A Hands-On Guide to the Common Component Architecture

Common Component Architecture Forum Tutorial Working Group

Version: 0.7.1

A Hands-On Guide to the Common Component Architecture

Common Component Architecture Forum Tutorial Working Group
Version: 0.7.1

Licensing Information

Requested Attribution

> Common Component Architecture Forum Tutorial Working Group, *A Hands-On Guide to the Common Component Architecture*, version 0.7.1, http://www.cca-forum.org/tutorials/.

Or in BibTeX format:

```
@Manual{cca-tutorial:0.7.1,
    title   =  {A Hands-On Guide to the Common Component Architecture},
    author  =  {Common Component Architecture Forum Tutorial Working Group},
    edition =  {0.7.1},
    year    =  2010,
    URL     =  {http://www.cca-forum.org/tutorials/}
}
```

Contents

Preface

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.

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 cca-tutorial@cca-forum.org [`mailto:cca-tutorial@cca-forum.org`] with your comments and questions.

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 [`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.

Typographic Conventions

- `This style` is used for file and directory names.
- `$ This style` is used for user-issued shell commands.
- **`This style`** is used for code 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 are presented in square brackets after the name of the resource they describe in the print version of this Guide [`http://www.cca-forum.org/tutorials/#sources`].

- 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.

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 E and Appendix F). 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 E.4).

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 E.1.)

TAU_ROOT **($TAU_ROOT)** The installation location of the TAU Portable Profiling package. (See Section E.3.)

TAU_CMPT_ROOT **($TAU_CMPT_ROOT)** The installation location of the TAU performance component. (See Section E.3.)

TUTORIAL_SRC **($TUTORIAL_SRC)** The top of the tree where the "ODE" part of the tutorial code has been built. (See Appendix F.) In an organized tutorial, this is likely to a central location shared by all students, which may not be modified.

***STUDENT_SRC* (`$STUDENT_SRC`)** The top of the student's private copy of the ODE code tree. If you're doing this tutorial on your own, *STUDENT_SRC* can be the same as *TUTORIAL_SRC*, but if you part of an organized tutorial, the *STUDENT_SRC* tree is the one you can modify and rebuild.

***PDE_SRC* (`$PDE_SRC`)** The location containing the PDE example code tree. (See Appendix F.) In an organized tutorial, this is likely to a central location shared by all students, which may not be modified.

***PDE_STUDENT_SRC* (`$PDE_STUDENT_SRC`)** The student's private copy of the PDE code tree, analagous to *STUDENT_SRC*.

***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. If you're part of an organized tutorial, you will be assigned a *WORKDIR*.

Warning

If you're part of an organized tutorial please be careful to *use the* `WORKDIR` *you are assigned*! Often there are special considerations in such an environment, which might not be obvious to you as a participant. For example, it is fairly common for all cluster nodes to mount user home directories from a single NFS file server. An entire class of students working on I/O-intensive activities (like building the tutorial code) at the same time has been known to kill servers from time to time. So frequently, you will be asked to use directories local to your assigned cluster node.

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

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

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, you will use pre-built components to assemble and run several different numerical integration applications.
- In Chapter 3, you will construct your own components for the numerical integration example, using the `bocca` tool.
- In Chapter 5, you will use the TAU performance observation tool [`http://www.cs.uoregon.edu/research/paracomp/tau/tautools/`] to automatically instrument a component interface and monitor the performance of the application.
- In Chapter 6, you will see examples of how to work with arrays in a multi-language environment, including writing your own component. (Languages: F77, F90, C)

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

In Chapter 2, you'll be working with a complete version, pre-built of the tutorial code tree. Then in Chapter 3 you'll start from scratch to create components on your own, replicating those in Chapter 2. 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 cca-tutorial@cca-forum.org [`mailto:cca-tutorial@cca-forum.org`].

Tip

Some of 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 [`http://www.cca-forum.org/tutorials/#sources`]) to cut and paste the necessary inputs. *Note, however, that not everything can be cut-and-pasted directly.* Take particular care with lines that had to be broken for purposes of documentation, and for placeholder values such as "*`TUTORIAL_SRC`*".

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:

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

Bocca A tool for generating and manipulating projects using CCA components or other SIDL based code. Bocca is designed to simplify the tedious and mechanical aspects of CCA and Babel. 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.

Ccaffeine A CCA framework which emphasizes local and parallel high-performance computing, and the most common CCA framework in real applications. For more information, see the Ccaffeine page [`http://www.cca-forum.org/ccafe/`].

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 (see Section 1.2.1) or participating in an organized tutorial (see Section 1.2.2), this may include getting logged in to a remote system, preparing the CCA environment, and building the tutorial code. Once you've setup everything as outlined below, you should be ready to proceed to Chapter 2.

1.2.1 For Self-Study Users

Getting Connected: 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 D.

Preparing the CCA Environment: 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, following the instructions in Appendix E.

Building the Tutorial Code: You'll also need to download and build the tutorial code tree following the instructions in Appendix F.

1.2.2 For Organized Tutorial Participants

If you're participating in an organized tutorial, most of the preparatory work will have been done in advance by the tutorial instructors. Usually, they will provide you with a separate set of instructions tailored to the arrangements for the particular tutorial you're attending.

Getting Connected: Look to the separate handout or the tutorial instructors for details of your account, your machine assignment, etc. Appendix D should give you sufficient information to get logged in to the remote machine. If you have any problems, ask the tutorial instructors.

Preparing the CCA Environment: In this case, the CCA tools (Ccaffeine , Babel , and Bocca) will already have been built in a common area. The handout or tutorial instructors will provide you with the procedures you might need to follow to setup your environment (the `PATH` and other environment variables). Some general notes can be found in Section E.4.

Building the Tutorial Code: Once again, the tutorial code will already have been built in a central location, and the details will be noted on the handout.

Tip

In some of the later exercises (starting with Section 4), you will need to build your own copy of (parts of) the tutorial code tree so that you can modify them. Depending on your hardware environment, this may be pretty time consuming. If you're pressed for time you may want to go ahead and start building your own copy of the PDE and then ODE portions of the tutorial code tree before starting the first exercise. Follow the instructions in Appendix F, and you probably want to launch the build in a separate window. The tutorial instructors should be able to tell you whether this "pre-build" step is necessary.

Chapter 2

Assembling and Running a CCA Application

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 shown in Table 2.1.

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.[1] 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) and later discuss how actions in the GUI map onto instructions in a script (see Section 2.2).

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.

[1] It is also possible to control the assembly and execution of a CCA application entirely from within a program using the `BuilderService` interface, a standard part of the CCA specification implemented by all CCA-compliant frameworks, including Ccaffeine . This approach is not interactive (unless of course your program makes it so), and is considered advanced CCA usage.

Table 2.1: Integrator Application Components (details may vary depending on the languages your CCA tools installation is configured for).

Components	Notes
Drivers	
`drivers.CXXDriver`	Identical functionality, illustrating
`drivers.F90Driver`	components in different languages.
`drivers.PYDriver`	
Integrators	Various integration algorithms
`integrators.Boole`	
`integrators.MonteCarlo`	
`integrators.Midpoint`	
`integrators.Simpson`	
`integrators.Simpson38`	
`integrators.Trapezoid`	
Functions	
`functions.CosFunction`	$\cos(x)$; integrates to $\sin(1) \approx 0.841$
`functions.CubeFunction`	x^3; integrates to 0.25
`functions.LinearFunction`	x; integrates to 0.5
`functions.PiFunction`	$\frac{4}{1+x^2}$; integrates to π
`functions.QuinticFunction`	$x^5 - 4x^4$; integrates to $\frac{1}{6} - \frac{4}{5} \approx -0.633$
`functions.SquareFunction`	x^2; integrates to $\frac{1}{3}$
Random Number Generators	
`randomgens.RandNumGenerator`	Required for Monte Carlo integration

2.1 Using the GUI Front-End to Ccaffeine

Note

At this point, you will start using the `tutorial-src` code tree. If you're doing this tutorial as a self-study exercise, you'll need to make sure it has been built according to the instructions in Appendix F. For organized tutorials, this is generally done in advance by the tutorial instructors.

There is a graphical front-end for Ccaffeine (known as Ccaffeine GUI , or "the GUI" which provides a fairly simple visual programming metaphor for the assembly of applications using CCA components. 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 project's `utils` subdirectory, 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 Running the GUI Locally (*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, launches both Ccaffeine and the GUI, and automatically initializes the framework with a *palette* consisting of all of the components in the Bocca project.

Tip

Always make sure you're using the `run-gui.sh` script generated for the particular project you're working on because they'll be initialized with different sets of components. In this chapter, the command is ***$TUTORIAL_SRC*/utils/run-gui.sh**, but in later exercises it will be different!

2.1.2 Running the GUI Remotely (*GUI host* and *Ccaffeine host* are Distinct)

There are two different mechanisms to run the GUI remotely:

Using X11 on *GUI host* : If you have an X11 server on *GUI host* , you can use the **run-gui.sh** command on *Ccaffeine host* and let X11 take care of the graphics. However many users find the performance unacceptable using this approach, especially over slower network connections. In that case, try the method below.

Using a socket connection: In this case, you run the GUI locally on *GUI host* (it is a Java application, and works on Linux/unix, Mac, and Windows platforms) and connect to the remote framework on *Ccaffeine host* over a TCP/IP socket.

Tip

Connections between the GUI and the framework can be tunneled through an ssh connection. This may help in cases where firewalls or other network setups that prevent direct point to point connections (i.e. cluster compute nodes accessible only through the head node). See Appendix D and in particular Section D.3.

Note

This procedure requires that you have GUI on your *GUI host* . This includes the `simple-gui.sh` (on Windows, `simple-gui.bat`) script and the `ccafe-gui.jar` file. In a normal build of the CCA tools, you can find these files in the `$CCA_TOOLS_ROOT/lib` directory. If you're participating ain an organized tutorial, they also have been made available on a web site or other more convenient location. No special installation procedure is required, but the script and the `jar` file do need to be in the same directory.

On *Ccaffeine host*: First, launch the framework, using the Bocca -generated utility script for your project, telling it what port to listen on for the connection from the GUI: **utils/bocca-gui-backend.sh --port *port_num***. The script automatically initializes the framework with a *palette* consisting of all of the components in the Bocca project.

The port number must be in the range 1025–65535, and cannot be in use by another application on the host (if it is, you will get an error message; simply try another port). *You must use the same port number for the framework and the GUI.* In organized tutorials, you may be assigned a port to help reduce the chances of collisions. If so, please use it!

Tip

Always make sure you're using the `bocca-gui-backend.sh` script generated for the particular project you're working on because they'll be initialized with different sets of components. In this chapter, the command is ***$TUTORIAL_SRC*/utils/bocca-gui-backend.sh**, but in later exercises it will be different!

On *GUI host*: Start the GUI locally by executing the `simple-gui.sh` (on Windows, `simple-gui.bat`) script, specify *both* the host and port the framework is listening on: **simple-gui.sh --port *port_num* --host *Ccaffeine _host***.

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 Ccaffeine GUI interface hanging (most often while populating the *palette* as the GUI starts up). We think this is due to a subtle race condition in the GUI, which we haven't yet been able to isolate. The best advice seems to be to kill both the framework and GUI and try launching them again.

Other Ways to Launch the GUI and Ccaffeine (OPTIONAL READING)

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 *`$TUTORIAL_SRC`*`/utils/run-gui.sh` as described in Section 2.1.1 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:

   ```
   my rank: -1, my pid: 9625
   Type: Server
   ```

2. Run **`simple-gui.sh --port`** ***`port_num`*** **`--host`** ***`backend_host`*** (on Windows, **`simple-gui.bat`**) 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 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 Figure 2.1.

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.

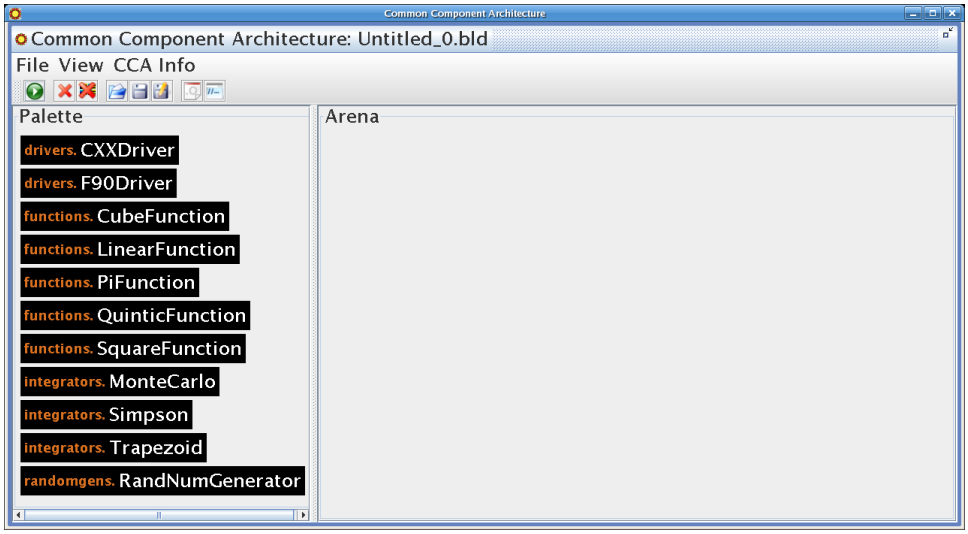

Figure 2.1: GUI with components in *palette* and empty *arena* (Step 2).

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.

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 Figure 2.2.

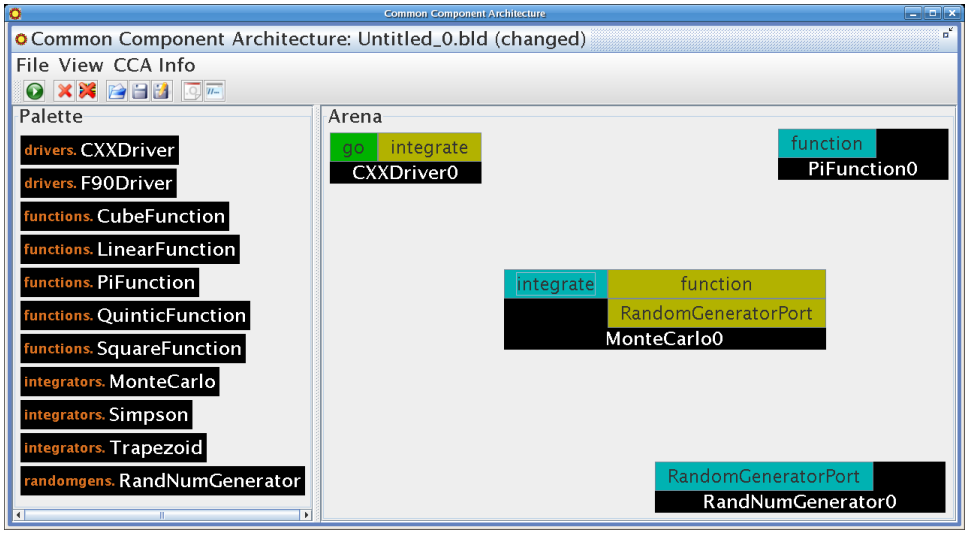

Figure 2.2: GUI with several components instantiated in *arena* (Step 4).

Tip

You can drag components around the *arena* to arrange them as suits you – just click on the black area of the component 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 (Figure 2.3).

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).

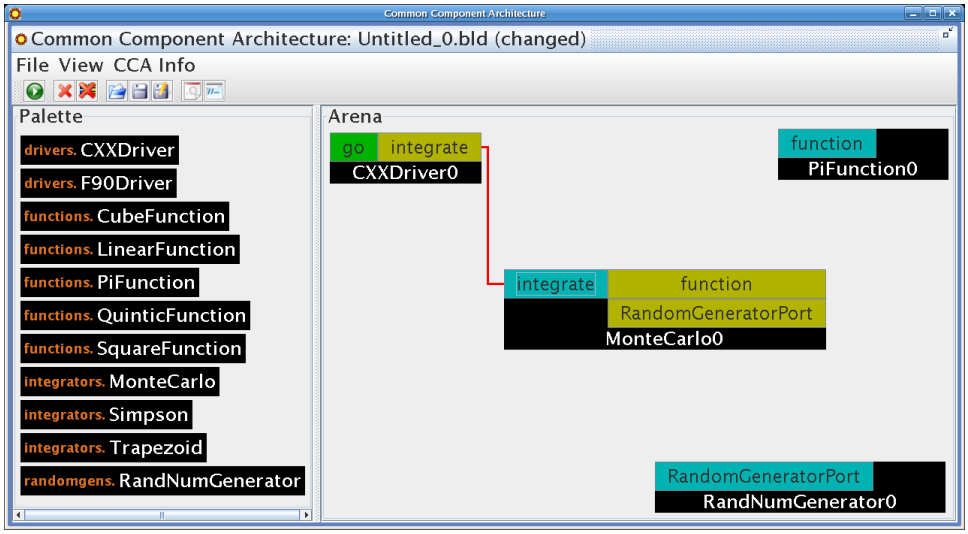

Figure 2.3: Driver's integrator port connected to integrator's integrator port (Step 5).

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 Figure 2.4.

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`"[2] and the message

   "`IN: ##specific go command successful`" in the *`GUI host`* terminal.

Tip

Remember that your application is running within the framework. Unless the application itself does something special, the output from the application will appear in the window in which the framework is running.

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

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

[2] 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 reasonably close to π

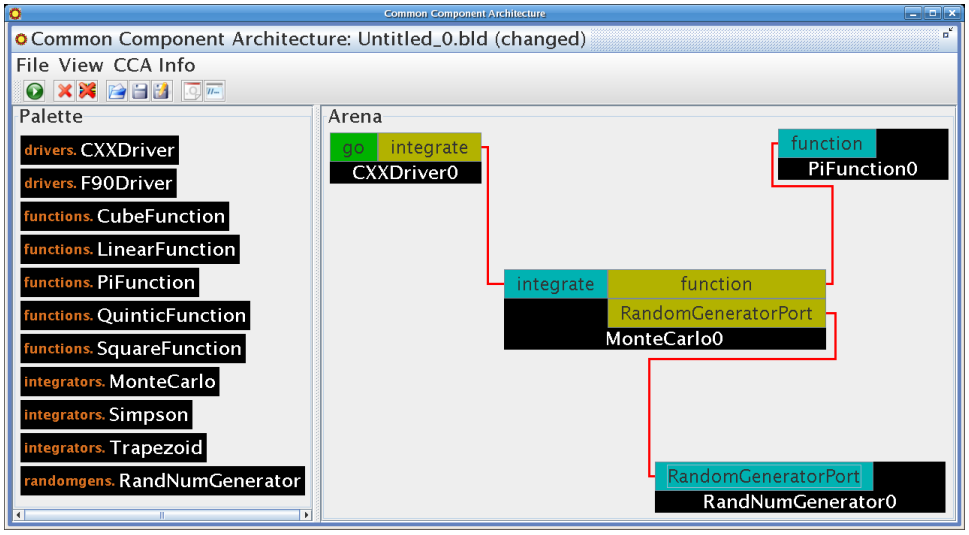

Figure 2.4: Fully connected application (Step 6).

components. Since they're already in the *palette* , you can instantiate them in the same way as Step 3. Your GUI should look like Figure 2.5.

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.[3] You will need to break the following connections:

[3] A bug in the GUI causes this dialog box to appear twice sometimes. Just respond appropriately both times.

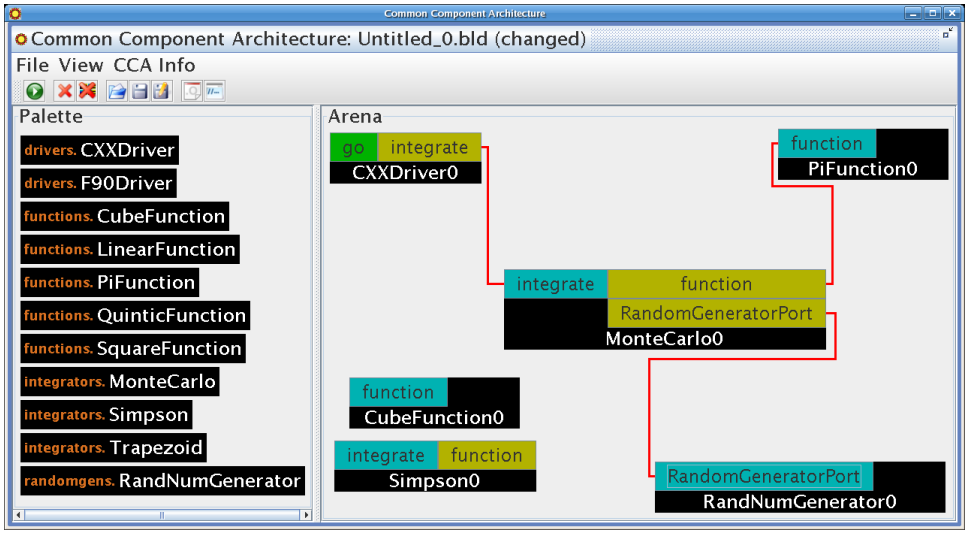

Figure 2.5: Additional components instantiated (Step 8).

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

There is no need to remove unused components from the *arena* – as long as they are not connected to active components, they will not interfere.[4] In fact in this case, neither `MonteCarlo0` nor `RandNumGenerator0` are used, so it is safe to leave them connected to each other.

Note

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

10. Assemble the new application (Figure 2.6) 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` (Figure 2.7). When you click on the `go` you should get "`Value = 0.250000`"[5] in the *`Ccaffeine host`* terminal.

[4] A bug in the GUI makes it impossible to remove them anyway.
[5] With a deterministic integrator, the result should be repeatable

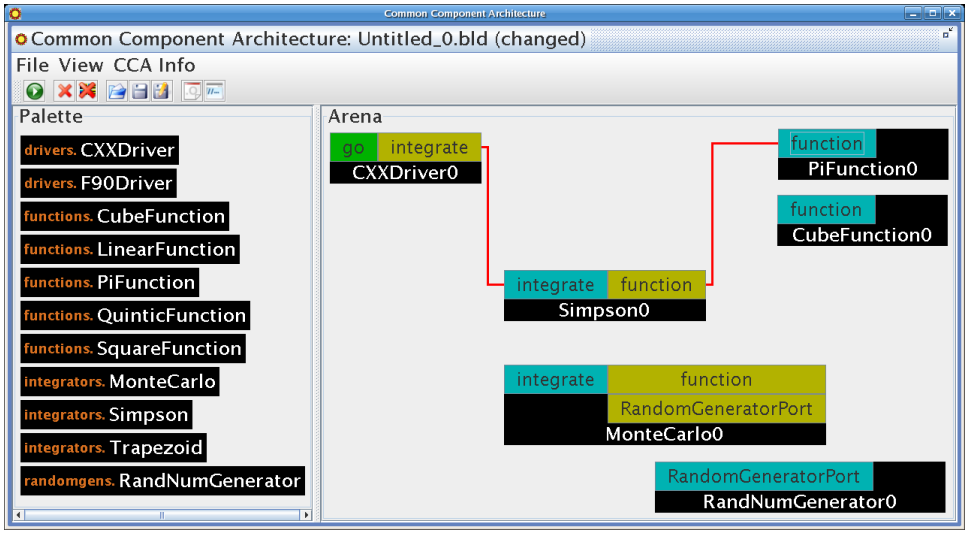

Figure 2.6: Another application, with additional unused components still in *arena* (Step 10).

