, we've already prepared a login file with these commands, and others needed for the tutorial, which you simply source in your login file. Specific instructions on how to set this up should have been provided to you along with your tutorial account information.

If you are using Python, you also need to set your `PYTHONPATH` environment variable to include the locations of Python modules associated with the CCA tools and the tutorial itself.

csh, tcsh and Related Shells.

```
setenv PYTHONPATH CCA_TOOLS_ROOT/lib/python2.3/site-packages/:\
$TUTORIAL_SRC/ports/lib/python:\
$TUTORIAL_SRC/components/lib/python
```

bash, ksh, sh and Related Shells.

```
export
PYTHONPATH=CCA_TOOLS_ROOT/lib/python2.3/site-packages/:\
$TUTORIAL_SRC/ports/lib/python:\
$TUTORIAL_SRC/components/lib/python
```

Unfortunately, because of the way Python works, you will have to modify the `PYTHONPATH` any time you add new Python components to your application.

Appendix C. Building the Tutorial Code Tree

$Revision: 1.2 $
$Date: 2007/09/20 11:46:17 $

The file `tutorial-src-`*`version`*`.tar.gz` at http://www.cca-forum.org/tutorials/#sources has the full code for all of the components created in this Guide as well as a number of others. These components are used in Chapter 2, *Assembling and Running a CCA Application* (once the they are built) to give you some experience working with existing components. In later chapters, the code itself can serve as a model and a reference for the components you're writing.

Note

At the time this particular version of the Hands-On Guide was generated, the *`version`* was 0.5.2_rc1. If there's a more recent version available, you should probably use it, but you should also look for a more current version of this Guide to go with it. Both should have the same base version number (i.e. 0.5.2) perhaps with different release numbers. Take the highest available release number. Note too that because both the CCA tools and the tutorial code are evolving over time, you should make sure to use the version of the CCA tools distribution that is recommended for the particular tutorial version you're working with.

If you're participating in an organized tutorial, we will have built the `tutorial-src` tree for you in advance in a common location, whereas if you're working through these exercises on your own, you'll need to build it yourself.

Tip

Make sure you've setup your login environment per Section B.4, “Setting Up Your Login Environment”. To complete the procedures in this section, you will need to have Babel and Ccaffeine in your execution path, and your `LD_LIBRARY_PATH`.

1. Download the file you need from the location above.

2. Untar the file in a convenient place with **`tar zxf tutorial-src-`*`version`*`.tar.gz`**. When it completes, change directories into the new code tree.

3. Run **`./configure`** to configure the tree for the build location.

4. The code tree includes components written in C, C++, F90, F77, Python, and Java. You may need to configure the code tree according to the languages you have available (dependent on how the CCA tools were built in Appendix B, *Building the CCA Tools and TAU and Setting Up Your Environment*). Run **`./configure --with-languages="`*`c cxx f77 f90 java python`*`"`** using the appropriate space-separated list of languages for your environment. The default is to include the languages for which Babel was configured when the CCA tools were installed (see Appendix B, *Building the CCA Tools and TAU and Setting Up Your Environment*).

5. Once the tree is configured, type **`make`** to build it. This step may take several minutes. At the end of the build output, you should see a list of components that were successfully built, such as:

```
SUCCESS building drivers.PYDriver
```

and when it finally completes, you should see this message:

```
################ Finished building everything #################
####### You can run some simple tests with 'make check' #######
```

If the build terminates with an error message instead, please ask for assistance.

6. Once the build is complete, you can type **`make check`** to perform a basic check that the component have been built correctly. This is a convenience of the `Makefile` system generated by bocca that tries to instantiate each component within the Ccaffeine framework. This provides a basic check that the software you've built are "well-formed" CCA components. You should see a message like this, along with a couple of lines of output from **make** itself:

```
### Test library load and instantiation for the following languages: c cxx f90
Running instantiation tests only
Test script: tutorial-src/components/tests/test_rc
==> Instantiation tests passed for all built components (see tutorial-src/compo
make[1]: Leaving directory `tutorial-src/components'
```

Appendix D. License (Creative Commons Attribution 2.5)

licly by means of a digital audio transmission the Work including as incorporated in Collective Works;

d. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission Derivative Works.

e. For the avoidance of doubt, where the work is a musical composition:

   i. **Performance Royalties Under Blanket Licenses.** Licensor waives the exclusive right to collect, whether individually or via a performance rights society (e.g. ASCAP, BMI, SESAC), royalties for the public performance or public digital performance (e.g. webcast) of the Work.

   ii. **Mechanical Rights and Statutory Royalties.** Licensor waives the exclusive right to collect, whether individually or via a music rights agency or designated agent (e.g. Harry Fox Agency), royalties for any phonorecord You create from the Work ("cover version") and distribute, subject to the compulsory license created by 17 USC Section 115 of the US Copyright Act (or the equivalent in other jurisdictions).

f. **Webcasting Rights and Statutory Royalties.** For the avoidance of doubt, where the Work is a sound recording, Licensor waives the exclusive right to collect, whether individually or via a performance-rights society (e.g. SoundExchange), royalties for the public digital performance (e.g. webcast) of the Work, subject to the compulsory license created by 17 USC Section 114 of the US Copyright Act (or the equivalent in other jurisdictions).

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. All rights not expressly granted by Licensor are hereby reserved.

4. **Restrictions.** The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

   a. You may distribute, publicly display, publicly perform, or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Work with any technological measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement. The above applies to the Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the Collective Work any credit as required by clause 4(b), as requested. If You create a Derivative Work, upon notice from any Licensor You must, to the extent practicable, remove from the Derivative Work any credit as required by clause 4(b), as requested.

   b. If you distribute, publicly display, publicly perform, or publicly digitally perform the Work or any Derivative Works or Collective Works, You must keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or (ii) if the Original Author and/or Licensor designate another party or parties (e.g. a sponsor institute, publishing entity, journal) for attribution in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; the title of the Work if supplied; to the extent reasonably practicable, the Uniform Resource Identifier, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and in the case of a Derivative Work, a credit identifying the use of the Work in the Derivative Work (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). Such credit may be implemented in any reasonable manner; provided, however, that in the case of a Derivative Work or Collective Work, at a minimum such credit will appear where any other comparable authorship credit appears and in a manner at least as prominent as such other comparable authorship credit.

5. **Representations, Warranties and Disclaimer**

   UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. **Limitation on Liability.** EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. **Termination**

   a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Derivative Works or Collective Works from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

   b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. **Miscellaneous**

   a. Each time You distribute or publicly digitally perform the Work or a Collective Work, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

   b. Each time You distribute or publicly digitally perform a Derivative Work, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

   c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

Additional Information. For more information about the Creative Commons and this license, please see their web site, http://creativecommons.org.

Requested Attribution. CCA Forum Tutorial Working Group, *A Hands-On Guide to the Common Component Architecture, version 0.5.2_rc1*, 2007, http://www.cca-forum.org/tutorials/.

Or in BibTeX format:

```
@Manual{hog-cca:0.5.2_rc1,
  title =   {A Hands-On Guide to the Common Component Architecture},
  author =  {The Common Component Architecture Forum Tutorial
            Working Group},
  edition = {0.5.2_rc1},
  year =    2007,
  note =    {http://www.cca-forum.org/tutorials/}
}
```