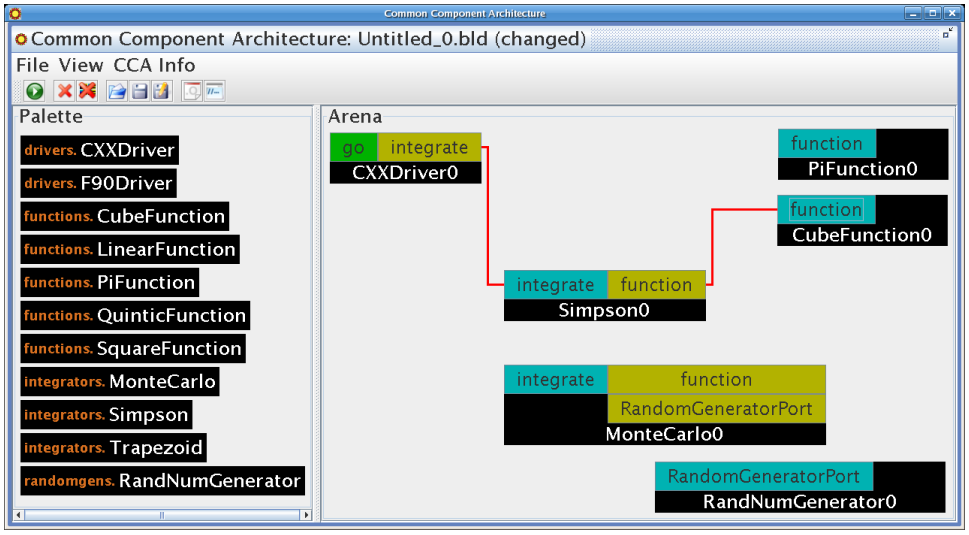

Figure 2.7: A third application, with extraneous components still in *arena* (Step 11).

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 $\Rightarrow$ 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 $\Rightarrow$ Detach option to close the GUI but leave the backend running. *However it is currently impossible to reattach to a running session.*

Tip

If the backend crashes while the GUI is running, exit the GUI by using Detach. Trying to Quit without a running backend will cause the GUI to hang.

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. These commands are part of the CCA tools installation, and should be in your path if you've followed the procedures in Appendix E.4.

1. Change directories to your *WORKDIR* 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 2>&1
   ```

   (if you're using the csh or tcsh shell, the output redirection should be "**>& task0.out**" instead of "**> task0.out 2>&1**").

   View the task0.out file satisfy yourself that the script ran. (Of course you can view the script itself too, 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 appearing in the script comments should correspond to the steps in the preceding 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/lib
   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.

As you saw in the GUI, Ccaffeine 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/lib
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/guitest.gen.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 MonteCarlo0
Component FRAMEWORK of type Ccaffeine-Support
Component MonteCarlo0 of type integrators.MonteCarlo
 is using function connected to Port: function provided by component PiFunction0
 is using RandomGeneratorPort connected to Port: RandomGeneratorPort provided by component RandNumGenerator0
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 by component RandNumGenera
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 PiFunction0

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 by component RandNumGenera
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 CubeFunction0

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 2.2. 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 : A Project Manager for SIDL or CCA

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 components and a build system. Naturally Bocca cannot create your implementation for you, but all of the glue code for multi-language interoperability and component interfaces in a CCA application is created and maintained with a few commands. The advantage of this approach is that a lot of build and component defaults have been chosen for you. The downside is that, while some customization is possible, the project directory and file structures are largely predetermined.

3.1 Creating a Bocca Project

If your CCA environment is configured properly (Section E) 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 `demo`. Create the project directory now:

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

```
The project was created successfully in /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo
```

Here *LANG* specifies the *default* implementation language for components in this project, if you don't specifically indicate a language when creating the component (a project can contain components in any mixture of Babel -supported languages). For this exercise, choose the one of **`c`**, **`cxx`**, **`f90`**, **`java`**, or **`python`** with which you are most comfortable (some of these choices may not be available if your Babel installation is not configured for them, but these are the languages for which this Guide has detailed instructions). If you don't specify a default language when creating the project, Bocca will use C++.

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 -F
```

```
demo/
```

Feel free to poke around a bit:

```
$ ls -F demo
```

```
BOCCA/
buildutils/
components/
config/
configure*
configure.ac*
configure.ac1
depl/
external/
install/
Makefile
make.project
make.project.in
make.rules.user
make.vars.user
ports/
README
utils/
```

Before using a new Bocca project or working with an existing project just checked out from a source code repository, you will need to configure it for the details of your local environment. For a new project this is easy: `./configure` from within your new project directory.

```
$ cd demo && ./configure
```

```
checking for bash... /bin/sh
checking for GNU make... make
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... /usr/local/ACTS/cca/bin/bocca
c cxx f90 f77 python java
configure: Configuring with languages: c cxx f90 f77 python java
configure: Project source dir apparently /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/dem
configure: Using 4 processe(s) in calls to make.
checking whether make sets $(MAKE)... yes
configure: creating ./config.status
config.status: creating make.project
config.status: creating buildutils/make.vars.common
config.status: creating utils/run-gui.sh
config.status: creating utils/bocca-gui-backend.sh
config.status: creating utils/demo-config
config.status: creating utils/config-data
config.status: creating utils/demo-config.h
config.status: executing outmsg commands
```

3.2 Creating Ports and Components

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

```
$ pwd
```

```
/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo
```

It is important to be in the project directory (or its subdirectories) 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
```

```
Babel updating the cxx implementation of component demo.emptyComponent ...
```

Notice that Bocca selected `demo` as the default package name for `emptyComponent` since no package name was specified when creating the component. Normally, Bocca uses the project name as the package name for both ports and components unless a different default package name was specified when the project was created. We have named our component `emptyComponent` because it has no *uses* nor *provides* ports and thus is rather uninteresting. Nonetheless, Bocca has generated all of the necessary make system scaffolding and code for the component, including the `setServices`call. Listing the directory shows the files Bocca has generated (shown here for *LANG* =**cxx**):

```
$ ls components/demo.emptyComponent
```

```
BOCCA
demo_emptyComponent_Impl.cxx
demo_emptyComponent_Impl.hxx
demo_emptyComponent_Impl.hxx.rej
glue
Makefile
make.rules.user
make.vars.user
```

Components created in Fortran, C, and Python will contain a similar set files appropriate to the language. In the `components` directory a new directory, `demo.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: `demo_emptyComponent_Impl.cxx`, `demo_emptyComponent_Impl.hxx` with some Babel glue code in the `glue` subdirectory. Note the file ending in .rej named `demo_emptyComponent_Impl.hxx.rej`. This file is produced by the Bocca splicing process. It records code fragments that Bocca discarded while generating `demo_emptyComponent_Impl.hxx` and can usually be ignored and even deleted.

An Empty Component in Ccaffeine (OPTIONAL READING)

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

```
$ make
```

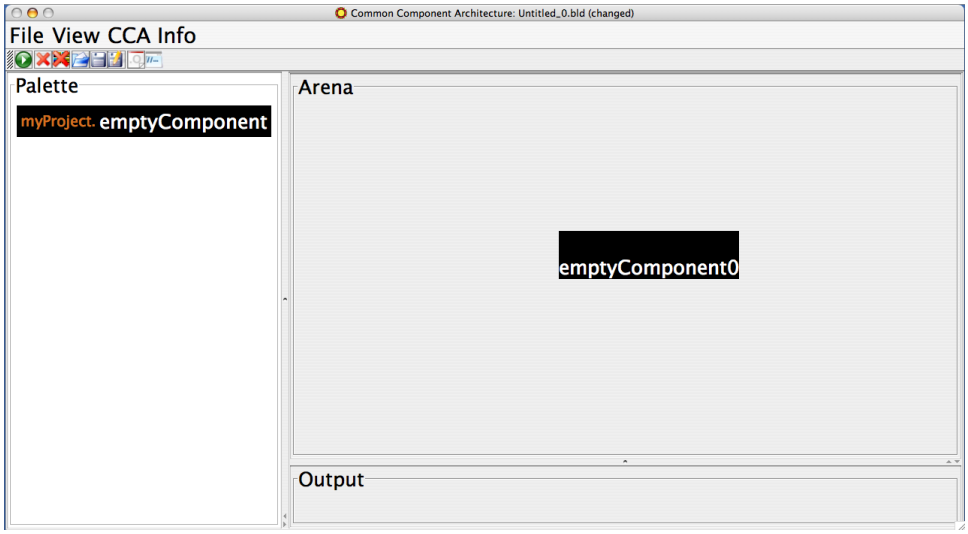

Figure 3.1: GUI showing the `emptyComponent` generated in Section 3.2 instantiated in the *arena* .

```
make[1]: Entering directory '/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo'
# ============================================================================
# No SIDL files in external/sidl, skipping build for external
# ============================================================================
# ============================================================================
# No SIDL files in ports/sidl, skipping build for ports
# ============================================================================
# ============================================================================
# Building in components/clients/, languages: cxx
# ============================================================================
 ## Building clients...
# ============================================================================
# Building in components/, languages: cxx
# ============================================================================
   [s] Building class/component demo.emptyComponent:
   [s] using Babel to generate cxx implementation code from demo.emptyComponent.sidl...
   [s] compiling sources...
   [s] creating class/component library: libdemo.emptyComponent.la ...
   [s] finished libtooling: components/demo.emptyComponent/libdemo.emptyComponent.la ...
   [s] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/install/share/cca/demo/
   [s] creating Ccaffeine test script (components/tests/instantiation.gen.rc)...
Build summary:
SUCCESS building demo.emptyComponent
### To test instantiation of successfully built components, run 'make check' ###
################ Finished building everything #################
####### You can run some simple tests with 'make check' #######
make[1]: Leaving directory '/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo'
```

(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. If you instantiate the `emptyComponent` , you should see something similar to Figure 3.1. Of course it lacks any *uses* or *provides* ports and thus cannot be used for anything, but it *is* a full-fledged CCA component.

3.2.1 Creating the Integrator and Function Components

In order to have some exportable or importable functionality in a component we must have some *uses* and *provides* ports. Bocca will also create the scaffolding and code for ports. Following the model of the integrator application of Chapter 2, we will create a `Function` , an `Integrator` , and a `Driver` . However before we can do that we will have to create some ports for these components to *use* and *provide* .

Let's begin by creating a `FunctionPort` and an `Integration` :

```
$ bocca create port Integration
```

```
Updating makefiles (for demo.Integration)...
```

```
$ bocca create port FunctionPort
```

```
Updating makefiles (for demo.FunctionPort)...
```

Notice that we are continuing to use the default package `demo`, though we could specify something different.

Now, create a set of components similar to those that you used in Chapter 2, specifying that they will *provide* or *use* the appropriate ports:

```
$ bocca create component Function --provides=FunctionPort@fun
```

```
Babel updating the cxx implementation of component demo.Function ...
```

```
$ bocca create component Integrator \
  --provides=Integration@integrate \
  --uses=FunctionPort@odeRHS
```

```
Babel updating the cxx implementation of component demo.Integrator ...
```

```
$ bocca create component Driver --go=run \
  --uses=Integration@integrate
```

```
Babel updating the cxx implementation of component demo.Driver ...
```

This last `bocca create` decorates our `Driver` component with a CCA standard `go` port, which is not specified as part of this project. Since `gov.cca.ports.GoPort` is a part of the CCA specification, Bocca takes care of knowing where to find the SIDL definition of this port. The special `--go` option allows Bocca to generate a default `go` implementation which prefetches the uses ports so that all the user needs to do for our example is add numerical code. In languages which are not object-oriented, this substantially reduces the errors in handling ports, exceptions, and memory deallocation.

Note

It is not necessary to know at component creation time all ports that will be used or provided or other implementation details. Bocca provides various commands for changing project entities, e.g. adding or removing *uses* and *provides* ports.

As we have defined a number of new things, this would be a good time to rebuild the project:

```
$ make
```

```
make[1]: Entering directory '/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo'
# ============================================================================
# No SIDL files in external/sidl, skipping build for external
# ============================================================================
# ============================================================================
# Building in ports/, languages: cxx
# ============================================================================
 ## Building ports...
   [c] using Babel to generate cxx client code for demo.FunctionPort...
   [c] creating library: libdemo.FunctionPort-cxx.la...
   [c] installing demo.FunctionPort.sidl
   [c] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/install/share/cca/demo/
   [c] using Babel to generate cxx client code for demo.Integration...
   [c] creating library: libdemo.Integration-cxx.la...
   [c] installing demo.Integration.sidl
   [c] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/install/share/cca/demo/
# ============================================================================
# Building in components/clients/, languages: cxx
# ============================================================================
 ## Building clients...
# ============================================================================
# Building in components/, languages: cxx
# ============================================================================
   [s] Building class/component demo.Driver:
   [s] using Babel to generate cxx implementation code from demo.Driver.sidl...
   [s] compiling sources...
   [s] creating class/component library: libdemo.Driver.la ...
   [s] finished libtooling: components/demo.Driver/libdemo.Driver.la ...
   [s] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/install/share/cca/demo/
   [s] creating Ccaffeine test script (components/tests/instantiation.gen.rc)...
   [s] Building class/component demo.Function:
   [s] using Babel to generate cxx implementation code from demo.Function.sidl...
   [s] compiling sources...
   [s] creating class/component library: libdemo.Function.la ...
   [s] finished libtooling: components/demo.Function/libdemo.Function.la ...
   [s] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/install/share/cca/demo/
   [s] creating Ccaffeine test script (components/tests/instantiation.gen.rc)...
   [s] Building class/component demo.Integrator:
   [s] using Babel to generate cxx implementation code from demo.Integrator.sidl...
   [s] compiling sources...
   [s] creating class/component library: libdemo.Integrator.la ...
   [s] finished libtooling: components/demo.Integrator/libdemo.Integrator.la ...
   [s] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/install/share/cca/demo/
   [s] creating Ccaffeine test script (components/tests/instantiation.gen.rc)...
   [s] Building class/component demo.emptyComponent:
doing nothing -- library is up-to-date.
Build summary:
SUCCESS building demo.Driver
SUCCESS building demo.Function
SUCCESS building demo.Integrator
### To test instantiation of successfully built components, run 'make check' ###
################ Finished building everything #################
####### You can run some simple tests with 'make check' #######
make[1]: Leaving directory '/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo'
```

Note that this operation can be very time-consuming when your project is managing many ports and components with the fully supported set of Babel language bindings.

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

```
$ make check
```

```
make[1]: Entering directory `/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo'
make --no-print-directory --no-builtin-rules -C components check
### Test library load and instantiation for the following languages: cxx
Running instantiation tests only
###
LDPATH=/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/install/lib:/usr/local/ACTS/cca/lib:/u
###
PYTHONPATH=/usr/local/ACTS/cca/lib/python2.5/site-packages:/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scr
###
CLASSPATH=/usr/local/ACTS/cca/lib/sidl-1.4.0.jar:/usr/local/ACTS/cca/lib/sidlstub_1.4.0.jar:/usr/local/ACTS/cca/lib/
###
Test script: /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/components/tests/instantiation.g
Log file: /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/components/tests/instantiation.gen.
SUCCESS:
==> Instantiation tests passed for all built components (see /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/s
make --no-print-directory --no-builtin-rules check-user
make[1]: Leaving directory `/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo'
```

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. If you were to run the GUI (Section 2.1) or do the command-line equivalent in Ccaffeine (Section 2.2), 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 ports). But if you attempted to run an application using these components, nothing would happen because they're only skeletons, providing the component-ness, but not the functionality.

Note

Along with everything else it does, Bocca generates a set of utility scripts which are tailored to the specific Bocca project (i.e. having the right paths, the right sets of components, etc.). In Section 2 you ran Bocca-generated utility scripts in the pre-built tutorial tree (i.e. **`$TUTORIAL_SRC/utils/run-gui.sh`**). When you're working in your own Bocca projects, make sure you use the utility scripts associated *with that project*, or things won't work properly.

3.3 How to Edit and Find Files in Bocca Projects

The next step in developing components requires you implement the component's intended functionality within the Bocca -generated skeleton. There are two places that we have to change things to make that happen: add methods to the interface definitions (`.sidl` file) and then put the implementation code into the components in the language chosen in Section 3.1.

Because Bocca generates all the files in the project, it knows where to find the code associated with each SIDL symbol. Using the `bocca edit` command, you can specify the SIDL symbol you're interested in and Bocca will bring up the appropriate file in your editor of choice. Additionally, after you exit the editor, Bocca regenerates all other source files that depend on the source file edited.

- To edit the `.sidl` file defining a given symbol:

```
$ bocca edit SIDL_SYMBOL
```

- To edit the header/module file of a given component class:

```
$ bocca edit -m SIDL_CLASS
```

- To edit the implementation file of the given class or component:

```
$ bocca edit -i SIDL_CLASS
```

- To edit the named method in the class or component:

```
$ bocca edit -i SIDL_CLASS METHOD_NAME
```

The last invocation requires that your editor support the +*N* option, which is used for specifying the initial position in the file. All `emacs` and `vi` versions support this feature. If your favorite editor does not support +*N*, omit the method name and search for it in the opened file using the editor's search capability.

If you replace `edit` in any of the above, with `whereis`, Bocca prints out the path of the file that would be edited without starting up an editor.

The environment variable *BOCCA_EDITOR* (and if that is not set, then *EDITOR*) controls what editor gets invoked by `bocca edit`.

Tip

If you need to set *BOCCA_EDITOR* to get the editor you want, you might want to add the appropriate setting to your login files.

Tip

Users of `emacs` may want to set *BOCCA_EDITOR* to "`emacs -nw`" when editing on a remote cluster with slow or no X11 connections.

There is also a way for you to tell Bocca that you've edited a file by some means other than `bocca edit`.

- Something was done to `FunctionPort`. Update its dependencies, if any:

```
$ bocca edit --touch FunctionPort
```

- Something was done to `Driver` code. Update its dependencies, if any:

```
$ bocca edit --touch -i Driver
```

If you do not tell Bocca about files you've modified, you might find that the project's files are not in a consistent state. For example, if you add a method to a `.sidl` file it will not appear it in the implementation file until Bocca updates it.

3.4 Adding Methods to Ports

In Section 3.2.1, we had Bocca create skeleton `.sidl` files for the `Integration` and `FunctionPort` . Now we need to flesh out the ports by actually specifying the methods they contain.

```
$ bocca edit Integration
```

The SIDL code for `Integration` looks like this:

```
// DO-NOT-DELETE bocca.splicer.begin(demo.comment)

// Insert-UserCode-Here {demo.comment} (Insert your package comments here)
// DO-NOT-DELETE bocca.splicer.end(demo.comment)
package demo version 0.0 {

    // DO-NOT-DELETE bocca.splicer.begin(demo.Integration.comment)

// Insert-UserCode-Here {demo.Integration.comment} (Insert your port comments here)
    // DO-NOT-DELETE bocca.splicer.end(demo.Integration.comment)
    interface Integration extends gov.cca.Port
    {
        // DO-NOT-DELETE bocca.splicer.begin(demo.Integration.methods)

// Insert-UserCode-Here {demo.Integration.methods} (Insert your port methods here)
        // DO-NOT-DELETE bocca.splicer.end(demo.Integration.methods)
    }
}
```

Insert the `march` method:

```
// DO-NOT-DELETE bocca.splicer.begin(demo.comment)

// Insert-UserCode-Here {demo.comment} (Insert your package comments here)
// DO-NOT-DELETE bocca.splicer.end(demo.comment)
package demo version 0.0 {

    // DO-NOT-DELETE bocca.splicer.begin(demo.Integration.comment)

// Insert-UserCode-Here {demo.Integration.comment} (Insert your port comments here)
    // DO-NOT-DELETE bocca.splicer.end(demo.Integration.comment)
    interface Integration extends gov.cca.Port
    {
        // DO-NOT-DELETE bocca.splicer.begin(demo.Integration.methods)


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


        // DO-NOT-DELETE bocca.splicer.end(demo.Integration.methods)
    }
}
```

After you quit the editor, `bocca edit` then automatically updates the components that depend on the port edited:

```
Updating makefiles (for demo.Integration, demo.Driver, demo.Integrator)...
Using Babel to validate the SIDL for port demo.Integration ...
Babel updating the cxx implementation of component demo.Driver ...
Babel updating the cxx implementation of component demo.Integrator ...
```

Next edit the file `FunctionPort.sidl`:

```
$ bocca edit FunctionPort
```

Add two methods, `init` and `evaluate` so that `function` looks like this:

```
// DO-NOT-DELETE bocca.splicer.begin(demo.comment)

// Insert-UserCode-Here {demo.comment} (Insert your package comments here)
// DO-NOT-DELETE bocca.splicer.end(demo.comment)
package demo version 0.0 {

    // DO-NOT-DELETE bocca.splicer.begin(demo.FunctionPort.comment)

// Insert-UserCode-Here {demo.FunctionPort.comment} (Insert your port comments here)
    // DO-NOT-DELETE bocca.splicer.end(demo.FunctionPort.comment)
    interface FunctionPort extends gov.cca.Port
    {
        // DO-NOT-DELETE bocca.splicer.begin(demo.FunctionPort.methods)


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


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

Again quit the editor and the dependent components are updated as indicated by this output from `bocca edit`:

```
Updating makefiles (for demo.FunctionPort, demo.Integrator, demo.Function)...
Using Babel to validate the SIDL for port demo.FunctionPort ...
Babel updating the cxx implementation of component demo.Integrator ...
Babel updating the cxx implementation of component demo.Function ...
```

These files contain the language-independent specification of the ports and their methods, expressed using SIDL. When you type `make` all of the the new method information is propagated to the language-dependent implementation files using Babel . Of course the methods will be unimplemented but the components will build anyway. So let's do that now:

```
$ make && make check
```

```
make[1]: Entering directory `/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo'
# ============================================================================
# No SIDL files in external/sidl, skipping build for external
# ============================================================================
# ============================================================================
# Building in ports/, languages: cxx
# ============================================================================
 ## Building ports...
   [c] using Babel to generate cxx client code for demo.FunctionPort...
   [c] creating library: libdemo.FunctionPort-cxx.la...
15,16c15,16
<
< // Insert-UserCode-Here {demo.FunctionPort.methods} (Insert your port methods here)
---
>         void   init(in array<double,1> params);
>         double evaluate(in double x);
   [c] installing modified demo.FunctionPort.sidl in /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/c
   [c] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/install/share/cca/demo/demo.F
   [c] using Babel to generate cxx client code for demo.Integration...
   [c] creating library: libdemo.Integration-cxx.la...
15,16c15
<
< // Insert-UserCode-Here {demo.Integration.methods} (Insert your port methods here)
---
>         double march(in double lowBound, in double upBound, in int count);
   [c] installing modified demo.Integration.sidl in /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cx
   [c] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/install/share/cca/demo/demo.I
# ============================================================================
# Building in components/clients/, languages: cxx
# ============================================================================
 ## Building clients...
# ============================================================================
# Building in components/, languages: cxx
# ============================================================================
   [s] Building class/component demo.Driver:
   [s] using Babel to generate cxx implementation code from demo.Driver.sidl...
   [s] compiling sources...
   [s] creating class/component library: libdemo.Driver.la ...
   [s] finished libtooling: components/demo.Driver/libdemo.Driver.la ...
   [s] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/install/share/cca/demo/demo.D
   [s] creating Ccaffeine test script (components/tests/instantiation.gen.rc)...
   [s] Building class/component demo.Function:
   [s] using Babel to generate cxx implementation code from demo.Function.sidl...
   [s] compiling sources...
   [s] creating class/component library: libdemo.Function.la ...
   [s] finished libtooling: components/demo.Function/libdemo.Function.la ...
   [s] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/install/share/cca/demo/demo.F
   [s] creating Ccaffeine test script (components/tests/instantiation.gen.rc)...
   [s] Building class/component demo.Integrator:
   [s] using Babel to generate cxx implementation code from demo.Integrator.sidl...
   [s] compiling sources...
   [s] creating class/component library: libdemo.Integrator.la ...
   [s] finished libtooling: components/demo.Integrator/libdemo.Integrator.la ...
   [s] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/install/share/cca/demo/demo.I
   [s] creating Ccaffeine test script (components/tests/instantiation.gen.rc)...
   [s] Building class/component demo.emptyComponent:
doing nothing -- library is up-to-date.
Build summary:
SUCCESS building demo.Driver
SUCCESS building demo.Function
SUCCESS building demo.Integrator
### To test instantiation of successfully built components, run 'make check' ###
################ Finished building everything #################
####### You can run some simple tests with 'make check' #######
make[1]: Leaving directory `/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo'
make[1]: Entering directory `/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo'
make --no-print-directory --no-builtin-rules -C components check
### Test library load and instantiation for the following languages: cxx
```

```
Running instantiation tests only
###
LDPATH=/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/install/lib:/usr/local/ACTS/cca/
###
PYTHONPATH=/usr/local/ACTS/cca/lib/python2.5/site-packages:/home/livetau/workshop-acts/cca/WORK/tutorial-src/d
###
CLASSPATH=/usr/local/ACTS/cca/lib/sidl-1.4.0.jar:/usr/local/ACTS/cca/lib/sidlstub_1.4.0.jar:/usr/local/ACTS/cc
###
Test script: /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/components/tests/instantia
Log file: /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/components/tests/instantiatio
SUCCESS:
==> Instantiation tests passed for all built components (see /home/livetau/workshop-acts/cca/WORK/tutorial-src
make --no-print-directory --no-builtin-rules check-user
make[1]: Leaving directory '/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo'
```

The methods you inserted into the SIDL port specifications have now been inserted into your already generated components. At this point we are ready to insert the actual implementation into the bodies of these methods. Notice that up to this point, you created the skeleton for an entire application without having to write any code at all, except for the SIDL for the ports. Finally, it is time to implement the components' functionality in the programming language of your choice.

3.5 Language-Specific Function, Integrator, and Driver Code

Please jump to the appropriate section for the implementation language you chose in Section 3.1 and then continue with Section 3.6 (p. 87).

Language	Section	Page
C++	3.5.1	36
F90	3.5.2	46
C	3.5.3	59
Python	3.5.4	69
Java	3.5.5	78

3.5.1 C++ Implementation

Note

Assumes you created the project with **`bocca create project demo --language=cxx`** or did not specify a language (`cxx` is the default).

Edit the `evaluate` method in the implementation file (also known as "the impl") that Bocca has generated for you (by invoking Babel). Use the `bocca edit -i` to go directly to each method.

```
$ bocca edit -i Function evaluate
```

The editor opens up in the place where the implementation code for `evaluate` must be put. You see a default implementation generated by Babel for all user methods: the throwing of an exception which says the method is not yet implemented.

```
double
demo::Function_impl::evaluate_impl (
```

```
  /* in */double x )
{
  // DO-NOT-DELETE splicer.begin(demo.Function.evaluate)
  // Insert-Code-Here {demo.Function.evaluate} (evaluate method)
  //
  // This method has not been implemented
  //
  // DO-DELETE-WHEN-IMPLEMENTING exception.begin(demo.Function.evaluate)
  ::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(demo.Function.evaluate)
  // DO-NOT-DELETE splicer.end(demo.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 method is called by mistake. Delete this exception code and substitute an implementation for the `PiFunction` (i.e., the integral from 0 to 1 of $4/(1+x^2)$ is an approximation of π).

```
  // DO-NOT-DELETE splicer.begin(demo.Function.evaluate)


    return 4.0 / (1.0 + x * x);


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

Now in the same file just above the `evaluate` method, find the second method for the `FunctionPort` `init` method:

```
  // DO-NOT-DELETE splicer.begin(demo.Function.init)


    // Do nothing.


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

We don't have any initialization in this simple example, so we just eliminate the code that throws the exception when the method is executed.

After quitting the editor the state of the source code tree is updated if there are any dependencies on the edited implementation. Usually there are no dependencies on the implementation file, so Bocca does very little after you exit the editor and all you see is the information from the edit command about what file was edited:

```
/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/components/demo.Function/demo_Function_Impl.c
```

Similarly edit the `march` method in the `Integrator` with

```
$ bocca edit -i Integrator march
```

```
double
demo::Integrator_impl::march_impl (
  /* in */double lowBound,
  /* in */double upBound,
  /* in */int32_t count )
{
  // DO-NOT-DELETE splicer.begin(demo.Integrator.march)
  // Insert-Code-Here {demo.Integrator.march} (march method)
  //
  // This method has not been implemented
  //
  // DO-DELETE-WHEN-IMPLEMENTING exception.begin(demo.Integrator.march)
  ::sidl::NotImplementedException ex = ::sidl::NotImplementedException::_create();
  ex.setNote("This method has not been implemented");
  ex.add(__FILE__, __LINE__, "march");
  throw ex;
  // DO-DELETE-WHEN-IMPLEMENTING exception.end(demo.Integrator.march)
  // DO-NOT-DELETE splicer.end(demo.Integrator.march)
}
```

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

```
  // DO-NOT-DELETE splicer.begin(demo.Integrator.march)


  demo::FunctionPort  odeRHS;
  gov::cca::Port          generalPort;

  try {
    generalPort = d_services.getPort("odeRHS");
  } catch ( ::gov::cca::CCAException ex) {
    // we cannot go on. add to the error report.
    ex.add( __FILE__, __LINE__,
           "odeRHS port not available in Integrator.march");
    throw;
  }

  odeRHS = ::babel_cast< demo::FunctionPort >(generalPort);
  if (odeRHS._is_nil()){
    // we cannot go on. toss an exception after cleaning up.
    try {
      d_services.releasePort("odeRHS");
    } catch (...) {
      // suppress framework complaints; we're already handling an exception.
    }
    ::sidl::SIDLException ex = ::sidl::SIDLException::_create();
    ex.setNote("Error: odeRHS port is nil. Weird.");
    ex.add(__FILE__, __LINE__, "demo::Integrator::integrate_impl");
    throw ex;
  }

  double h = (upBound - lowBound) / count;
  double retval = 0.0;
```

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


  // DO-NOT-DELETE splicer.end(demo.Integrator.march)
```

After quitting the editor the state of the source code tree is updated if there are any dependencies on the edited implementation. Usually there are no dependencies on the implementation file, so Bocca does very little after you exit the editor and all you see is the information from the edit command about what file was edited:

```
/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/components/demo.Integrator/demo_Integrator_Im
```

Finally for the `Driver` component we have to implement the `GoPort` details to get things going. Bocca will take you to the generated method, which looks like this:

```
$ bocca edit -i Driver go
```

```
int32_t
demo::Driver_impl::go_impl ()

{
  // DO-NOT-DELETE splicer.begin(demo.Driver.go)
// User editable portion is in the middle at the next Insert-UserCode-Here line.


// Bocca generated code. bocca.protected.begin(demo.Driver.go:boccaGoProlog)
  int bocca_status = 0;
  // The user's code should set bocca_status 0 if computation proceeded ok.
  // The user's code should set bocca_status -1 if computation failed but might
  // succeed on another call to go(), e.g. when a required port is not yet
  // connected.
  // The user's code should set bocca_status -2 if the computation failed and
  // can never succeed in a future call.
  // The user's code should NOT use return in this function.
  // Exceptions that are not caught in user code will be converted to
  // status -2.

  gov::cca::Port port;

  // nil if not fetched and cast successfully:
  demo::Integration integrate;
  // True when releasePort is needed (even if cast fails):
  bool integrate_fetched = false;
  // Use a demo.Integration port with port name integrate
  try{
```

```
    port = this->d_services.getPort("integrate");
  } catch ( ::gov::cca::CCAException ex )  {
    // we will continue with port nil (never successfully assigned) and
    // set a flag.

#ifdef _BOCCA_STDERR
    std::cerr << "demo.Driver: Error calling getPort(\"integrate\") "
              " at " << __FILE__ << ":" << __LINE__ -5 << ": " << ex.getNote()
              << std::endl;
#endif // _BOCCA_STDERR

  }
  if ( port._not_nil() ) {
    // even if the next cast fails, must release.
    integrate_fetched = true;
    integrate = ::babel_cast< demo::Integration >(port);
    if (integrate._is_nil()) {

#ifdef _BOCCA_STDERR
      std::cerr << "demo.Driver: Error casting gov::cca::Port "
                << "integrate to type "
                << "demo::Integration" << std::endl;
#endif //_BOCCA_STDERR

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


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



  // When this try/catch block is rewritten by the user, we will not change it.
  try {

    // All port instances should be rechecked for ._not_nil before calling in
    // user code. Not all ports need be connected in arbitrary use.
    // The uses ports appear as local variables here named exactly as on the
    // bocca commandline.

    // Insert-UserCode-Here {demo.Driver.go}

    // REMOVE ME BLOCK.begin(demo.Driver.go)

#ifdef _BOCCA_STDERR
    std::cerr << "USER FORGOT TO FILL IN THEIR GO FUNCTION HERE." << std::endl;
#endif

    // REMOVE ME BLOCK.end(demo.Driver.go)

  }
  // If unknown exceptions in the user code are tolerable and restart is ok,
  // return -1 instead. -2 means the component is so confused that it and
```

```
  // probably the application should be destroyed.
  // babel requires exact exception catching due to c++ binding of interfaces.
  catch (gov::cca::CCAException ex) {
    bocca_status = -2;
    std::string enote = ex.getNote();

#ifdef _BOCCA_STDERR
    std::cerr << "CCAException in user go code: " << enote << std::endl;
    std::cerr << "Returning -2 from go()" << std::endl;;
#endif

  }
  catch (sidl::RuntimeException ex) {
    bocca_status = -2;
    std::string enote = ex.getNote();

#ifdef _BOCCA_STDERR
    std::cerr << "RuntimeException in user go code: " << enote << std::endl;
    std::cerr << "Returning -2 from go()" << std::endl;;
#endif

  }
  catch (sidl::SIDLException ex) {
    bocca_status = -2;
    std::string enote = ex.getNote();

#ifdef _BOCCA_STDERR
    std::cerr << "SIDLException in user go code: " << enote << std::endl;
    std::cerr << "Returning -2 from go()" << std::endl;;
#endif

  }
  catch (sidl::BaseException ex) {
    bocca_status = -2;
    std::string enote = ex.getNote();

#ifdef _BOCCA_STDERR
    std::cerr << "BaseException in user go code: " << enote << std::endl;
    std::cerr << "Returning -2 from go()" << std::endl;;
#endif

  }
  catch (std::exception ex) {
    bocca_status = -2;

#ifdef _BOCCA_STDERR
    std::cerr << "C++ exception in user go code: " << ex.what() << std::endl;
    std::cerr << "Returning -2 from go()"  << std::endl;
#endif

  }
  catch (...) {
    bocca_status = -2;
```

```
#ifdef _BOCCA_STDERR
    std::cerr << "Odd exception in user go code " << std::endl;
    std::cerr << "Returning -2 from go()" << std::endl;
#endif

  }


  BOCCAEXIT:; // target point for error and regular cleanup. do not delete.
// Bocca generated code. bocca.protected.begin(demo.Driver.go:boccaGoEpilog)

  // release integrate
  if (integrate_fetched) {
    integrate_fetched = false;
    try{
      this->d_services.releasePort("integrate");
    } catch ( ::gov::cca::CCAException ex )  {

#ifdef _BOCCA_STDERR
      std::cerr << "demo.Driver: Error calling releasePort("
                << "\"integrate\") at "
                << __FILE__ << ":" << __LINE__ -4 << ": " << ex.getNote()
                << std::endl;
#endif // _BOCCA_STDERR

    }
    // c++ port reference will be dropped when function exits, but we
    // must tell framework.
  }


  return bocca_status;
// Bocca generated code. bocca.protected.end(demo.Driver.go:boccaGoEpilog)

  //
  // This method has not been implemented
  //
  // DO-NOT-DELETE splicer.end(demo.Driver.go)
}
```

Find the `REMOVE` block within the `go` method implementation, delete it, and insert the numerical logic needed to *use* the `integrator.IntegratorPort` port. Any required local variables should be inserted just before the `boccaGoProlog` protected block.

The `go` subroutine will be called by the framework when the component's `run` button (the name of this particular `GoPort` instance) is pushed in the GUI. Bocca generates the code to the `Integration` that the `Driver` is connected to. We just have to use it to compute the integral and return the proper value for *bocca_status*.

```
  // DO-NOT-DELETE splicer.begin(demo.Driver.go)
// User editable portion is in the middle at the next Insert-UserCode-Here line.


// Bocca generated code. bocca.protected.begin(demo.Driver.go:boccaGoProlog)
```

```
  int bocca_status = 0;
  // The user's code should set bocca_status 0 if computation proceeded ok.
  // The user's code should set bocca_status -1 if computation failed but might
  // succeed on another call to go(), e.g. when a required port is not yet
  // connected.
  // The user's code should set bocca_status -2 if the computation failed and
  // can never succeed in a future call.
  // The user's code should NOT use return in this function.
  // Exceptions that are not caught in user code will be converted to
  // status -2.

  gov::cca::Port port;

  // nil if not fetched and cast successfully:
  demo::Integration integrate;
  // True when releasePort is needed (even if cast fails):
  bool integrate_fetched = false;
  // Use a demo.Integration port with port name integrate
  try{
    port = this->d_services.getPort("integrate");
  } catch ( ::gov::cca::CCAException ex )  {
    // we will continue with port nil (never successfully assigned) and
    // set a flag.

#ifdef _BOCCA_STDERR
    std::cerr << "demo.Driver: Error calling getPort(\"integrate\") "
              " at " << __FILE__ << ":" << __LINE__ -5 << ": " << ex.getNote()
              << std::endl;
#endif // _BOCCA_STDERR

  }
  if ( port._not_nil() ) {
    // even if the next cast fails, must release.
    integrate_fetched = true;
    integrate = ::babel_cast< demo::Integration >(port);
    if (integrate._is_nil()) {

#ifdef _BOCCA_STDERR
      std::cerr << "demo.Driver: Error casting gov::cca::Port "
                << "integrate to type "
                << "demo::Integration" << std::endl;
#endif //_BOCCA_STDERR

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


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

  // When this try/catch block is rewritten by the user, we will not change it.
  try {

    // All port instances should be rechecked for ._not_nil before calling in
```

```
    // user code. Not all ports need be connected in arbitrary use.
    // The uses ports appear as local variables here named exactly as on the
    // bocca commandline.

    // Insert-UserCode-Here {demo.Driver.go}



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

    // operate on the port
    value = integrate.march(lowerBound, upperBound, count);
    std::cout << "Value = " << value << std::endl;



  }
  // If unknown exceptions in the user code are tolerable and restart is ok,
  // return -1 instead. -2 means the component is so confused that it and
  // probably the application should be destroyed.
  catch (sidl::BaseException ex) {
    bocca_status = -2;
    std::string enote = ex.getNote();

#ifdef _BOCCA_STDERR
    std::cerr << "Exception in user go code: " << enote << std::endl;
    std::cerr << "Returning -2 from go()" << std::endl;;
#endif

  }
  catch (std::exception ex) {
    bocca_status = -2;

#ifdef _BOCCA_STDERR
    std::cerr << "C++ exception in user go code: " << ex.what() << std::endl;
    std::cerr << "Returning -2 from go()"  << std::endl;
#endif

  }
  catch (...) {
    bocca_status = -2;

#ifdef _BOCCA_STDERR
    std::cerr << "Odd exception in user go code " << std::endl;
    std::cerr << "Returning -2 from go()" << std::endl;
#endif

  }


  BOCCAEXIT:; // target point for error and regular cleanup. do not delete.
// Bocca generated code. bocca.protected.begin(demo.Driver.go:boccaGoEpilog)
```

```
  // release integrate
  if (integrate_fetched) {
    integrate_fetched = false;
    try{
      this->d_services.releasePort("integrate");
    } catch ( ::gov::cca::CCAException ex )  {

#ifdef _BOCCA_STDERR
      std::cerr << "demo.Driver: Error calling releasePort("
                << "\"integrate\") at "
                << __FILE__ << ":" << __LINE__ -4 << ": " << ex.getNote()
                << std::endl;
#endif // _BOCCA_STDERR

    }
    // c++ port reference will be dropped when function exits, but we
    // must tell framework.
  }


  return bocca_status;
// Bocca generated code. bocca.protected.end(demo.Driver.go:boccaGoEpilog)

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

After quitting the editor the state of the source code tree is updated if there are any dependencies on the edited implementation. Usually there are no dependencies on the implementation file, so Bocca does very little after you exit the editor and all you see is the information from the edit command about what file was edited.

```
/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/components/demo.Driver/demo_Driver_Impl.cxx
```

Now remake your project tree to finish the components:

```
$ make
```

```
make[1]: Entering directory `/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo'
# ============================================================================
# No SIDL files in external/sidl, skipping build for external
# ============================================================================
# ============================================================================
# Building in ports/, languages: cxx
# ============================================================================
 ## Building ports...
# ============================================================================
# Building in components/clients/, languages: cxx
# ============================================================================
 ## Building clients...
# ============================================================================
# Building in components/, languages: cxx
# ============================================================================
   [s] Building class/component demo.Driver:
   [s] creating class/component library: libdemo.Driver.la ...
   [s] finished libtooling: components/demo.Driver/libdemo.Driver.la ...
   [s] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/install/share/cca/demo/demo.D
   [s] creating Ccaffeine test script (components/tests/instantiation.gen.rc)...
   [s] Building class/component demo.Function:
```

```
   [s] creating class/component library: libdemo.Function.la ...
   [s] finished libtooling: components/demo.Function/libdemo.Function.la ...
   [s] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/install/share/cca/demo/
   [s] creating Ccaffeine test script (components/tests/instantiation.gen.rc)...
   [s] Building class/component demo.Integrator:
   [s] creating class/component library: libdemo.Integrator.la ...
   [s] finished libtooling: components/demo.Integrator/libdemo.Integrator.la ...
   [s] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/install/share/cca/demo/
   [s] creating Ccaffeine test script (components/tests/instantiation.gen.rc)...
   [s] Building class/component demo.emptyComponent:
doing nothing -- library is up-to-date.
Build summary:
SUCCESS building demo.Driver
SUCCESS building demo.Function
SUCCESS building demo.Integrator
### To test instantiation of successfully built components, run 'make check' ###
################ Finished building everything #################
####### You can run some simple tests with 'make check' #######
make[1]: Leaving directory '/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo'
```

It is good practice to do a `make check` at this point:

```
$ make check
```

```
make[1]: Entering directory '/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo'
make --no-print-directory --no-builtin-rules -C components check
### Test library load and instantiation for the following languages: cxx
Running instantiation tests only
###
LDPATH=/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/install/lib:/usr/local/ACTS/cca/
###
PYTHONPATH=/usr/local/ACTS/cca/lib/python2.5/site-packages:/home/livetau/workshop-acts/cca/WORK/tutorial-src/d
###
CLASSPATH=/usr/local/ACTS/cca/lib/sidl-1.4.0.jar:/usr/local/ACTS/cca/lib/sidlstub_1.4.0.jar:/usr/local/ACTS/cc
###
Test script: /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/components/tests/instantia
Log file: /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/components/tests/instantiatio
SUCCESS:
==> Instantiation tests passed for all built components (see /home/livetau/workshop-acts/cca/WORK/tutorial-src
make --no-print-directory --no-builtin-rules check-user
make[1]: Leaving directory '/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo'
```

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

3.5.2 Fortran9X Implementation

Note

Assumes you created the project with **`bocca create project demo --language=f90`**.

Edit the `evaluate` method in the implementation file (also known as "the impl") that Bocca has generated for you (by invoking Babel). Use the `bocca edit -i` to go directly to each method.

```
$ bocca edit -i Function evaluate
```

The editor opens up in the place where the implementation code for `evaluate` must be put. You see a default implementation generated by Babel for all user methods: the throwing of an exception which says the method is not yet implemented.

```
recursive subroutine demo_Function_evaluate_mi(self, x, retval, exception)
  use sidl
  use sidl_NotImplementedException
  use sidl_BaseInterface
  use sidl_RuntimeException
  use demo_Function
  use demo_Function_impl
  ! DO-NOT-DELETE splicer.begin(demo.Function.evaluate.use)
  ! Insert-Code-Here {demo.Function.evaluate.use} (use statements)
  ! DO-NOT-DELETE splicer.end(demo.Function.evaluate.use)
  implicit none
  type(demo_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(demo.Function.evaluate)
! Insert-Code-Here {demo.Function.evaluate} (evaluate method)
!
! This method has not been implemented
!

  ! DO-DELETE-WHEN-IMPLEMENTING exception.begin(demo.Function.evaluate)
  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(demo.Function.evaluate)
! DO-NOT-DELETE splicer.end(demo.Function.evaluate)
end subroutine demo_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 method is called by mistake. Delete this exception code and substitute an implementation for the `PiFunction` (i.e., the integral from 0 to 1 of $4/(1+x^2)$ is an approximation of π).

```
! DO-NOT-DELETE splicer.begin(demo.Function.evaluate)


  retval = 4.0 / (1.0 + x * x)
```

```
! DO-NOT-DELETE splicer.end(demo.Function.evaluate)
```

Now in the same file just above the `evaluate` method, find the second method for the `FunctionPort` `init` method:

```
! DO-NOT-DELETE splicer.begin(demo.Function.init)


  ! Do nothing


! DO-NOT-DELETE splicer.end(demo.Function.init)
```

We don't have any initialization in this simple example, so we just eliminate the code that throws the exception when the method is executed.

After quitting the editor the state of the source code tree is updated if there are any dependencies on the edited implementation. Usually there are no dependencies on the implementation file, so Bocca does very little after you exit the editor and all you see is the information from the edit command about what file was edited:

```
/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/f90/demo/components/demo.Function/demo_Function_
```

Similarly edit the `march` method in the `Integrator` with

```
$ bocca edit -i Integrator march
```

```
recursive subroutine demo_Integrator_march_mi(self, lowBound, upBound, count,  &
  retval, exception)
  use sidl
  use sidl_NotImplementedException
  use sidl_BaseInterface
  use sidl_RuntimeException
  use demo_Integrator
  use demo_Integrator_impl
  ! DO-NOT-DELETE splicer.begin(demo.Integrator.march.use)
  ! Insert-Code-Here {demo.Integrator.march.use} (use statements)
  ! DO-NOT-DELETE splicer.end(demo.Integrator.march.use)
  implicit none
  type(demo_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(demo.Integrator.march)
! Insert-Code-Here {demo.Integrator.march} (march method)
!
! This method has not been implemented
!

  ! DO-DELETE-WHEN-IMPLEMENTING exception.begin(demo.Integrator.march)
  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(demo.Integrator.march)
! DO-NOT-DELETE splicer.end(demo.Integrator.march)
end subroutine demo_Integrator_march_mi
```

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

```
! DO-NOT-DELETE splicer.begin(demo.Integrator.march.use)


  ! the port types we need go here.
  use gov_cca_Port
  use demo_FunctionPort


! DO-NOT-DELETE splicer.end(demo.Integrator.march.use)

! DO-NOT-DELETE splicer.begin(demo.Integrator.march)


  ! User's local declarations. We follow the pattern generated for us in Driver.go()
  type(gov_cca_Port_t) :: port
  type(gov_cca_Services_t) :: services
  type(SIDL_BaseInterface_t) :: throwaway
  type(SIDL_BaseInterface_t) :: dumex
  type(demo_Integrator_wrap) :: dp
  logical dr_port ! if dr_X true, the deleteRef(X) is needed before return.

  type(demo_FunctionPort_t) :: odeRHS__p
  ! odeRHS__p is non-null if specific uses port obtained.

  logical odeRHS_fetched
  ! odeRHS_fetched true if releaseport is needed for this port.

! a small message catalog for exception reporting
  character (LEN=*) errMsg00
  character (LEN=*) errMsg0
  character (LEN=*) errMsg1
```

```
  character (LEN=*) errMsg2
  character (LEN=*) errMsg3
  character (LEN=*) errMsg4
  parameter(errMsg00= &
     'NULL d_services pointer in demo.Integrator.march()')
  parameter(errMsg0= &
    'demo.Integrator: Error go() getPort(odeRHS) failed.')
  parameter(errMsg1= &
    'demo.Integrator: Error casting odeRHS to FunctionPort')
  parameter(errMsg2= &
     'demo.Integrator: Error in deleteRef(port) while getting odeRHS')
  parameter(errMsg3= &
     'demo.Integrator: Error calling releasePort(odeRHS).')
  parameter(errMsg4= &
     'demo.Integrator: Error in deleteRef for port odeRHS.')

! numerical method variable, other than the call arguments:
  real (kind=sidl_double) :: h, fvalueleft, fvalueright, sum, left, right
  integer i

  BOCCA_EXTERNAL
  ! not crashing if something fails .eq. good bookkeeping and exception handling.
  ! start with initialization
  call set_null( odeRHS__p)
  odeRHS_fetched = .false.
  call set_null(services)
  call set_null(port)
  call set_null(throwaway)
  call set_null(dumex)
  dr_port = .false.
  call demo_Integrator__get_data_m(self,dp);
  services =  dp%d_private_data%d_services
  retval = -4.0

  if (is_null(services) ) then
    call BOCCA_SIDL_THROW_F90(exception, errMsg00)
  endif

  ! Use a demo.FunctionPort port with port name odeRHS
  call getPort(services,"odeRHS", port, exception)
  BOCCA_SIDL_CHECK_F90(exception, errMsg0)

  odeRHS_fetched = .true.
  ! even if the next cast fails, must releasePort per odeRHS_fetched.
  call cast(port, odeRHS__p, exception)
  BOCCA_SIDL_CHECK_F90(exception, errMsg1)

  ! done with the generic port pointer. drop it.
  call deleteRef(port, exception)
  call set_null(port)
  BOCCA_SIDL_CHECK_F90(exception, errMsg2)

  !! here's the numerical work
```

```
  ! the trapezoidal rule
  h = (upBound - lowBound) / count
  sum = 0.0
  fvalueleft = 0.0
  fvalueright = 0.0
  do i =  1,count
    left = lowBound + (i - 1) * h
    call evaluate(odeRHS__p, left, fvalueleft, exception)
    BOCCA_SIDL_CHECK_F90(exception, 'error calculating fvalueleft')

    right = lowBound + i * h
    call evaluate(odeRHS__p, right, fvalueright, exception)
    BOCCA_SIDL_CHECK_F90(exception, 'error calculating fvalueright')

    sum = sum + fvalueleft + fvalueright
  enddo
  retval = h/2.0 * sum;

  !! the numerical work is done.

BOCCAEXIT continue ! target point for normal and error cleanup.

  if (not_null(port)) then
    call deleteRef(port,throwaway)
    call checkException(self, throwaway, 'cleanup port error', .false., dumex)
    call set_null(port)
  endif

  ! release odeRHS
  if (odeRHS_fetched) then
    odeRHS_fetched = .false.
    call releasePort(services, 'odeRHS', throwaway)
    call checkException(self, throwaway, errMsg3, .false., dumex)

    if ( not_null(odeRHS__p) ) then
      call deleteRef(odeRHS__p, throwaway)
      call checkException(self, throwaway, errMsg4, .false., dumex)
      call set_null(odeRHS__p)
    endif

  endif


! DO-NOT-DELETE splicer.end(demo.Integrator.march)
```

After quitting the editor the state of the source code tree is updated if there are any dependencies on the edited implementation. Usually there are no dependencies on the implementation file, so Bocca does very little after you exit the editor and all you see is the information from the edit command about what file was edited:

```
/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/f90/demo/components/demo.Integrator/demo_Integrator_Im
```

Finally for the `Driver` component we have to implement the `GoPort` details to get things going. Bocca will take you to the generated method, which looks like this:

```
$ bocca edit -i Driver go
```

```
recursive subroutine demo_Driver_go_mi(self, retval, exception)
  use sidl
  use sidl_NotImplementedException
  use sidl_BaseInterface
  use sidl_RuntimeException
  use demo_Driver
  use demo_Driver_impl
  ! DO-NOT-DELETE splicer.begin(demo.Driver.go.use)

! Bocca generated code. bocca.protected.begin(demo.Driver.go.use)
  use gov_cca_Port
  use demo_Integration

! Bocca generated code. bocca.protected.end(demo.Driver.go.use)

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



! DO-NOT-DELETE splicer.begin(demo.Driver.go)



! Insert-User-Declarations-Here

! Bocca generated code. bocca.protected.begin(demo.Driver.go:boccaGoProlog)

  integer bocca_status
!  The user's code should set bocca_status 0 if computation proceeded ok.
!  The user's code should set bocca_status -1 if computation failed but might
!  succeed on another call to go(), e.g. wheh a required port is not yet connected.
!  The user's code should set bocca_status -2 if the computation failed and can
!  never succeed in a future call.
!  The users's code should NOT use return in this function;
!  Exceptions that are not caught in user code will be converted to status -2.
!


  type(gov_cca_Port_t) :: port
  type(gov_cca_Services_t) :: services
  type(SIDL_BaseInterface_t) :: throwaway
  type(SIDL_BaseInterface_t) :: dumex
  type(demo_Driver_wrap) :: dp
  logical dr_port ! if dr_X true, the deleteRef(X) is needed before return.
```

```
type(demo_Integration_t) ::  integrate__p ! non-null if specific uses port obtained.
logical integrate_fetched            ! true if releaseport is needed for this port.
character (LEN=*) errMsg0_integrate
character (LEN=*) errMsg1_integrate
character (LEN=*) errMsg2_integrate
character (LEN=*) errMsg3_integrate
character (LEN=*) errMsg4_integrate
parameter(errMsg0_integrate= &
  'demo.Driver: Error go() getPort(integrate) failed.')
parameter(errMsg1_integrate= &
  'demo.Driver: Error casting gov.cca.Port integrate to type demo.Integration')
parameter(errMsg2_integrate= &
   'demo.Driver: Error in deleteRef(port) while getting integrate')
parameter(errMsg3_integrate= &
   'demo.Driver: Error calling releasePort(integrate). Continuing.')
parameter(errMsg4_integrate = &
   'demo.Driver: Error in deleteRef for port integrate. Continuing.')


BOCCA_EXTERNAL
! not crashing if something fails requires good bookkeeping and exception handling.
call set_null(services)
call set_null(port)
call set_null(throwaway)
call set_null(dumex)
dr_port = .false.
bocca_status = 0
call demo_Driver__get_data_m(self,dp);
services =  dp%d_private_data%d_services

if (is_null(services) ) then
  call BOCCA_SIDL_THROW_F90(exception, 'NULL d_services pointer in demo.Driver.go()')
endif

! Use a demo.Integration port with port name integrate
call getPort(services,"integrate", port, throwaway)
if ( not_null(throwaway) ) then
  call set_null(port)
  call checkException(self, throwaway, errMsg0_integrate, .false., dumex)
  ! we will continue with port null (never successfully assigned) and set a flag.
endif

call set_null( integrate__p)
integrate_fetched = .false.
if ( not_null(port)) then
  integrate_fetched = .true. ! even if the next cast fails, must releasePort.
  call cast(port, integrate__p, exception)
  BOCCA_SIDL_CHECK_F90(exception, errMsg1_integrate)
  call deleteRef(port, exception)
  call set_null(port)
  BOCCA_SIDL_CHECK_F90(exception, errMsg2_integrate)
endif
```

```
! Bocca generated code. bocca.protected.end(demo.Driver.go:boccaGoProlog)



! When this block is rewritten by the user, we will not change it.
! All port instances should be rechecked for NULL before calling in user code.
! Not all ports need be connected in arbitrary use.
! The port instance names used in registerUsesPort appear as local variable
! names here with the suffix __p added.


! BEGIN REMOVE ME BLOCK
#ifdef _BOCCA_STDERR
  write(*,*) 'USER FORGOT TO FILL IN THEIR FUNCTION demo.Driver.go.'
#endif
! END REMOVE ME BLOCK


!    If unknown exceptions in the user code are tolerable and restart is ok,
!    set bocca_status -1 instead.
!    -2 means the component is so confused that it and probably the application
!    should be destroyed.
!


BOCCAEXIT continue ! target point for normal and error cleanup. do not delete.
! Bocca generated code. bocca.protected.begin(demo.Driver.go:boccaGoEpilog)

  if (not_null(port)) then
    call deleteRef(port,throwaway)
    call checkException(self, throwaway, 'cleanup port error', .false., dumex)
    call set_null(port)
  endif

  ! release integrate
  if (integrate_fetched) then
    integrate_fetched = .false.
    call releasePort(services, 'integrate', throwaway)
    call checkException(self, throwaway, errMsg3_integrate, .false., dumex)

    if ( not_null(integrate__p) ) then
      call deleteRef(integrate__p, throwaway)
      call checkException(self, throwaway, errMsg4_integrate, .false., dumex)
      call set_null(integrate__p)
    endif

  endif


! Bocca generated code. bocca.protected.end(demo.Driver.go:boccaGoEpilog)
```

```
! Insert-User-Exception-Cleanup-Here

  retval = bocca_status
!
! This method has not been implemented
!

! DO-NOT-DELETE splicer.end(demo.Driver.go)
end subroutine demo_Driver_go_mi
```

Find the `REMOVE` block within the `go` method implementation, delete it, and insert the numerical logic needed to *use* the `integrator.IntegratorPort` port. Any required local variables should be inserted just before the `boccaGoProlog` protected block.

The `go` subroutine will be called by the framework when the component's `run` button (the name of this particular `GoPort` instance) is pushed in the GUI. Bocca generates the code to the `Integration` that the `Driver` is connected to. We just have to use it to compute the integral and return the proper value for *`bocca_status`*.

```
! DO-NOT-DELETE splicer.begin(demo.Driver.go)
! Insert-User-Declarations-Here


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


! Bocca generated code. bocca.protected.begin(demo.Driver.go:boccaGoProlog)

  integer bocca_status
!  The user's code should set bocca_status 0 if computation proceeded ok.
!  The user's code should set bocca_status -1 if computation failed but might
!  succeed on another call to go(), e.g. wheh a required port is not yet connected.
!  The user's code should set bocca_status -2 if the computation failed and can
!  never succeed in a future call.
!  The users's code should NOT use return in this function;
!  Exceptions that are not caught in user code will be converted to status -2.
!


  type(gov_cca_Port_t) :: port
  type(gov_cca_Services_t) :: services
  type(SIDL_BaseInterface_t) :: throwaway
  type(SIDL_BaseInterface_t) :: dumex
  type(demo_Driver_wrap) :: dp
  logical dr_port ! if dr_X true, the deleteRef(X) is needed before return.

  type(demo_Integration_t) ::  integrate__p ! non-null if specific uses port obtained.
  logical integrate_fetched            ! true if releaseport is needed for this port.
  character (LEN=*) errMsg0_integrate
```

```
  character (LEN=*) errMsg1_integrate
  character (LEN=*) errMsg2_integrate
  character (LEN=*) errMsg3_integrate
  character (LEN=*) errMsg4_integrate
  parameter(errMsg0_integrate= &
    'demo.Driver: Error go() getPort(integrate) failed.')
  parameter(errMsg1_integrate= &
    'demo.Driver: Error casting gov.cca.Port integrate to type demo.Integration')
  parameter(errMsg2_integrate= &
     'demo.Driver: Error in deleteRef(port) while getting integrate')
  parameter(errMsg3_integrate= &
     'demo.Driver: Error calling releasePort(integrate). Continuing.')
  parameter(errMsg4_integrate = &
     'demo.Driver: Error in deleteRef for port integrate. Continuing.')


  BOCCA_EXTERNAL
  ! not crashing if something fails requires good bookkeeping and exception handling.
  call set_null(services)
  call set_null(port)
  call set_null(throwaway)
  call set_null(dumex)
  dr_port = .false.
  bocca_status = 0
  call demo_Driver__get_data_m(self,dp);
  services =  dp%d_private_data%d_services

  if (is_null(services) ) then
    call BOCCA_SIDL_THROW_F90(exception, 'NULL d_services pointer in demo.Driver.go()')
  endif

  ! Use a demo.Integration port with port name integrate
  call getPort(services,"integrate", port, throwaway)
  if ( not_null(throwaway) ) then
    call set_null(port)
    call checkException(self, throwaway, errMsg0_integrate, .false., dumex)
    ! we will continue with port null (never successfully assigned) and set a flag.
  endif

  call set_null( integrate__p)
  integrate_fetched = .false.
  if ( not_null(port)) then
    integrate_fetched = .true. ! even if the next cast fails, must releasePort.
    call cast(port, integrate__p, exception)
    BOCCA_SIDL_CHECK_F90(exception, errMsg1_integrate)
    call deleteRef(port, exception)
    call set_null(port)
    BOCCA_SIDL_CHECK_F90(exception, errMsg2_integrate)
  endif


! Bocca generated code. bocca.protected.end(demo.Driver.go:boccaGoProlog)
```

```
! When this block is rewritten by the user, we will not change it.
! All port instances should be rechecked for NULL before calling in user code.
! Not all ports need be connected in arbitrary use.
! The port instance names used in registerUsesPort appear as local variable
! names here with the suffix __p added.



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

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

     ! operate on the port. if successful, set the status to 0 for ok.
     bocca_status = -2
     call march(integrate__p, lowBound, upBound, count, value, exception)
     ! jump to BOCCAEXIT if an error.
     BOCCA_SIDL_CHECK_F90(exception,'Driver:go: problem calling integrate')
     write(*,*) 'Value = ', value
     bocca_status = 0
  else
     bocca_status = -1 ; ! integratorPort is not connected.
     write(*,*) 'Driver: integrate port not connected. connect and try again'
  endif



!    If unknown exceptions in the user code are tolerable and restart is ok,
!    set bocca_status -1 instead.
!    -2 means the component is so confused that it and probably the application
!    should be destroyed.
!


BOCCAEXIT continue ! target point for normal and error cleanup. do not delete.
! Bocca generated code. bocca.protected.begin(demo.Driver.go:boccaGoEpilog)

  if (not_null(port)) then
    call deleteRef(port,throwaway)
    call checkException(self, throwaway, 'cleanup port error', .false., dumex)
    call set_null(port)
  endif

  ! release integrate
  if (integrate_fetched) then
    integrate_fetched = .false.
    call releasePort(services, 'integrate', throwaway)
```

```
    call checkException(self, throwaway, errMsg3_integrate, .false., dumex)

    if ( not_null(integrate__p) ) then
      call deleteRef(integrate__p, throwaway)
      call checkException(self, throwaway, errMsg4_integrate, .false., dumex)
      call set_null(integrate__p)
    endif

  endif


! Bocca generated code. bocca.protected.end(demo.Driver.go:boccaGoEpilog)


! Insert-User-Exception-Cleanup-Here

  retval = bocca_status


! DO-NOT-DELETE splicer.end(demo.Driver.go)
```

After quitting the editor the state of the source code tree is updated if there are any dependencies on the edited implementation. Usually there are no dependencies on the implementation file, so Bocca does very little after you exit the editor and all you see is the information from the edit command about what file was edited.

```
/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/f90/demo/components/demo.Driver/demo_Driver_Impl
```

Now remake your project tree to finish the components:

```
$ make
```

```
make[1]: Entering directory '/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/f90/demo'
# ============================================================================
# No SIDL files in external/sidl, skipping build for external
# ============================================================================
# ============================================================================
# Building in ports/, languages: f90
# ============================================================================
 ## Building ports...
# ============================================================================
# Building in components/clients/, languages: f90
# ============================================================================
 ## Building clients...
# ============================================================================
# Building in components/, languages: f90
# ============================================================================
   [s] Building class/component demo.Driver:
   [s] creating class/component library: libdemo.Driver.la ...
   [s] finished libtooling: components/demo.Driver/libdemo.Driver.la ...
   [s] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/f90/demo/install/share/cca/demo/
   [s] creating Ccaffeine test script (components/tests/instantiation.gen.rc)...
   [s] Building class/component demo.Function:
   [s] creating class/component library: libdemo.Function.la ...
   [s] finished libtooling: components/demo.Function/libdemo.Function.la ...
   [s] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/f90/demo/install/share/cca/demo/
   [s] creating Ccaffeine test script (components/tests/instantiation.gen.rc)...
   [s] Building class/component demo.Integrator:
   [s] creating class/component library: libdemo.Integrator.la ...
```

```
   [s] finished libtooling: components/demo.Integrator/libdemo.Integrator.la ...
   [s] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/f90/demo/install/share/cca/demo/demo.I
   [s] creating Ccaffeine test script (components/tests/instantiation.gen.rc)...
   [s] Building class/component demo.emptyComponent:
doing nothing -- library is up-to-date.
Build summary:
SUCCESS building demo.Driver
SUCCESS building demo.Function
SUCCESS building demo.Integrator
### To test instantiation of successfully built components, run 'make check' ###
################ Finished building everything #################
####### You can run some simple tests with 'make check' #######
make[1]: Leaving directory '/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/f90/demo'
```

It is good practice to do a `make check` at this point:

```
$ make check
```

```
make[1]: Entering directory '/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/f90/demo'
make --no-print-directory --no-builtin-rules -C components check
### Test library load and instantiation for the following languages: f90
Running instantiation tests only
###
LDPATH=/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/f90/demo/install/lib:/usr/local/ACTS/cca/lib:/u
###
PYTHONPATH=/usr/local/ACTS/cca/lib/python2.5/site-packages:/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scr
###
CLASSPATH=/usr/local/ACTS/cca/lib/sidl-1.4.0.jar:/usr/local/ACTS/cca/lib/sidlstub_1.4.0.jar:/usr/local/ACTS/cca/lib/
###
Test script: /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/f90/demo/components/tests/instantiation.g
Log file: /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/f90/demo/components/tests/instantiation.gen.
SUCCESS:
==> Instantiation tests passed for all built components (see /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/s
make --no-print-directory --no-builtin-rules check-user
make[1]: Leaving directory '/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/f90/demo'
```

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

3.5.3 C Implementation

Note

Assumes you created the project with **`bocca create project demo --language=c`**.

Edit the `evaluate` method in the implementation file (also known as "the impl") that Bocca has generated for you (by invoking Babel). Use the `bocca edit -i` to go directly to each method.

```
$ bocca edit -i Function evaluate
```

The editor opens up in the place where the implementation code for `evaluate` must be put. You see a default implementation generated by Babel for all user methods: the throwing of an exception which says the method is not yet implemented.

```
double
impl_demo_Function_evaluate(
  /* in */ demo_Function self,
  /* in */ double x,
  /* out */ sidl_BaseInterface *_ex)
{
  *_ex = 0;
  {
    /* DO-NOT-DELETE splicer.begin(demo.Function.evaluate) */
    /* Insert-Code-Here {demo.Function.evaluate} (evaluate method) */
    /*
     * This method has not been implemented
     */

    /* DO-DELETE-WHEN-IMPLEMENTING exception.begin(demo.Function.evaluate) */
    SIDL_THROW(*_ex, sidl_NotImplementedException,     "This method has not been impleme
  EXIT:;
    /* DO-DELETE-WHEN-IMPLEMENTING exception.end(demo.Function.evaluate) */
    /* DO-NOT-DELETE splicer.end(demo.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 method is called by mistake. Delete this exception code and substitute an implementation for the `PiFunction` (i.e., the integral from 0 to 1 of $4/(1+x^2)$ is an approximation of π).

```
  /* DO-NOT-DELETE splicer.begin(demo.Function.evaluate) */


  return 4.0 / (1.0 + x * x);


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

Now in the same file just above the `evaluate` method, find the second method for the `FunctionPort` `init` method:

```
  /* DO-NOT-DELETE splicer.begin(demo.Function.init) */


  /* Do nothing.*/


  /* DO-NOT-DELETE splicer.end(demo.Function.init) */
```

We don't have any initialization in this simple example, so we just eliminate the code that throws the exception when the method is executed.

After quitting the editor the state of the source code tree is updated if there are any dependencies on the edited implementation. Usually there are no dependencies on the implementation file, so Bocca does very little after you exit the editor and all you see is the information from the edit command about what file was edited:

/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/c/demo/components/demo.Function/demo_Function_Impl.c

Similarly edit the `march` method in the `Integrator` with

```
$ bocca edit -i Integrator march
```

```
double
impl_demo_Integrator_march(
  /* in */ demo_Integrator self,
  /* in */ double lowBound,
  /* in */ double upBound,
  /* in */ int32_t count,
  /* out */ sidl_BaseInterface *_ex)
{
  *_ex = 0;
  {
    /* DO-NOT-DELETE splicer.begin(demo.Integrator.march) */
    /* Insert-Code-Here {demo.Integrator.march} (march method) */
    /*
     * This method has not been implemented
     */

    /* DO-DELETE-WHEN-IMPLEMENTING exception.begin(demo.Integrator.march) */
    SIDL_THROW(*_ex, sidl_NotImplementedException,     "This method has not been implemented"
  EXIT:;
    /* DO-DELETE-WHEN-IMPLEMENTING exception.end(demo.Integrator.march) */
    /* DO-NOT-DELETE splicer.end(demo.Integrator.march) */
  }
}
```

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

```
  /* DO-NOT-DELETE splicer.begin(demo.Integrator.march) */


  gov_cca_Port port = NULL;
  gov_cca_Services services = NULL;
  sidl_BaseInterface throwaway_excpt = NULL;
  sidl_BaseInterface dummy_excpt = NULL;
  struct demo_Integrator__data *pd = NULL;
  const char *errMsg = NULL;
  double retval = 0.0;

  demo_FunctionPort odeRHS = NULL;
  /* odeRHS non-null if specific uses port obtained. */

  int odeRHS_fetched = FALSE;
  /* odeRHS_fetched true if releaseport is needed for this port. */

  pd = demo_Integrator__get_data(self);
  if (pd == NULL) {
```

```
    SIDL_THROW(*_ex, sidl_SIDLException,
     "NULL object data pointer in demo.Integrator.march()");
  }
  services = pd->d_services;
  if (services == NULL) {
    SIDL_THROW(*_ex, sidl_SIDLException,
      "NULL pd->d_services pointer in demo.Integrator.march()");
  }

  /* Use a demo.Integration port with port name odeRHS */
  port =
    gov_cca_Services_getPort(services,"odeRHS", _ex); SIDL_CHECK(*_ex);
  odeRHS_fetched = TRUE;
  /* even if the next cast fails, must releasePort. */

  errMsg="demo.Integrator: Error casting odeRHS to FunctionPort";
  odeRHS = gov_cca_Services__cast2(port,
                                          "demo.FunctionPort",
                                           _ex); SIDL_CHECK(*_ex);
  gov_cca_Port_deleteRef(port, _ex); port = NULL; SIDL_CHECK(*_ex);

  {
    double h;
    double sum = 0.0;
    double left, right, fvalueleft, fvalueright;
    int i;

    h = (upBound - lowBound) / (1.0*count);
    printf("Evaluating from %g to %g by %d\n",lowBound ,upBound, count);
    for (i = 1; i <= count; i++){

      left = lowBound + (i - 1) * h;
      fvalueleft = demo_FunctionPort_evaluate(odeRHS,
                     left,_ex); SIDL_CHECK(*_ex);

      right = lowBound + i * h;
      fvalueright = demo_FunctionPort_evaluate(odeRHS,
                      right,_ex); SIDL_CHECK(*_ex);

      sum += (fvalueleft + fvalueright);
    }
    retval = h * sum/2.0;
    printf("IP returning %g\n",retval);
  }
EXIT:; /* target point for normal and error cleanup. do not delete. */

  /* release integrate */
  if (odeRHS_fetched) {
    odeRHS_fetched = FALSE;
    gov_cca_Services_releasePort(services,"odeRHS",&throwaway_excpt);
    if ( throwaway_excpt != NULL) {
      errMsg= "demo.Integrator: Error calling"
              " releasePort(\"integrate\"). Continuing.";
      demo_Integrator_checkException(self, throwaway_excpt, errMsg,
```

```
                                          FALSE, &dummy_excpt);
    }
    if (odeRHS != NULL) {
      demo_FunctionPort_deleteRef(odeRHS, &throwaway_excpt);
      errMsg =  "Error in demo_FunctionPort_deleteRef"
                " for demo.Function port odeRHS";
      demo_Integrator_checkException(self, throwaway_excpt, errMsg,
                                          FALSE, &dummy_excpt);
      odeRHS = NULL;
    }
  }

  return retval;


  /* DO-NOT-DELETE splicer.end(demo.Integrator.march) */
```

After quitting the editor the state of the source code tree is updated if there are any dependencies on the edited implementation. Usually there are no dependencies on the implementation file, so Bocca does very little after you exit the editor and all you see is the information from the edit command about what file was edited:

```
/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/c/demo/components/demo.Integrator/demo_Integrator_Impl
```

Finally for the `Driver` component we have to implement the `GoPort` details to get things going. Bocca will take you to the generated method, which looks like this:

```
$ bocca edit -i Driver go
```

```
int32_t
impl_demo_Driver_go(
  /* in */ demo_Driver self,
  /* out */ sidl_BaseInterface *_ex)
{
  *_ex = 0;
  {
    /* DO-NOT-DELETE splicer.begin(demo.Driver.go) */

  /* User action portion is in the middle at the next Insert-UserCode-Here line. */


  /* Insert-User-Declarations-Here */

/* Bocca generated code. bocca.protected.begin(demo.Driver.go:boccaGoProlog) */

  int bocca_status = 0;
  /* The user's code should set bocca_status 0 if computation proceeded ok.
  // The user's code should set bocca_status -1 if computation failed but might
  // succeed on another call to go(), e.g. wheh a required port is not yet connected.
  // The user's code should set bocca_status -2 if the computation failed and can
  // never succeed in a future call.
  // The users's code should NOT use return in this function;
```

```
  // Exceptions that are not caught in user code will be converted to status -2.
  */


  gov_cca_Port port = NULL;
  gov_cca_Services services = NULL;
  sidl_BaseInterface throwaway_excpt = NULL;
  sidl_BaseInterface dummy_excpt = NULL;
  struct demo_Driver__data *pd = NULL;
  const char *errMsg = NULL;

  demo_Integration integrate = NULL; /* non-null if specific uses port obtained. */
  int integrate_fetched = FALSE; /* true if releaseport is needed for this port. */


  pd = demo_Driver__get_data(self);
  if (pd == NULL) {
    SIDL_THROW(*_ex, sidl_SIDLException,
       "NULL object data pointer in demo.Driver.go()");
  }
  services = pd->d_services;
  if (services == NULL) {
    SIDL_THROW(*_ex, sidl_SIDLException,
        "NULL pd->d_services pointer in demo.Driver.go()");
  }

  /* Use a demo.Integration port with port name integrate */
  port = gov_cca_Services_getPort(services,"integrate", &throwaway_excpt);
  if (throwaway_excpt != NULL) {
    port = NULL;
    errMsg="go() getPort(integrate) failed.";
    demo_Driver_checkException(self, throwaway_excpt, errMsg,
                                    FALSE, &dummy_excpt);
    /* we will continue with port NULL (never successfully assigned) and set a flag. */
    BOCCA_FPRINTF(stderr,
         "demo.Driver: Error calling getPort(\"integrate\") at %s:%d. Continuing.\n",
         __FILE__ , __LINE__ -8 );
  }

  if ( port != NULL ) {
    integrate_fetched = TRUE; /* even if the next cast fails, must releasePort. */
    errMsg="demo.Driver: Error casting gov.cca.Port integrate to type demo.Integration";
    integrate = demo_Integration__cast(port, _ex); SIDL_CHECK(*_ex);
    gov_cca_Port_deleteRef(port,_ex); port = NULL; SIDL_CHECK(*_ex);
  }


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




  /* When this block is rewritten by the user, we will not change it.
     All port instances should be rechecked for NULL before calling in user code.
```

```
     Not all ports need be connected in arbitrary use.
     The port instance names used in registerUsesPort appear as local variable
     names here.
     'return' should not be used here; set bocca_status instead.
   */

  /* Insert-UserCode-Here {demo.Driver.go} */

  /* BEGIN REMOVE ME BLOCK */
  BOCCA_FPRINTF(stderr,
        "USER FORGOT TO FILL IN THEIR GO FUNCTION %s:%d.\n",
        __FILE__,__LINE__);
  /* END REMOVE ME BLOCK */

  /* If unknown exceptions in the user code are tolerable and restart is ok,
     set bocca_status -1 instead.
     -2 means the component is so confused that it and probably the component
     or application should be destroyed.
   */


EXIT:; /* target point for normal and error cleanup. do not delete. */
/* Bocca generated code. bocca.protected.begin(demo.Driver.go:boccaGoEpilog) */

  /* release integrate */
  if (integrate_fetched) {
    integrate_fetched = FALSE;
    gov_cca_Services_releasePort(services,"integrate",&throwaway_excpt);
    if ( throwaway_excpt != NULL) {
      errMsg= "demo.Driver: Error calling releasePort(\"integrate\"). Continuing.";
      demo_Driver_checkException(self, throwaway_excpt, errMsg, FALSE, &dummy_excpt);
    }
    if (integrate != NULL) {
      demo_Integration_deleteRef(integrate, &throwaway_excpt);
      errMsg = "Error in demo_Integration_deleteRef for demo.Driver port integrate";
      demo_Driver_checkException(self, throwaway_excpt, errMsg, FALSE, &dummy_excpt);
      integrate = NULL;
    }
  }


/* Bocca generated code. bocca.protected.end(demo.Driver.go:boccaGoEpilog) */


/* Insert-User-Exception-Cleanup-Here */

  return bocca_status;
    /*
     * This method has not been implemented
     */

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

Find the REMOVE block within the go method implementation, delete it, and insert the numerical logic needed to *use* the integrator.IntegratorPort port. Any required local variables should be inserted just before the boccaGoProlog protected block.

The go subroutine will be called by the framework when the component's run button (the name of this particular GoPort instance) is pushed in the GUI. Bocca generates the code to the Integration that the Driver is connected to. We just have to use it to compute the integral and return the proper value for *bocca_status*.

```
  /* DO-NOT-DELETE splicer.begin(demo.Driver.go) */


  /* User action portion is in the middle at the next Insert-UserCode-Here line. */


  /* Insert-User-Declarations-Here */

/* Bocca generated code. bocca.protected.begin(demo.Driver.go:boccaGoProlog) */

  int bocca_status = 0;
  /* The user's code should set bocca_status 0 if computation proceeded ok.
  // The user's code should set bocca_status -1 if computation failed but might
  // succeed on another call to go(), e.g. wheh a required port is not yet connected.
  // The user's code should set bocca_status -2 if the computation failed and can
  // never succeed in a future call.
  // The users's code should NOT use return in this function;
  // Exceptions that are not caught in user code will be converted to status -2.
  */


  gov_cca_Port port = NULL;
  gov_cca_Services services = NULL;
  sidl_BaseInterface throwaway_excpt = NULL;
  sidl_BaseInterface dummy_excpt = NULL;
  struct demo_Driver__data *pd = NULL;
  const char *errMsg = NULL;

  demo_Integration integrate = NULL; /* non-null if specific uses port obtained. */
  int integrate_fetched = FALSE; /* true if releaseport is needed for this port. */


  pd = demo_Driver__get_data(self);
  if (pd == NULL) {
    SIDL_THROW(*_ex, sidl_SIDLException,
       "NULL object data pointer in demo.Driver.go()");
  }
  services = pd->d_services;
  if (services == NULL) {
    SIDL_THROW(*_ex, sidl_SIDLException,
        "NULL pd->d_services pointer in demo.Driver.go()");
  }

  /* Use a demo.Integration port with port name integrate */
  port = gov_cca_Services_getPort(services,"integrate", &throwaway_excpt);
```

```
  if (throwaway_excpt != NULL) {
    port = NULL;
    errMsg="go() getPort(integrate) failed.";
    demo_Driver_checkException(self, throwaway_excpt, errMsg,
                                    FALSE, &dummy_excpt);
    /* we will continue with port NULL (never successfully assigned) and set a flag. */
    BOCCA_FPRINTF(stderr,
         "demo.Driver: Error calling getPort(\"integrate\") at %s:%d. Continuing.\n",
         __FILE__ , __LINE__ -8 );
  }

  if ( port != NULL ) {
    integrate_fetched = TRUE; /* even if the next cast fails, must releasePort. */
    errMsg="demo.Driver: Error casting gov.cca.Port integrate to type demo.Integration";
    integrate = demo_Integration__cast(port, _ex); SIDL_CHECK(*_ex);
    gov_cca_Port_deleteRef(port,_ex); port = NULL; SIDL_CHECK(*_ex);
  }


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




  /* When this block is rewritten by the user, we will not change it.
     All port instances should be rechecked for NULL before calling in user code.
     Not all ports need be connected in arbitrary use.
     The port instance names used in registerUsesPort appear as local variable
     names here.
     'return' should not be used here; set bocca_status instead.
   */

  /* Insert-UserCode-Here {demo.Driver.go} */



  if (integrate == NULL) {
    bocca_status = -1; /* not connected. skip computation. */
  } else {
    int count = 100000;
    double value = -4;
    double lowerBound = 0.0;
    double upperBound = 1.0;
    fprintf(stdout, "Initvalue = %g\n", value);
    value = demo_Integration_march(integrate, lowerBound, upperBound,
                                                  count, _ex); SIDL_CHECK(*_ex);
    fprintf(stdout, "Value = %g\n", value);
    fflush(stdout);
  }



  /* If unknown exceptions in the user code are tolerable and restart is ok,
     set bocca_status -1 instead.
```

```
     -2 means the component is so confused that it and probably the component or applica
     should be destroyed.
   */


EXIT:; /* target point for normal and error cleanup. do not delete. */
/* Bocca generated code. bocca.protected.begin(demo.Driver.go:boccaGoEpilog) */

  /* release integrate */
  if (integrate_fetched) {
    integrate_fetched = FALSE;
    gov_cca_Services_releasePort(services,"integrate",&throwaway_excpt);
    if ( throwaway_excpt != NULL) {
      errMsg= "demo.Driver: Error calling releasePort(\"integrate\"). Continuing.";
      demo_Driver_checkException(self, throwaway_excpt, errMsg, FALSE, &dummy_excpt);
    }
    if (integrate != NULL) {
      demo_Integration_deleteRef(integrate, &throwaway_excpt);
      errMsg = "Error in demo_Integration_deleteRef for demo.Driver port integrate";
      demo_Driver_checkException(self, throwaway_excpt, errMsg, FALSE, &dummy_excpt);
      integrate = NULL;
    }
  }


/* Bocca generated code. bocca.protected.end(demo.Driver.go:boccaGoEpilog) */


/* Insert-User-Exception-Cleanup-Here */

  return bocca_status;
  /* DO-NOT-DELETE splicer.end(demo.Driver.go) */
```

After quitting the editor the state of the source code tree is updated if there are any dependencies on the edited implementation. Usually there are no dependencies on the implementation file, so Bocca does very little after you exit the editor and all you see is the information from the edit command about what file was edited.

```
/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/c/demo/components/demo.Driver/demo_Driver_Impl.c
```

Now remake your project tree to finish the components:

```
$ make
```

```
make[1]: Entering directory '/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/c/demo'
# ============================================================================
# No SIDL files in external/sidl, skipping build for external
# ============================================================================
# ============================================================================
# Building in ports/, languages: c
# ============================================================================
 ## Building ports...
# ============================================================================
# Building in components/clients/, languages: c
# ============================================================================
 ## Building clients...
```

```
# ========================================================================
# Building in components/, languages: c
# ========================================================================
   [s] Building class/component demo.Driver:
   [s] creating class/component library: libdemo.Driver.la ...
   [s] finished libtooling: components/demo.Driver/libdemo.Driver.la ...
   [s] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/c/demo/install/share/cca/demo/demo.Dri
   [s] creating Ccaffeine test script (components/tests/instantiation.gen.rc)...
   [s] Building class/component demo.Function:
   [s] creating class/component library: libdemo.Function.la ...
   [s] finished libtooling: components/demo.Function/libdemo.Function.la ...
   [s] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/c/demo/install/share/cca/demo/demo.Fun
   [s] creating Ccaffeine test script (components/tests/instantiation.gen.rc)...
   [s] Building class/component demo.Integrator:
   [s] creating class/component library: libdemo.Integrator.la ...
   [s] finished libtooling: components/demo.Integrator/libdemo.Integrator.la ...
   [s] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/c/demo/install/share/cca/demo/demo.Int
   [s] creating Ccaffeine test script (components/tests/instantiation.gen.rc)...
   [s] Building class/component demo.emptyComponent:
doing nothing -- library is up-to-date.
Build summary:
SUCCESS building demo.Driver
SUCCESS building demo.Function
SUCCESS building demo.Integrator
### To test instantiation of successfully built components, run 'make check' ###
################ Finished building everything #################
####### You can run some simple tests with 'make check' #######
make[1]: Leaving directory `/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/c/demo'
```

It is good practice to do a `make check` at this point:

```
$ make check
```

```
make[1]: Entering directory `/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/c/demo'
make --no-print-directory --no-builtin-rules -C components check
### Test library load and instantiation for the following languages: c
Running instantiation tests only
###
LDPATH=/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/c/demo/install/lib:/usr/local/ACTS/cca/lib:/usr
###
PYTHONPATH=/usr/local/ACTS/cca/lib/python2.5/site-packages:/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scr
###
CLASSPATH=/usr/local/ACTS/cca/lib/sidl-1.4.0.jar:/usr/local/ACTS/cca/lib/sidlstub_1.4.0.jar:/usr/local/ACTS/cca/lib/
###
Test script: /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/c/demo/components/tests/instantiation.gen
Log file: /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/c/demo/components/tests/instantiation.gen.rc
SUCCESS:
==> Instantiation tests passed for all built components (see /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/s
make --no-print-directory --no-builtin-rules check-user
make[1]: Leaving directory `/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/c/demo'
```

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

3.5.4 Python Implementation

Note

Assumes you created the project with **`bocca create project demo --language=python`**.

Edit the `evaluate` method in the implementation file (also known as "the impl") that Bocca has generated for you (by invoking Babel). Use the `bocca edit -i` to go directly to each method.

```
$ bocca edit -i Function evaluate
```

The editor opens up in the place where the implementation code for `evaluate` must be put. You see a default implementation generated by Babel for all user methods: the throwing of an exception which says the method is not yet implemented.

```
  def evaluate(self, x):
    #
    # sidl EXPECTED INCOMING TYPES
    # ============================
    # double x
    #

    #
    # sidl EXPECTED RETURN VALUE(s)
    # =============================
    # double _return
    #

# DO-NOT-DELETE splicer.begin(evaluate)
    #
    # This method has not been implemented
    #

    # DO-DELETE-WHEN-IMPLEMENTING exception.begin(evaluate)
    noImpl = sidl.NotImplementedException.NotImplementedException()
    noImpl.setNote("This method has not been implemented.")
    raise  sidl.NotImplementedException._Exception, noImpl
    # DO-DELETE-WHEN-IMPLEMENTING exception.end(evaluate)
# DO-NOT-DELETE splicer.end(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 method is called by mistake. Delete this exception code and substitute an implementation for the `PiFunction` (i.e., the integral from 0 to 1 of $4/(1+x^2)$ is an approximation of π).

```
# DO-NOT-DELETE splicer.begin(evaluate)


    return 4.0 / (1.0 + x * x)


# DO-NOT-DELETE splicer.end(evaluate)
```

Now in the same file just above the `evaluate` method, find the second method for the `FunctionPort init` method:

```
# DO-NOT-DELETE splicer.begin(init)


    # Do nothing.
    pass


# DO-NOT-DELETE splicer.end(init)
```

We don't have any initialization in this simple example, so we just eliminate the code that throws the exception when the method is executed.

After quitting the editor the state of the source code tree is updated if there are any dependencies on the edited implementation. Usually there are no dependencies on the implementation file, so Bocca does very little after you exit the editor and all you see is the information from the edit command about what file was edited:

```
/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/python/demo/components/demo.Function/demo/Function_Imp
```

Similarly edit the `march` method in the `Integrator` with

```
$ bocca edit -i Integrator march
```

```
  def march(self, lowBound, upBound, count):
    #
    # sidl EXPECTED INCOMING TYPES
    # ============================
    # double lowBound
    # double upBound
    # int count
    #

    #
    # sidl EXPECTED RETURN VALUE(s)
    # =============================
    # double _return
    #

# DO-NOT-DELETE splicer.begin(march)
    #
    # This method has not been implemented
    #

    # DO-DELETE-WHEN-IMPLEMENTING exception.begin(march)
    noImpl = sidl.NotImplementedException.NotImplementedException()
    noImpl.setNote("This method has not been implemented.")
    raise  sidl.NotImplementedException._Exception, noImpl
    # DO-DELETE-WHEN-IMPLEMENTING exception.end(march)
# DO-NOT-DELETE splicer.end(march)
```

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

```
# DO-NOT-DELETE splicer.begin(march)


    # Use a demo.FunctionPort port with port name odeRHS
    try:
        port = self.d_services.getPort("odeRHS")
    except Exception,e:
        if self.bocca_print_errs:
            print "demo.Integrator: port odeRHS not connected."
        e.args = "demo.Integrator: port odeRHS not connected:\n%s" \
                 % e.args
        raise
    odeRHS = demo.FunctionPort.FunctionPort(port);
    if not odeRHS:
        if self.bocca_print_errs:
            print "demo.Integrator: Error casting port gov.cca.Port " \
                + "to odeRHS type demo.FunctionPort"
        ex = sidl.SIDLException.SIDLException()
        ex.setNote(__name__,0,
                   'Error casting self Port to demo.FunctionPort')
        raise sidl.SIDLException._Exception, ex

    # do the math
    h = (upBound - lowBound) / count
    retval = 0.0
    sum = 0.0
    for i in range(1,count+1):
        sum += odeRHS.evaluate(lowBound + (i - 1) * h)
        sum += odeRHS.evaluate(lowBound + i * h)

    retval = h/2.0 * sum
    self.d_services.releasePort("odeRHS")
    return retval


# DO-NOT-DELETE splicer.end(march)
```

After quitting the editor the state of the source code tree is updated if there are any dependencies on the edited implementation. Usually there are no dependencies on the implementation file, so Bocca does very little after you exit the editor and all you see is the information from the edit command about what file was edited:

```
/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/python/demo/components/demo.Integrator/demo/Inte
```

Finally for the `Driver` component we have to implement the `GoPort` details to get things going. Bocca will take you to the generated method, which looks like this:

```
$ bocca edit -i Driver go
```

```
  def go(self):
    #
    # sidl EXPECTED RETURN VALUE(s)
```

```
    # ============================
    # int _return
    #

    """\

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.
"""
# DO-NOT-DELETE splicer.begin(go)


# Bocca generated code. bocca.protected.begin(go:boccaGoProlog)
    bocca_status = 0
    # The user's code should set bocca_status 0 if computation proceeded ok.
    # The user's code should set bocca_status -1 if computation failed but might
    # succeed on another call to go(), e.g. when a required port is not yet connected.
    # The user's code should set bocca_status -2 if the computation failed and can
    # never succeed in a future call.
    # The users's code should NOT use return in this function;
    # Exceptions that are not caught in user code will be converted to status 2.

    if bocca_status == 0: # skip this getport if a problem already occured.
        # Use a demo.Integration port with port name integrate
        try:
            port = self.d_services.getPort("integrate")
        except sidl.BaseException._Exception, e:
            port = None
            if self.bocca_print_errs:
                (etype, eobj, etb) = sys.exc_info()
                msg="demo.Driver: Error calling getPort('integrate'): "                    +
                print >>sys.stderr,'Exception:', msg

        integrate_fetched = False;
        if not port:
            if self.bocca_print_errs:
                print 'demo.Driver: getPort("integrate") returned nil.'
        else:
            integrate_fetched = True # even if the next cast fails, must release.
            integrate = demo.Integration.Integration(port);
            if not integrate:
                bocca_status = -1
                if self.bocca_print_errs:
                    print "demo.Driver: Error casting port gov.cca.Port to "


    if bocca_status == 0: # all is ok so far and we do the user code, else cleanup and return
        # user code indents to match this.
# Bocca generated code. bocca.protected.end(go:boccaGoProlog)


        # If this try/catch block is rewritten by the user, we will not change it.
```

```
        try:
            try:
                # The user might not require all ports to be connected in all configurat
                # Each uses port is available as the local variable with the same name.
                # Those that are properly connected will be a value other than None.
                # the proper test for an unavailable port is "if not portinstancename:"


                # BEGIN REMOVE ME BLOCK
                ex = sidl.SIDLException.SIDLException()
                ex.setNote("USER FORGOT TO FILL IN THEIR FUNCTION demo.Driver.go()")
                raise sidl.SIDLException._Exception, ex
                # END REMOVE ME BLOCK

            except sidl.BaseException._Exception, e:
                bocca_status = -2
                if self.bocca_print_errs:
                    (etype, eobj, etb) = sys.exc_info()
                    msg="demo.Driver: Error in go() execution: "+eobj.exception.getNote(
                    print >>sys.stderr,'Exception:', msg
                # if specific exceptions in the user code are tolerable
                # and restart is ok, bocca_status -1 instead.
                # 2 means the component is so confused that it and probably
        # the application should be destroyed.
            except Exception,e:
                bocca_status = -2
                if self.bocca_print_errs:
                    print >> sys.stderr, 'Exception in demo.Driver go():'+str(e)
            except:
                bocca_status = -2
                print >> sys.stderr, 'Unclassified Exception in demo.Driver go()'
        finally:
            # always executed.
            pass
        # This version of TryExceptFinally for compatibility with python 2.3 and up


# Bocca generated code. bocca.protected.begin(go:boccaGoEpilog)
        # end user code.
    # end if bocca_status == 0.

    # release integrate
    if integrate_fetched:
        integrate_fetched = False
        try:
            self.d_services.releasePort("integrate")
        except sidl.BaseException._Exception, e:
            port = None
            if self.bocca_print_errs:
                (etype, eobj, etb) = sys.exc_info()
                msg="demo.Driver: Error calling releasePort('integrate'): "
                print >>sys.stderr,'Exception:', msg
```

```
    return bocca_status
# Bocca generated code. bocca.protected.end(go:boccaGoEpilog)


    #
    # This method has not been implemented
    #

# DO-NOT-DELETE splicer.end(go)
```

Find the `REMOVE` block within the `go` method implementation, delete it, and insert the numerical logic needed to *use* the `integrator.IntegratorPort` port. Any required local variables should be inserted just before the `boccaGoProlog` protected block.

The `go` subroutine will be called by the framework when the component's `run` button (the name of this particular `GoPort` instance) is pushed in the GUI. Bocca generates the code to the `Integration` that the `Driver` is connected to. We just have to use it to compute the integral and return the proper value for *bocca_status*.

```
# DO-NOT-DELETE splicer.begin(go)
# Bocca generated code. bocca.protected.begin(go:boccaGoProlog)
    bocca_status = 0
    # The user's code should set bocca_status 0 if computation proceeded ok.
    # The user's code should set bocca_status -1 if computation failed but might
    # succeed on another call to go(), e.g. when a required port is not yet connected.
    # The user's code should set bocca_status -2 if the computation failed and can
    # never succeed in a future call.
    # The users's code should NOT use return in this function;
    # Exceptions that are not caught in user code will be converted to status 2.

    if bocca_status == 0: # skip this getport if a problem already occured.
        # Use a demo.Integration port with port name integrate
        try:
            port = self.d_services.getPort("integrate")
        except sidl.BaseException._Exception, e:
            port = None
            if self.bocca_print_errs:
                (etype, eobj, etb) = sys.exc_info()
                msg="demo.Driver: Error calling getPort('integrate'): "                    +
                print >>sys.stderr,'Exception:', msg

        integrate_fetched = False;
        if not port:
            if self.bocca_print_errs:
                print 'demo.Driver: getPort("integrate") returned nil.'
        else:
            integrate_fetched = True # even if the next cast fails, must release.
            integrate = demo.Integration.Integration(port);
            if not integrate:
                bocca_status = -1
                if self.bocca_print_errs:
                    print "demo.Driver: Error casting port gov.cca.Port to "
```

```
    if bocca_status == 0: # all is ok so far and we do the user code, else cleanup and r
        # user code indents to match this.
# Bocca generated code. bocca.protected.end(go:boccaGoProlog)


        # If this try/catch block is rewritten by the user, we will not change it.
        try:
            try:
                # The user might not require all ports to be connected in all configurat
                # Each uses port is available as the local variable with the same name.
                # Those that are properly connected will be a value other than None.
                # the proper test for an unavailable port is "if not portinstancename:"



                count = 100000
                lowerBound = 0.0
                upperBound = 1.0

                # operate on the port
                value = integrate.march(lowerBound, upperBound, count)
                print 'Value =', value



            except sidl.BaseException._Exception, e:
                bocca_status = -2
                if self.bocca_print_errs:
                    (etype, eobj, etb) = sys.exc_info()
                    msg="demo.Driver: Error in go() execution: " \
                        + eobj.exception.getNote()
                    print >>sys.stderr,'Exception:', msg
                # if specific exceptions in the user code are tolerable
                # and restart is ok, bocca_status -1 instead.
                # 2 means the component is so confused that it and probably
        # the application should be destroyed.
            except Exception,e:
                bocca_status = -2
                if self.bocca_print_errs:
                    print >> sys.stderr, 'Exception in demo.Driver go():'+str(e)
            except:
                bocca_status = -2
                print >> sys.stderr, 'Unclassified Exception in demo.Driver go()'
        finally:
            # always executed.
            pass
        # This version of TryExceptFinally for compatibility with python 2.3 and up

# Bocca generated code. bocca.protected.begin(go:boccaGoEpilog)
        # end user code.
    # end if bocca_status == 0.

    # release integrate
```

```
    if integrate_fetched:
        integrate_fetched = False
        try:
            self.d_services.releasePort("integrate")
        except sidl.BaseException._Exception, e:
            port = None
            if self.bocca_print_errs:
                (etype, eobj, etb) = sys.exc_info()
                msg="demo.Driver: Error calling releasePort('integrate'): "
                print >>sys.stderr,'Exception:', msg


    return bocca_status
# Bocca generated code. bocca.protected.end(go:boccaGoEpilog)
# DO-NOT-DELETE splicer.end(go)
```

After quitting the editor the state of the source code tree is updated if there are any dependencies on the edited implementation. Usually there are no dependencies on the implementation file, so Bocca does very little after you exit the editor and all you see is the information from the edit command about what file was edited.

```
/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/python/demo/components/demo.Driver/demo/Driver_Impl.py
```

Now remake your project tree to finish the components:

```
$ make
```

```
make[1]: Entering directory '/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/python/demo'
# ============================================================================
# No SIDL files in external/sidl, skipping build for external
# ============================================================================
# ============================================================================
# Building in ports/, languages: python
# ============================================================================
 ## Building ports...
# ============================================================================
# Building in components/clients/, languages: python
# ============================================================================
 ## Building clients...
# ============================================================================
# Building in components/, languages: python
# ============================================================================

   [s] Building class/component demo.Driver:
make[3]: '.gencode' is up to date.
   [s] creating class/component library: libdemo.Driver.la ...
   [s] finished libtooling: components/demo.Driver/libdemo.Driver.la ...
   [s] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/python/demo/install/share/cca/demo/dem
   [s] creating Ccaffeine test script (components/tests/instantiation.gen.rc)...

   [s] Building class/component demo.Function:
make[3]: '.gencode' is up to date.
   [s] creating class/component library: libdemo.Function.la ...
   [s] finished libtooling: components/demo.Function/libdemo.Function.la ...
   [s] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/python/demo/install/share/cca/demo/dem
   [s] creating Ccaffeine test script (components/tests/instantiation.gen.rc)...

   [s] Building class/component demo.Integrator:
make[3]: '.gencode' is up to date.
   [s] creating class/component library: libdemo.Integrator.la ...
   [s] finished libtooling: components/demo.Integrator/libdemo.Integrator.la ...
```

```
   [s] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/python/demo/install/share/cca/de
   [s] creating Ccaffeine test script (components/tests/instantiation.gen.rc)...

   [s] Building class/component demo.emptyComponent:
make[3]: `.gencode' is up to date.
doing nothing -- library is up-to-date.
Build summary:
SUCCESS building demo.Driver
SUCCESS building demo.Function
SUCCESS building demo.Integrator
### To test instantiation of successfully built components, run 'make check' ###
################ Finished building everything #################
####### You can run some simple tests with 'make check' #######
make[1]: Leaving directory `/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/python/demo'
```

It is good practice to do a `make check` at this point:

```
$ make check
```

```
make[1]: Entering directory `/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/python/demo'
make --no-print-directory --no-builtin-rules -C components check
### Test library load and instantiation for the following languages: python
Running instantiation tests only
###
LDPATH=/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/python/demo/install/lib:/usr/local/ACTS/c
###
PYTHONPATH=/usr/local/ACTS/cca/lib/python2.5/site-packages:/home/livetau/workshop-acts/cca/WORK/tutorial-src/d
###
CLASSPATH=/usr/local/ACTS/cca/lib/sidl-1.4.0.jar:/usr/local/ACTS/cca/lib/sidlstub_1.4.0.jar:/usr/local/ACTS/cc
###
Test script: /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/python/demo/components/tests/instan
Log file: /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/python/demo/components/tests/instantia
SUCCESS:
==> Instantiation tests passed for all built components (see /home/livetau/workshop-acts/cca/WORK/tutorial-src
make --no-print-directory --no-builtin-rules check-user
make[1]: Leaving directory `/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/python/demo'
```

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

3.5.5 Java Implementation

Note

Assumes you created the project with **`bocca create project demo --language=java`**.

Edit the `evaluate` method in the implementation file (also known as "the impl") that Bocca has generated for you (by invoking Babel). Use the `bocca edit -i` to go directly to each method.

```
$ bocca edit -i Function evaluate
```

The editor opens up in the place where the implementation code for `evaluate` must be put. You see a default implementation generated by Babel for all user methods: the throwing of an exception which says the method is not yet implemented.

```
public double evaluate_Impl (
  /*in*/ double x )
  throws sidl.RuntimeException.Wrapper
{
  // DO-NOT-DELETE splicer.begin(demo.Function.evaluate)
  // Insert-Code-Here {demo.Function.evaluate} (evaluate)
  /*
   * This method has not been implemented
   */

  // DO-DELETE-WHEN-IMPLEMENTING exception.begin(demo.Function.evaluate)
  sidl.NotImplementedException noImpl = new sidl.NotImplementedException();
  noImpl.setNote("This method has not been implmented.");
  sidl.RuntimeException.Wrapper rex = (sidl.RuntimeException.Wrapper) sidl.RuntimeException
  throw rex;
  // DO-DELETE-WHEN-IMPLEMENTING exception.end(demo.Function.evaluate)
  // DO-NOT-DELETE splicer.end(demo.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 method is called by mistake. Delete this exception code and substitute an implementation for the `PiFunction` (i.e., the integral from 0 to 1 of $4/(1+x^2)$ is an approximation of π).

```
// DO-NOT-DELETE splicer.begin(demo.Function.evaluate)


return 4.0 / (1.0 + x * x);


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

Now in the same file just above the `evaluate` method, find the second method for the `FunctionPort init` method:

```
// DO-NOT-DELETE splicer.begin(demo.Function.init)


// Do nothing.


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

We don't have any initialization in this simple example, so we just eliminate the code that throws the exception when the method is executed.

After quitting the editor the state of the source code tree is updated if there are any dependencies on the edited implementation. Usually there are no dependencies on the implementation file, so Bocca does very little after you exit the editor and all you see is the information from the edit command about what file was edited:

```
/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/java/demo/components/demo.Function/demo/Function_Impl.
```

Similarly edit the `march` method in the `Integrator` with

```
$ bocca edit -i Integrator march
```

```
public double march_Impl (
  /*in*/ double lowBound,
  /*in*/ double upBound,
  /*in*/ int count )
  throws sidl.RuntimeException.Wrapper
{
  // DO-NOT-DELETE splicer.begin(demo.Integrator.march)
  // Insert-Code-Here {demo.Integrator.march} (march)
  /*
   * This method has not been implemented
   */

  // DO-DELETE-WHEN-IMPLEMENTING exception.begin(demo.Integrator.march)
  sidl.NotImplementedException noImpl = new sidl.NotImplementedException();
  noImpl.setNote("This method has not been implmented.");
  sidl.RuntimeException.Wrapper rex = (sidl.RuntimeException.Wrapper) sidl.RuntimeExce
  throw rex;
  // DO-DELETE-WHEN-IMPLEMENTING exception.end(demo.Integrator.march)
  // DO-NOT-DELETE splicer.end(demo.Integrator.march)
```

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

```
  // DO-NOT-DELETE splicer.begin(demo.Integrator.march)
```

```
gov.cca.Port port = null;
port = this.d_services.getPort("odeRHS");
demo.FunctionPort odeRHS;
odeRHS = ( demo.FunctionPort.Wrapper )
                demo.FunctionPort.Wrapper._cast((gov.cca.Port.Wrapper)port);
if (odeRHS == null) {
  if (bocca_print_errs) {
      System.err.println("demo.Integrator: Error casting gov.cca.Port"
                      + " odeRHS to type demo.FunctionPort");
  }
  sidl.SIDLException ex = new sidl.SIDLException();
  sidl.SIDLException.Wrapper msg = (sidl.SIDLException.Wrapper)
      sidl.SIDLException.Wrapper._cast(ex);
  throw msg;
}

double h = (upBound - lowBound) / count;
double retval = 0.0;
double sum = 0.0;
for (int i = 1; i <= count; i++){
  sum += odeRHS.evaluate(lowBound + (i - 1) * h) +
         odeRHS.evaluate(lowBound + i * h);
```

```
  }
  retval = h/2.0 * sum;

  this.d_services.releasePort("odeRHS");
  return retval;


    // DO-NOT-DELETE splicer.end(demo.Integrator.march)
```

After quitting the editor the state of the source code tree is updated if there are any dependencies on the edited implementation. Usually there are no dependencies on the implementation file, so Bocca does very little after you exit the editor and all you see is the information from the edit command about what file was edited:

```
/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/java/demo/components/demo.Integrator/demo/Integrator_I
```

Finally for the `Driver` component we have to implement the `GoPort` details to get things going. Bocca will take you to the generated method, which looks like this:

```
$ bocca edit -i Driver go
```

```
  public int go_Impl ()
    throws sidl.RuntimeException.Wrapper
  {
    // DO-NOT-DELETE splicer.begin(demo.Driver.go)


// Bocca generated code. bocca.protected.begin(demo.Driver.go:boccaGoProlog)
  int bocca_status = 0;
  // The user's code should set bocca_status 0 if computation proceeded ok.
  // The user's code should set bocca_status -1 if computation failed but might
  // succeed on another call to go(), e.g. wheh a required port is not yet connected.
  // The user's code should set bocca_status -2 if the computation failed and can
  // never succeed in a future call.
  // The users's code should NOT use return in this function;
  // Exceptions that are not caught in user code will be converted to status 2.

  gov.cca.Port port = null;

  boolean integrate_fetched = false;
  if (bocca_status == 0) { // skip further getports if problem occurs.
    // Use a demo.Integration port with port name integrate, unless we've hit
    // a problem already.
    try{
      port = this.d_services.getPort("integrate");
    } catch ( gov.cca.CCAException.Wrapper ex )  {
      // we will continue with port nil (never successfully assigned) and set a flag.
      if (bocca_print_errs) {
        System.err.println("Error calling getPort(\"integrate\")"
                           + ex.getNote());
        System.err.println("Continuing without integrate");
      }
```

```
    }
    demo.Integration integrate;
    if ( port != null  ) {
      integrate_fetched = true; // even if the next cast fails, must release.
      integrate = ( demo.Integration.Wrapper )
                  demo.Integration.Wrapper._cast((gov.cca.Port.Wrapper)port);
      if (integrate == null) {
        if (bocca_print_errs) {
          System.err.println("demo.Driver: Error casting gov.cca.Port "
                  + "integrate to type demo.Integration");
        }
        bocca_status = -1;
      }
    }
  }


  if (bocca_status == 0) {
// skip user code if we already have an unexpected error. go to cleanup.
// Bocca generated code. bocca.protected.end(demo.Driver.go:boccaGoProlog)



  // If this try/catch block is rewritten by the user, we will not change it.
  try {
    // All port instances may be rechecked for null before calling in user code.
    // Java will throw a null object exception when using the port if it's null.
    // The port instance names used in registerUsesPort appear as local variable
    // names here.

    // Insert-UserCode-Here {demo.Driver.go}

    // BEGIN REMOVE ME BLOCK
    sidl.SIDLException ex = new sidl.SIDLException();
    ex.setNote("USER FORGOT TO FILL IN THEIR FUNCTION demo.Driver.go()");
    sidl.BaseException.Wrapper bex =
      (sidl.BaseException.Wrapper)sidl.BaseException.Wrapper._cast(ex);
    throw bex;
    // END REMOVE ME BLOCK


  } catch (sidl.BaseException.Wrapper ex) {
    bocca_status = -2;
    if (bocca_print_errs) {
      System.err.println("SIDL Exception in user go code: "+ ex.getNote() );
      System.err.println("Returning 2 from go()");
    }
  } catch (java.lang.Exception jex) {
    bocca_status = -2;
    if (bocca_print_errs) {
      if (((sidl.BaseInterface)jex).isType("sidl.BaseException")) {
        System.err.println("sidl Exception in user go code: "
                           + ((sidl.BaseException)jex).getNote() );
      } else {
```

```
        System.err.println("java Exception in user go code: "+ jex.getMessage());
      }
      System.err.println("Returning 2 from go()");
    }
    // If unknown exceptions in the user code are tolerable and restart is ok,
    // use bocca_status -1 instead.
    // 2 means the component is so confused that it and probably the application
    // should be destroyed.
  }


// Bocca generated code. bocca.protected.begin(demo.Driver.go:boccaGoEpilog)
  } // cleanup

  // release integrate
  if (integrate_fetched) {
    integrate_fetched = false;
    try{
      this.d_services.releasePort("integrate");
    } catch ( gov.cca.CCAException.Wrapper ex )  {
      if (bocca_print_errs) {
        System.err.println("demo.Driver: Error calling "
            + "releasePort(\"integrate\"): " + ex.getNote());
      }
    }
    // java port reference will be dropped when function exits,
    // but we must tell framework.
  }


  return bocca_status;
// Bocca generated code. bocca.protected.end(demo.Driver.go:boccaGoEpilog)

    /*
     * This method has not been implemented
     */

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

Find the `REMOVE` block within the `go` method implementation, delete it, and insert the numerical logic needed to *use* the `integrator.IntegratorPort` port. Any required local variables should be inserted just before the `boccaGoProlog` protected block.

The `go` subroutine will be called by the framework when the component's `run` button (the name of this particular `GoPort` instance) is pushed in the GUI. Bocca generates the code to the `Integration` that the `Driver` is connected to. We just have to use it to compute the integral and return the proper value for *bocca_status*.

```
    // DO-NOT-DELETE splicer.begin(demo.Driver.go)
// Bocca generated code. bocca.protected.begin(demo.Driver.go:boccaGoProlog)
  int bocca_status = 0;
  // The user's code should set bocca_status 0 if computation proceeded ok.
  // The user's code should set bocca_status -1 if computation failed but might
```

```
  // succeed on another call to go(), e.g. wheh a required port is not yet connected.
  // The user's code should set bocca_status -2 if the computation failed and can
  // never succeed in a future call.
  // The users's code should NOT use return in this function;
  // Exceptions that are not caught in user code will be converted to status 2.

  gov.cca.Port port = null;

  boolean integrate_fetched = false;
  if (bocca_status == 0) { // skip further getports if problem occurs.
    // Use a demo.Integration port with port name integrate, unless we've hit
    // a problem already.
    try{
      port = this.d_services.getPort("integrate");
    } catch ( gov.cca.CCAException.Wrapper ex )  {
      // we will continue with port nil (never successfully assigned) and set a flag.
      if (bocca_print_errs) {
        System.err.println("Error calling getPort(\"integrate\")"
                           + ex.getNote());
        System.err.println("Continuing without integrate");
      }
    }
    demo.Integration integrate;
    if ( port != null  ) {
      integrate_fetched = true; // even if the next cast fails, must release.
      integrate = ( demo.Integration.Wrapper )
                  demo.Integration.Wrapper._cast((gov.cca.Port.Wrapper)port);
      if (integrate == null) {
        if (bocca_print_errs) {
          System.err.println("demo.Driver: Error casting gov.cca.Port "
                  + "integrate to type demo.Integration");
        }
        bocca_status = -1;
      }
    }
  }


  if (bocca_status == 0) {
// skip user code if we already have an unexpected error. go to cleanup.
// Bocca generated code. bocca.protected.end(demo.Driver.go:boccaGoProlog)

  // If this try/catch block is rewritten by the user, we will not change it.
  try {
    // All port instances may be rechecked for null before calling in user code.
    // Java will throw a null object exception when using the port if it's null.
    // The port instance names used in registerUsesPort appear as local variable
    // names here.



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

```
    // operate on the port
    value = integrate.march(lowerBound, upperBound, count);
    System.out.println("Value = "+ value);




  } catch (sidl.BaseException.Wrapper ex) {
    bocca_status = -2;
    if (bocca_print_errs) {
      System.err.println("SIDL Exception in user go code: "+ ex.getNote() );
      System.err.println("Returning 2 from go()");
    }
  } catch (java.lang.Exception jex) {
    bocca_status = -2;
    if (bocca_print_errs) {
      if (((sidl.BaseInterface)jex).isType("sidl.BaseException")) {
        System.err.println("sidl Exception in user go code: "
                           + ((sidl.BaseException)jex).getNote() );
      } else {
        System.err.println("java Exception in user go code: " + jex.getMessage() );
      }
      System.err.println("Returning 2 from go()");
    }
    // If unknown exceptions in the user code are tolerable and restart is ok,
    // use bocca_status -1 instead.
    // 2 means the component is so confused that it and probably the application
    // should be destroyed.
  }


// Bocca generated code. bocca.protected.begin(demo.Driver.go:boccaGoEpilog)
  } // cleanup

  // release integrate
  if (integrate_fetched) {
    integrate_fetched = false;
    try{
      this.d_services.releasePort("integrate");
    } catch ( gov.cca.CCAException.Wrapper ex )  {
      if (bocca_print_errs) {
        System.err.println("demo.Driver: Error calling "
            + "releasePort(\"integrate\"): " + ex.getNote());
      }
    }
    // java port reference will be dropped when function exits,
    // but we must tell framework.
  }


  return bocca_status;
// Bocca generated code. bocca.protected.end(demo.Driver.go:boccaGoEpilog)
    // DO-NOT-DELETE splicer.end(demo.Driver.go)
```

After quitting the editor the state of the source code tree is updated if there are any dependencies on the edited implementation. Usually there are no dependencies on the implementation file, so Bocca does very little after you exit the editor and all you see is the information from the edit command about what file was edited.

```
/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/java/demo/components/demo.Driver/demo/Driver_Imp
```

Now remake your project tree to finish the components:

```
$ make
```

```
make[1]: Entering directory '/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/java/demo'
# ============================================================================
# No SIDL files in external/sidl, skipping build for external
# ============================================================================
# ============================================================================
# Building in ports/, languages: java
# ============================================================================
 ## Building ports...
# ============================================================================
# Building in components/clients/, languages: java
# ============================================================================
 ## Building clients...
# ============================================================================
# Building in components/, languages: java
# ============================================================================

   [s] Building class/component demo.Driver:
make[3]: '.gencode' is up to date.
doing nothing -- library is up-to-date.

   [s] Building class/component demo.Function:
make[3]: '.gencode' is up to date.
doing nothing -- library is up-to-date.

   [s] Building class/component demo.Integrator:
make[3]: '.gencode' is up to date.
doing nothing -- library is up-to-date.

   [s] Building class/component demo.emptyComponent:
make[3]: '.gencode' is up to date.
doing nothing -- library is up-to-date.
################ Finished building everything #################
####### You can run some simple tests with 'make check' #######
make[1]: Leaving directory '/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/java/demo'
```

It is good practice to do a `make check` at this point:

```
$ make check
```

```
make[1]: Entering directory '/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/java/demo'
make --no-print-directory --no-builtin-rules -C components check
### Test library load and instantiation for the following languages: java
Running instantiation tests only
###
LDPATH=/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/java/demo/install/lib:/usr/local/ACTS/cca
###
PYTHONPATH=/usr/local/ACTS/cca/lib/python2.5/site-packages:/home/livetau/workshop-acts/cca/WORK/tutorial-src/d
###
CLASSPATH=/usr/local/ACTS/cca/lib/sidl-1.4.0.jar:/usr/local/ACTS/cca/lib/sidlstub_1.4.0.jar:/usr/local/ACTS/cc
###
Test script: /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/java/demo/components/tests/instanti
Log file: /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/java/demo/components/tests/instantiati
SUCCESS:
==> Instantiation tests passed for all built components (see /home/livetau/workshop-acts/cca/WORK/tutorial-src
make --no-print-directory --no-builtin-rules check-user
make[1]: Leaving directory '/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/java/demo'
```

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

3.6 Automated Testing of Assemblies

The gmake builder plugin that comes with Bocca provides support for automated regression testing of a project. The tests are Ccaffeine `rc` files, as introduced in Section 2.2. Tests are stored in the `components/tests` directory and end in the `rc` extension. The build system must be informed of the tests to be run by setting or extending the *`components_TESTS`* variable in `components/make.vars.user`. Once this is done, `make user-tests` will run all the tests defined in *`components_TESTS`*.

3.6.1 Creating a Portable Test

Tests require installation-dependent path information to find the components. The process that follows illustrates a portable test example.

1. Pick an suitably informative name for the test. Here we use "`odetest`".
2. In `components/tests/odetest.rc.in` create the portable test script by omitting any component path information:

```
create demo.Driver Driver
create demo.Integrator Integrator
create demo.Function Function
connect Driver integrate Integrator integrate
connect Integrator odeRHS Function fun
go Driver run
exit
```

3. Modify the *`post-build-user`* rule in `components/make.rules.user` to generate the non-portable, complete test. Between the `dprint` statements, insert code to join the build-generated instantiation test `rc` file to your portable test.

```
post-build-user::
        $(dprint) "Build hook $@ in $(MYDIR) started"
cat tests/instantiation.gen.rc | \
grep -v instantiate | grep -v display | \
grep -v remove | grep -v quit > tests/load.gen.rc
cat tests/load.gen.rc \
tests/odetest.rc.in > tests/odetest.rc
        $(dprint) "Build hook $@ in $(MYDIR) completed"
```

   Remember that makefile lines must be tabbed, not started with spaces.

4. Add the completed test script to the test list by simply:

```
$ echo "components_TESTS += odetest.rc" >> \
  components/make.vars.user
```

   As many tests as desired can be added to the list in this way.

5. Run **make** again to cause generation of `odetest.rc`.

6. Run **make user-tests** run the completed test file.

If you have multiple tests to manage, you may wish to create and add suffix rules that perform this conversion process. Future releases of Bocca will likely further automate the process of making test scripts portable.

3.6.2 Enabling Memory Testing with Valgrind

If you have the Valgrid package installed, you can add memory and other Valgrind-supported testing automatically by defining *CCAFE_VALGRIND* in the environment. In `bash`, for example:

```
export CCAFE_VALGRIND="valgrind -v"
```

In `tcsh`:

```
setenv CCAFE_VALGRIND "valgrind -v"
```

The value of *CCAFE_VALGRIND* is inserted before the Ccaffeine invocation, so tools other than valgrind may also be usable in this manner. In particular, if you have MPI included in your Ccaffeine framework build, you may be able to use *CCAFE_VALGRIND* to automatically launch with `mpiexec`. Note that if you have MPI included in your Ccaffeine framework, you may *not* be able to use `valgrind` directly in this way.

Chapter 4

A Simple PDE Toolkit

Tip

For this chapter, you will need to have your own copy of the PDE code tree built in your *WORKDIR*. If you haven't done this already, you might want to start it now, following the instructions in Appendix F. It takes a while to build.

4.1 Introduction

In this chapter, we will demonstrate how one builds a sophisticated software capability using a large collection of components. These collections of components are known as "toolkits", since the components are generally selected to solve a certain class of problems, e.g. partial differential equations (PDEs).

The toolkit being considered here is simple and rather incomplete with regard to its capability to solve complicated physical problems. For educational purposes it is meant to solve PDEs on Cartesian meshes. It has a mesh component (that takes care of the spatial discretization), a data object (that stores the dependent variables of a PDE on the space discretized by the mesh), a time-integrator, various components that model the physics and other components that enable data writing to disk and visualization. The algorithms and models being encoded in the components are simple, well known, easily understood and instructive in how a mathematical problem is decomposed and rendered in component software. Thoughtful design of the software architecture – component functionality and their level of granularity – is critical when designing toolkits. Design decisions about *interfaces* or *ports* are fundamental, and can be very hard change later without requiring wide-spread changes throughout the toolkit.

This simple toolkit is, however, quite complete from the software point of view. It contains a set of interfaces that are sufficient for the simple PDE example it solves. However, these interfaces can also serve quite well for the solution of nonlinearly coupled sets of PDEs and the numerical algorithms that are usually associated with them. The methodology and the infrastructure that you will learn, namely the interface design, the setup of a component using Bocca , the `makefiles`, the SIDL files are theoretically and practically industrial strength; you will not need anything more for research and publication-quality code.

In Section 4.2, we describe a PDE and its functional decomposition along numerical and physical-model lines. The "work flow", as relating to the algorithmic steps, is sketched out. In

Section 4.3, the functional pieces are rendered into components which are thereafter connected into a "code" (defined as a set of components, properly configured and connected, that can solve a given PDE). In Section 4.4 we outline a few exercises with components that we have created. In Section 4.5 we dig deeper into two components and suggest modifications; these can then become additions to the toolkit. In Section 4.6, we draw some conclusions.

4.2 A Problem and its Decomposition

We will model a Belusov-Zhabotinsky oscillator in which we've crafted the initial conditions to prevent oscillations, resulting in a travelling wave solution instead.

The system can be thought of as a reactant (ϕ_1) turning into a product (ϕ_2) in a temperature field (ϕ_0). The temperature, reactants, and products diffuse in space, from regions of higher concentration to regions of lower concentration. The initial conditions are such that the reactant is unevenly distributed in space, so it starts diffusing while simultaneously producing products. These products are produced only where the reactant is present, so they too start diffusing the moment they are produced.

The reaction term for the reactant is such that it is always positive (think of the reactant being produced from an infinite substrate), so it does not run out. It also diffuses out, thus forming a traveling wave. The same thing is reflected in the product; however its production gets shut off when it reaches a certain level, vis-a-vis the reactant concentration.

This system can be described by the following PDE on domain $\mathcal{D}$

$$\begin{aligned} \frac{\partial \mathbf{\Phi}}{\partial t} &= \mathbf{D}(\mathbf{\Phi}, \nabla\mathbf{\Phi}, \nabla^2\mathbf{\Phi}, \ldots; \mathbf{k}_D) + \mathbf{R}(\mathbf{\Phi}; \mathbf{k}_R) \\ \mathbf{\Phi}(\mathbf{x}, t=0) &= \mathbf{\Phi}^{(0)} \\ \mathbf{\Phi}(\mathbf{x}, t) &= \mathbf{q}(\mathbf{x}) \qquad \text{on } \partial\mathcal{D} \end{aligned} \tag{4.1}$$

This equation consists of two terms (on the right hand side): $\mathbf{D}$, which includes spatial derivatives and $\mathbf{R}$, which is a local term. Each of these terms are dependent on parameters $\mathbf{k}_D$ and $\mathbf{k}_R$. Initial and Dirichlet boundary conditions are also prescribed. We further define $\mathbf{D}$ and $\mathbf{R}$ to be

$$\mathbf{D} = \begin{pmatrix} \epsilon\nabla^2\phi_0 \\ \epsilon\nabla^2\phi_1 \\ \epsilon\nabla^2\phi_2 \end{pmatrix}, \qquad \mathbf{R} = \begin{pmatrix} 0 \\ \frac{1}{\epsilon}\left(\phi_1(1-\phi_1)(\phi_1-\phi_{th})\right) \\ (\phi_1 - \phi_2) \end{pmatrix} \tag{4.2}$$

where $\mathbf{k}_D = \epsilon$, $\mathbf{k}_R = \{0, \epsilon, 1\}$ and $\phi_{th} = (\phi_2 + q)/f$, $q = 0.01, f = 0.3$. ∇^2 is the Laplacian operator.

We wish to solve this equation on the 2D unit square defined on $0 \leq x \leq 1, 0 \leq y \leq 1$. $\mathbf{q}(\mathbf{x}) = \{0.1, 0.1, 0.1\}, \epsilon = 10^{-3}$. The initial condition for ϕ_0, ϕ_2 are 0.1, while for ϕ_1,

$$\phi_1(\mathbf{x}, 0) = 0.1 + 0.01\left(\exp(-a(\mathbf{x}-\mathbf{x}_1)^2) + \exp(-a(\mathbf{x}-\mathbf{x}_2)^2)\right) \tag{4.3}$$

where $a = (0.03)^{-2}, \mathbf{x}_1 = \{0.35, 0.35\}, \mathbf{x}_2 = \{0.65, 0.65\}$

While this equation can be solved in many ways, we will use second-order centered finite-difference stencils for ∇^2, i.e.

$$\nabla^2\phi = \frac{\phi_{i+1,j} - 2\phi_{ij} + \phi_{i-1,j}}{(\Delta x)^2} + \frac{\phi_{i,j+1} - 2\phi_{ij} + \phi_{i,j-1}}{(\Delta y)^2} \tag{4.4}$$

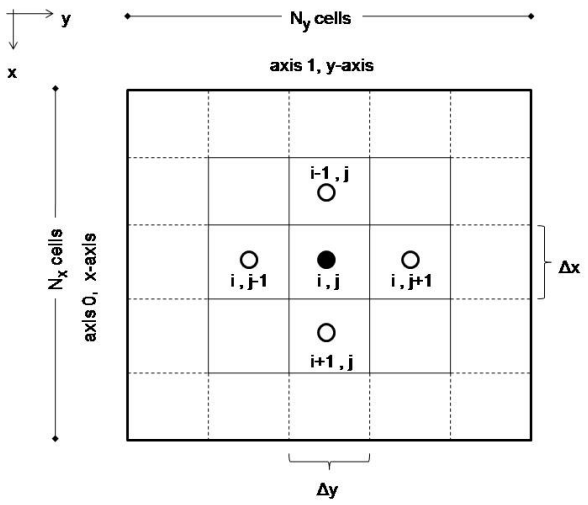

Figure 4.1: 2D central differences stencil.

where $\phi_{ij} = \phi(i\Delta x, j\Delta y)$. We will use a conventional RK2 time-integrator to solve this set of equations. We will use a $N_x \times N_y$ mesh (i.e. there will be $N_x \times N_y$ *grid cells*) to discretize $\mathcal{D}$ and a time step of Δt; N_x, N_y and Δt will be inputs into the code. Note, that N_x, N_y and Δt are not independent; $\Delta t \propto min\{{N_x}^{-2}, {N_y}^{-2}\}$ for stability. $\mathbf{\Phi}$ will be stored at cell-centers, and each cell will be $\Delta x \times \Delta y$ in size, $\Delta x = 1/N_x$ and $\Delta y = 1/N_y$. Fig. 4.1 depicts the mesh.

Solving this equation requires the following functionalities:

- **a Mesh**, which discretizes the domain;
- **a data object**, that contains the dependent variables on the cell-centers of the mesh;
- **initial conditions**, to impose $\mathbf{\Phi}^{(0)}$;
- **boundary conditions**, to impose $\mathbf{q}(\mathbf{x})$;
- **an RK2 integrator**, to advance in time;
- **physics components**, to evaluate $\mathbf{D}$ and $\mathbf{R}$;
- **data output components**, to store the data and a **visualizer**, to visualize the data; and
- **a driver**, to orchestrate the creation of variables, and the time-stepping.

These functionalities, implemented as components, can be used to solve the PDE and form a starting point for a PDE-solving toolkit.

4.3 Components and Assemblies

In this section we describe the components mentioned in Section 4.2 and a way in which they may be assembled together to solve Eq. 4.1. We start with the components:

Mesh: The Mesh is an object (providing the `mesh` interface) that addresses the discretization of the domain $\mathcal{D}$ using a Cartesian mesh. It decomposes the domain into a set of rectangular subdomains called *Regions* (they have a certain minimum size, so that they do not end up being just a single grid cell) which are then distributed among various processors (see Appendix A). The Mesh also allows the creation of data objects (which provide the `FieldVar` interface) on the Mesh. FieldVars can hold data at various times (this is required if one uses a multistep time-integrator like Crank-Nicholson) and this is reflected in the Mesh; i.e. it has a concept of time steps. The Mesh also provides the ability to define certain parameters which are common to all FieldVars e.g. the number of cells in the halo around the domain boundary (so that, for example, Neumann boundary conditions may be applied), and the radius of the spatial stencil (which determines the width of the ghost-cell halos one keeps around subdomains when computing in parallel). It also allows the specification of the domain (in terms of size and the $N_x \times N_y$ grid cells itself) as well as the Collocation type (see Chapter C).

FieldVar: A FieldVar is a data object, containing domain-decomposed dependent variables. It has the ability to supply the number of on-processor *Regions* and their particulars; the position in the domain that the Region corresponds to, and the references to the arrays that store the dependent variables declared in that Region. It has methods that allow the updating of ghost-cells and the imposition of boundary conditions. The data is stored in 1D arrays, and the FieldVar provides methods that translate the 2D (i, j) coordinates (as well as the 3D version) into an index into the 1D array which stores the variables.

RK2: A time-integrator that advances a field $\mathbf{\Phi}^n$ at time n, to time $n+1$. For the ODE system

$$\frac{d\mathbf{\Phi}}{dt} = \mathbf{F}(\mathbf{\Phi}, t) \tag{4.5}$$

it adopts the following algorithm

$$\begin{aligned} \mathbf{\Phi}^{(1)} &= \mathbf{\Phi}^n + \Delta t F(\mathbf{\Phi}^n, n\Delta t) \qquad \text{Stage 1} \\ \mathbf{\Phi}^{n+1} &= \mathbf{\Phi}^n + \frac{\Delta t}{2}\left(F(\mathbf{\Phi}^n, n\Delta t) + F(\mathbf{\Phi}^{(1)}, (n+1)\Delta t)\right) \end{aligned} \tag{4.6}$$

Therefore, the integrator has no idea of the dimensionality of space, but makes copious use of the functionality in the FieldVar for the imposition of boundary conditions and ghost-cell updates.

Initial Conditions: Given a FieldVar (or a vector of them), this component will impose $\mathbf{\Phi}^{(0)}$. This involves iterating through the various Regions and filling up the arrays containing $\mathbf{\Phi}$ with $\mathbf{\Phi}^{(0)}$.

Diffusion This is a component that operates on a Region-by-Region basis. Given a Region, it computes the term $\mathbf{D}$ using Eq. 4.3.

Reaction: This component computes the term $\mathbf{R}$, given a Region. Both $\mathbf{D}$ and $\mathbf{R}$ involve looping over all the points in a rectangular Region, accessing $\mathbf{\Phi}$ for the each of the points (and their neighbors) and computing the appropriate term.

RHSCombiner: This is an "adaptor" component that sums up $\mathbf{D}$ and $\mathbf{R}$ in a Region before returning the result as a 1D "right-hand-side vector" $\mathbf{F}$ to the integrator.

Visualizer: This is a component that given a FieldVar, writes each of the constituent fields (in our case, 3) to disk in a format readable by gnuplot.

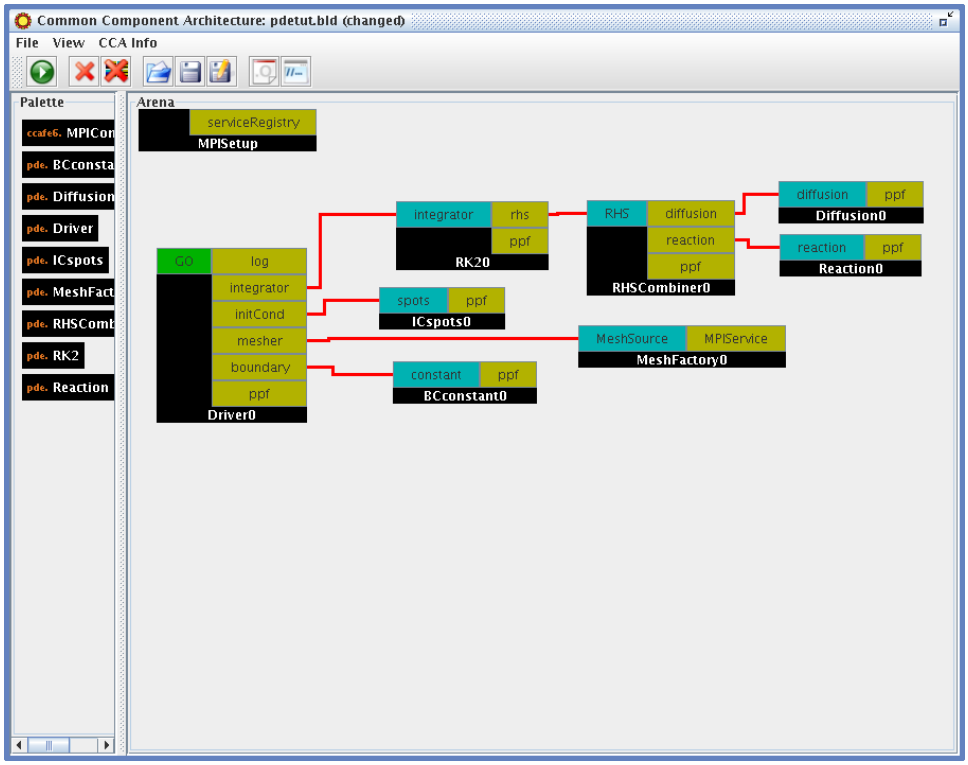

Figure 4.2: Example PDE application assembled from the components described in this section.

Driver: The Driver orchestrates the simulation. It declares the mesh (i.e. the unit square and the resolutions N_x and N_y) and creates the FieldVar containing the correct number of fields (3). It ensures that the Collocation is cell-centered. It then imposes the initial condition on the FieldVar (by sending it to the Initial Condition component).

Once the initialization is complete, the Driver goes into a time-loop. For each time step, it specifies a Δt to the integrator and sends in the FieldVar to the integrator to be advanced forward in time.

The integrator calls the RHSCombiner to obtain $\mathbf{F}$, given a $\mathbf{\Phi}$. The RHSCombiner obtains $\mathbf{D}$ and $\mathbf{R}$ separately from the Diffusion and Reaction components and returns the sum as $\mathbf{F}$ to the integrator.

Every so often, the Driver sends the FieldVar to the Visualizer for conversion to images and writing to disk.

This interaction among components described above can be seen in the connection diagram Fig. 4.2.

4.4 Tests

In this section, we will carry out some numerical experiments with a PDE application based on the toolkit just described.

The "code", i.e. the set of components, properly configured and connected is in *PDE_STUDENT_SRC*`/components/tests/example1.rc`; (see also Appendix B). Note that

the connection between the `RHSCombiner` and `Reaction` is commented out.

The `Driver` has been designed with a parameter port (part of the CCA specification) which allows you to change some of the basic simulation parameters without changing the code. In `example1.rc`, the application is configured to take 2000 time steps of size $dt = 1.0 \times 10^{-4}$.

We will now integrate forward in time and try to view the output of the code.

1. ```
   $ cd $PDE_STUDENT_SRC/components
   ```

2. In developing the PDE toolkit, we've added a user "test" to the build system, as described in Section 3.6. This allows us to easily run the application with the command

   ```
   $ make USER_TESTS=example1.rc example1.rc
   ```

   The simulation takes a few moments to execute, but at the end of the output from your `make` command, you should see messages similar to:

   ```
   Test script: /san/homedirs/bernhold/tutorial-src/obj/pde/components/tests/example1.rc
   SUCCESS:
   ==> Test passed, go command(s) executed successfully (see /san/homedirs/bernhold/tutorial-src/obj/pde/co
   ```

3. While the build scripts are designed to check and report whether or not the test completed successfully, it is a good idea to double check the log file until you become confident with the CCA and the particular application.

   Notice that the messages from the build system indicate where the log of the test is located (in this case *$PDE_STUDENT_SRC*`/components/tests/example1.rc.log`). Edit it, and jump to the bottom to confirm that it appears to have executed correctly.

   What you're seeing in this log is the output of the Ccaffeine session in which the application was run. So you see the `stdout` and `stderr` streams from the application and from Ccaffeine itself.

   A message that is repeated throughout the log hints at an interesting point about the CCA that we haven't discussed yet:

   ```
   !Info: Requested uses port dump in component RHSCombiner is not connected.
   ```

   (and similarly for the `reaction` port). A CCA component can *use* and *provide* whatever ports it wants, and use them in whatever way it considers appropriate. For example, a component may consider some port connections as "optional" – to be used if the port is connected, but if there is no connection, the component can do its job in some other way. In this case, the `RHSCombiner` component can *use* a `NamedPatchPort` (which it refers to as `dump`) to expose internal data for debugging or visualization purposes. But if it is not connected, the component skips those logging activities.
   ```

4. The application has also left us some more interesting and informative output in the directory in which we ran it (*$PDE_STUDENT_SRC*/components). You should see files with names like driver.ex1.step.1100.out.0,[1] which were dumped every 100 time steps by the application. These files are designed to be viewed using gnuplot (other visualization components are under development).

Note

If you're working remotely from the machine where you are running Ccaffeine , you'll need to have an X11 server on your local machine in order to view the simulation results using gnuplot. Linux/unix and Mac users may need to make sure that the X11 protocol is being tunneled through their ssh connection (see Appendix D.2). Windows users will need to run an X11 server (unfortunately, we have no documentation for that yet).

Let's gnuplot and then use it to explore the simulation output a little:

```
$ gnuplot
```

(a) We've created an input file for gnuplot that sets up a reasonable way to view the output from this application. To load it, type **load 'tests/show.gp'** at the "gnuplot>" prompt. You should see two "hot spots" which were part of the initial conditions given for this run.

(b) To view other time steps (for example step 1500), enter a command like **splot 'driver.ex1.step.1500.out' u 1:2:4** at the "gnuplot>" prompt. Notice that the hot spots diffuse away over time.

The triplet 1:2:4 at the end of the command line above tells gnuplot to use columns 1, 2, and 4 of the data file. The data file contains five columns of numbers: x, y, ϕ_0 (temperature), ϕ_1 (reactant), and ϕ_2 (product). So you can view other fields by changing the final element of the triplet.

5. Now we will run another simulation by modifying the parameters in *$PDE_STUDENT_SRC*/components/tests/example1.rc and re-running the simulation as in Step 2. Use the procedures of Step 4 to view the results.

Try changing the time step *dt*. Note that Changing the time step without corresponding changes to the mesh resolutions (parameters *NX* and *NY*) will cause the time integration algorithm to become unstable at some point, giving incorrect results. For example, with *dt* set to 10^2 instead of 10^{-4}, the system will rapidly become unstable, and you will see the boundaries of the regions of high reactant concentration (ϕ_1) become jagged instead of smooth, and may cease to be simply connected. Also, to minimum and maximum values of ϕ_1 will grow without bound with time.

[1] This filename includes a .0 suffix to associate it with MPI rank 0. If you built the CCA tools without MPI support, the filename will not have this suffix

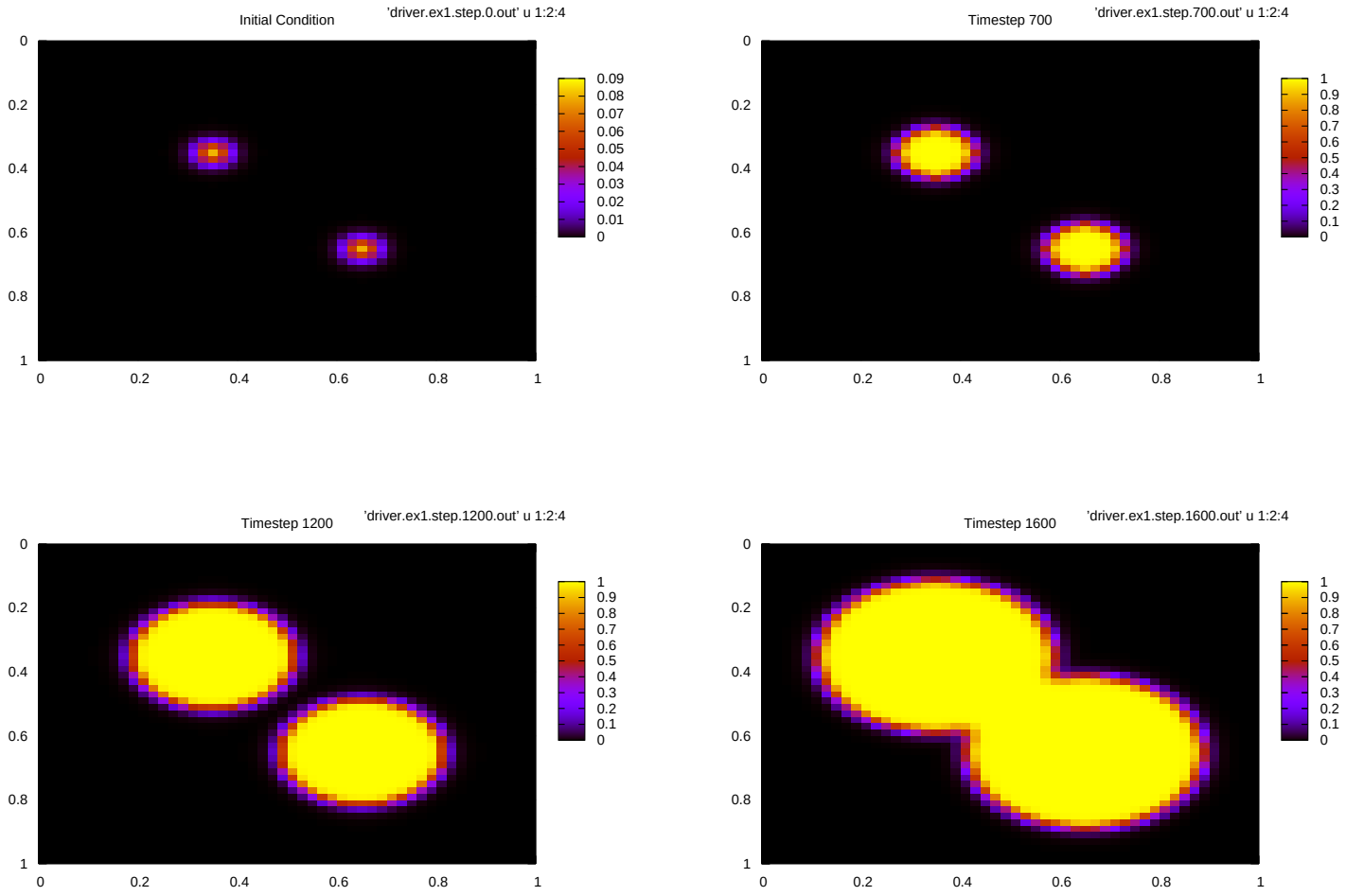

Figure 4.3: Traveling wave solution at t = 0, 700Δt, 1200Δt and 1600Δt, where Δt is 10^{-4} (Step 6

6. Finally, we will modify the application itself. Once again, edit *$PDE_STUDENT_SRC*/components/tests/example1.rc. First, restore the Driver parameters to their original values (see Appendix B if you've forgotten them). The *uncomment* the line that connects the RHSCombiner to the Reaction component. This changes the simulation from a pure diffusion problem to a reaction-diffusion problem. If you run this experiment and view the output, you should see traveling waves as shown in Fig. 4.3.

4.5 Exercises

In this section, we will learn to modify existing components to get "hands on" experience. This will give you confidence that you can modify components properly and correctly, before implementing components from scratch.

4.5.1 Changing the Initial Conditions

The current initial conditions component, `ICbzchem`, has two hot spots, which are implemented as Gaussian functions of radius *radiusSize_* centered at $\{0.35, 0.35\}$ (*center1*) and $\{0.65, 0.65\}$ (*center2*). It loops over all of the points in the given Region, and determines the point's distances to the two Gaussian centers (*d1* and *d2*). Based on these distances, the perturbation due to the hot spots (*perturb*) is computed, and added to the background (*var1Mag_*) to obtain the final value for ϕ_1, the concentration of the reactant. We're going to add another hot spot at $\{0.50, 0.65\}$.

1. Go to the top of the Bocca project for the PDE exercises:

```
$ cd $PDE_STUDENT_SRC
```

2. Edit the module (header) file for the `ICbzchem` component:

```
$ bocca edit -m ICbzchem
```

   and find the declaration for the variables *spotOne_* and *spotTwo_*. Add ***spotThree_*** to this declaration.

   (This component defines the locations of the hot spots in a rather quirky way. You'll see in moment how these variables are used.)

3. Edit the constructor for the `ICbzchem` component:

```
$ bocca edit -i ICbzchem ctor
```

   and find where the variables *spotOne_* and *spotTwo_* are initialized. Add ***spotThree_ = 0.50;*** to this declaration.

4. Now, edit the method that provides the initial conditions:

```
$ bocca edit -i ICbzchem setInitialConditions
```

Tip

This method is in the same file as the constructor you just edited, so you could simply have continued your editing session from the previous step and used the editor's search capability to find the `setInitialConditions` method.

   (a) Locate the declaration of the variables *center1* and *center2* (as vectors) and add ***center3***. Then right below that, initialize ***center3*** following the pattern you see for the other two centers. To place the new hot spot at $\{0.50, 0.65\}$, you'll need to use *spotThree_* and *spotTwo_* in the two positions of the vector.

   (I told you it was quirky!)

(b) Next go down to the loops over the `ix` and `iy` points in the Region, and locate where the distances `d1` and `d2` are computed. Add an analogous computation of the distance to the third center, ***d3***.

(c) Finally, add a third Gaussian to the definition of `perturb` using the distance ***d3***.

5. Recompile the project and verify that the modified component passes the basic instantiation tests:

```
$ make && make check
```

6. Use the procedures of Section 4.4 to run a simulation with the new initial conditions and view the results. You should see three hot spots instead of two. *Note that you've been working in the `$PDE_STUDENT_SRC`, while for the Section 4.4 procedures you need to be in `$PDE_STUDENT_SRC/components`.*

4.5.2 Modifying the Reaction Physics

In this exercise we're going to make a simple change to the model implemented by `Reaction` component. Instead of the temperature (ϕ_0) being constant, it will increase with time. Heat is produced in a localized area, and diffuses away from the source.

1. Go to the top of the Bocca project for the PDE exercises:

```
$ cd $PDE_STUDENT_SRC
```

2. Edit the implementation of the `compute` method of the `Reaction` component:

```
$ bocca edit -i Reaction compute
```

3. Scroll down to the line

```
double r0 = 0.0 ; // Temperature does not change in a BZ reaction
```

and comment it out with a **`//`** at the beginning of the line. Insert a new definition for `r0` *after* `r1` and `r2` are defined: **`double r0 = sqrt(r1*r2);`**.

4. Recompile the project and verify that the modified component passes the basic instantiation tests:

```
$ make && make check
```

5. Use the procedures of Section 4.4 to run a simulation with the new initial conditions and view the results. Now the ϕ_0 field should show structure. Remember that you can plot this field in gnuplot with the triplet `1:2:3` at the end of your `splot` command. *Note that you've been working in the `$PDE_STUDENT_SRC`, while for the Section 4.4 procedures you need to be in `$PDE_STUDENT_SRC/components`.*

4.6 Conclusions

We have the basis of a toolkit for solving PDEs. The toolkit allows the addition of new components, which is necessary since the current set of components is rather simple. It also allows the addition of new ports and interfaces. The exercises should give you some confidence in your ability to modify the component code in order to perform new simulations. Further, the current set of components, especially `Mesh` and `FieldVar` are quite sophisticated and can be used in realistic parallel codes. However, in order to use them properly, one has to understand their usage in detail. This is described in Appendix C.

Chapter 5

Using TAU to Monitor the Performance of Components

In this exercise, you will use the TAU performance observation tools to automatically generate a *proxy* component that monitors all of the method invocations on a port allowing you to track their performance information. While this approach won't provide all of the performance details of what is going on *inside* each component, it gives you a very simple way to begin analyzing the performance of a CCA-based application in order to identify which components might have performance issues.

We will start by creating a proxy component for the `Integration` port. Note that you only need to have completed Chapter 3 in order to follow these instructions. Though the proxy will be implemented in C++, it can be used as a proxy for components implemented in any language.

You should start this exercise on at the top of the Bocca project tree you created in Chapter 3,

```
$ cd $WORKDIR/demo
```

5.1 Creating the Proxy Component

The proxy component relies on the `Performance.Measurement` port that is part of the TAU CCA pacage installed in the directory *$TAU_CMPT_ROOT*. To enable the use of an externally-defined entity in your project, the SIDL type of this remote entity needs to be added to the bocca-generated build system. This can be done by adding a line of the form

```
CCA_TYPE_$(ENTITY_NAME) = ENTITY_TYPE
```

to the file `components/make.vars.user` in your project, where **`ENTITY_TYPE`** can be one of the values **`interface`**, **`port`**, **`class`**, or **`component`**.

In this case, we will be making use of the `Performance.Measurement` port. This means that, using your favorite editor, you will need to add the line

```
CCA_TYPE_Performance.Measurement = port
```

to the end of the file `components/make.vars.user`. You can then use Bocca to create the proxy component for the `Integration` port using the following command :

```
$ bocca create component IntegratorProxy \
  --language=cxx \
  --provides=Integration@IntegrationProvide \
  --uses=Integration@IntegrationUse \
  --uses=Performance.Measurement@measurement@/$TAU_CMPT_ROOT/ports/Per
```

```
Babel updating the cxx implementation of component demo.IntegratorProxy ...
```

Build the proxy component library:

```
$ make
```

```
make[1]: Entering directory '/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo'
# ============================================================================
# Building in external/, languages: cxx
# ============================================================================
 ## Building external...
   [c] using Babel to generate cxx client code for Performance.Measurement...
   [c] creating library: libPerformance.Measurement-cxx.la...
   [c] installing Performance.sidl
   [c] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/install/share/cca/demo/
# ============================================================================
# Building in ports/, languages: cxx
# ============================================================================
 ## Building ports...
# ============================================================================
# Building in components/clients/, languages: cxx
# ============================================================================
 ## Building clients...
# ============================================================================
# Building in components/, languages: cxx
# ============================================================================
   [s] Building class/component demo.Driver:
doing nothing -- library is up-to-date.
   [s] Building class/component demo.Function:
doing nothing -- library is up-to-date.
   [s] Building class/component demo.Integrator:
doing nothing -- library is up-to-date.
   [s] Building class/component demo.IntegratorProxy:
   [s] using Babel to generate cxx implementation code from demo.IntegratorProxy.sidl...
   [s] compiling sources...
   [s] creating class/component library: libdemo.IntegratorProxy.la ...
   [s] finished libtooling: components/demo.IntegratorProxy/libdemo.IntegratorProxy.la ...
   [s] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/install/share/cca/demo/
   [s] creating Ccaffeine test script (components/tests/instantiation.gen.rc)...
   [s] Building class/component demo.emptyComponent:
doing nothing -- library is up-to-date.
Build summary:
SUCCESS building demo.IntegratorProxy
### To test instantiation of successfully built components, run 'make check' ###
################# Finished building everything #################
####### You can run some simple tests with 'make check' #######
make[1]: Leaving directory '/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo'
```

This will give us a new component, called `IntegratorProxy` that implements the `Integration`

.

5.2 Using the Proxy Generator

1. In the `components/demo.IntegratorProxy` directory, invoke the proxy generator:

```
$ cd components/demo.IntegratorProxy
```

```
$ $TAU_CMPT_ROOT/bin/tau_babel_pg \
  -f demo_IntegratorProxy_Impl.cxx \
  -h ../../ports/demo.Integration/cxx/demo_Integration.hxx \
  -p integrate -t demo.Integration
```

The usage of the proxy generator is as follows:

```
Usage: tau_babel_pg <filename> -h <header file> -p <port name> \
-t <port type> [-f] [-m]
```

The `-h` option specifies the header file that needs to be included to use the port.

The `-p` option specifies the name of the port. The generated proxy will have two ports named with the port name given appended with "Provide" to distinguish them.

The `-t` option specifies the C++ type of the port. It can be found by examining the appropriate header file.

The `-f` option forces overwrite of the `_Impl.cc` and file `_Impl.hh` files.

The `-m` generates a MasterMind based proxy (not covered in this tutorial).

2. You can open `demo_IntegratorProxy_Impl.cxx` and look at the code that the proxy generator inserted between the splicer blocks to get a feel for what is really going on.

3. Build the proxy component:

```
$ cd ../..
```

```
$ make
```

```
make[1]: Entering directory '/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo'
# ============================================================================
# Building in external/, languages: cxx
# ============================================================================
 ## Building external...
   [c] using Babel to generate cxx client code for Performance.Measurement...
   [c] creating library: libPerformance.Measurement-cxx.la...
   [c] installing Performance.sidl
   [c] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/install/share/cca/demo/
# ============================================================================
```

```
# Building in ports/, languages: cxx
# ============================================================================
 ## Building ports...
# ============================================================================
# Building in components/clients/, languages: cxx
# ============================================================================
 ## Building clients...
# ============================================================================
# Building in components/, languages: cxx
# ============================================================================
   [s] Building class/component demo.Driver:
doing nothing -- library is up-to-date.
   [s] Building class/component demo.Function:
doing nothing -- library is up-to-date.
   [s] Building class/component demo.Integrator:
doing nothing -- library is up-to-date.
   [s] Building class/component demo.IntegratorProxy:
   [s] using Babel to generate cxx implementation code from demo.IntegratorProxy.sidl...
   [s] compiling sources...
   [s] creating class/component library: libdemo.IntegratorProxy.la ...
   [s] finished libtooling: components/demo.IntegratorProxy/libdemo.IntegratorProxy.la ...
   [s] building /home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo/install/share/cca
   [s] creating Ccaffeine test script (components/tests/instantiation.gen.rc)...
   [s] Building class/component demo.emptyComponent:
doing nothing -- library is up-to-date.
Build summary:
SUCCESS building demo.IntegratorProxy
### To test instantiation of successfully built components, run 'make check' ###
################ Finished building everything #################
####### You can run some simple tests with 'make check' #######
make[1]: Leaving directory '/home/livetau/workshop-acts/cca/WORK/tutorial-src/doc/scratch/cxx/demo'
```

5.3 Using the Proxy Component

1. Add the TAU performance component to the CCA path:

```
$ cp $TAU_CMPT_ROOT/components/TauPerformance-1.7.3/TauMeasurement.cc
  $WORKDIR/demo/install/share/cca
```

2. Next, add the following commands to construct the component assembly with the proxy component in place.

   Open `components/tests/guitest.gen.rc`, and add the following lines to the end of the file.

```
repository get-global TauPerformance.TauMeasurement

create TauPerformance.TauMeasurement tau
create demo.Driver driver
create demo.Function function
create demo.Integrator integrator
create demo.IntegratorProxy IntegratorProxy

connect driver integrate IntegratorProxy integrateProvide
connect IntegratorProxy MeasurementPort tau MeasurementPort
connect IntegratorProxy integrateUse integrator integrate
connect integrator odeRHS function fun
```

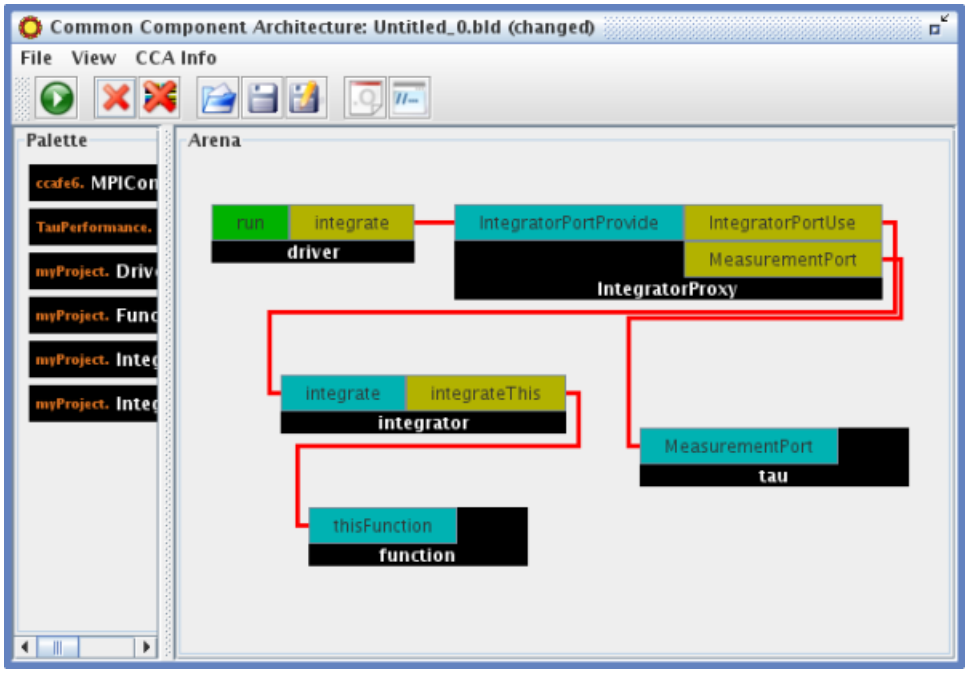

Figure 5.1: Integrator application with a proxy component to capture performance information for the `Integrator` component (Step 3).

3. Now run the component assembly, following the procedures you learned in Section 2.1. You GUI should look something like Figure 5.1.

4. Now look in the local directory and you should file a file called `profile.0.0.0`. This file contains profile data for the last run. View it by executing **`pprof`** and you should get output similar to this:

```
Reading Profile files in profile.*

NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------
100.0           26           26           3           0       8826 \
IntegratorProxy::march double (in *double, in *double, in *int32_t)
```

Users are encouraged to visit and read the documentation for TAU available at `http://www.cs.uoregon.edu/research/paracomp/tau/tautools/`.

Chapter 6

Understanding Arrays and Component State

Tip

To complete the exercises towards the end of this Chapter, you will need to have your own copy of the ODE code tree built in your *WORKDIR*. If you haven't done this already, you might want to start it now, following the instructions in Appendix F. It takes a while to build.

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.

6.1 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 are 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` are in *$TUTORIAL_SRC*`/components/`, in the directories `arrayOps.F90ArrayOp`, `arrayOps.F77ArrayOp`, and `arrayOps.CArrayOp`, respectively. In addition, a driver component that *uses* the `LinearOp` port is in *$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.

6.2 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.

6.2.1 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 row-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 per-

form 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,
                                    &throwaway_excpt);
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.

6.2.2 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 appears 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 *NumPy* 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 6.3.

6.3 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.

6.3.1 The `CArrayOp` Component

Code for the `CArrayOp` component is in *`$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__data));
   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`).

6.3.2 The `F77ArrayOp` Component

Code for the `F77ArrayOp` component is in *$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 is 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
```

Accessing 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
   . . .
```

6.3.3 The `F90ArrayOp` Component

Code for the `F90ArrayOp` component is in *$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 `arrayOps_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 builds the correct F90 array descriptor (dope vector) that correctly reflects the strides used to create the SIDL array.

6.4 Assignment: `NonLinearOp` Component and Driver

Note

Although you have been looking at the source code in *$TUTORIAL_SRC*, this exercise should be done in the Bocca project you created in Chapter 3.

```
$ cd $WORKDIR/demo
```

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 is in *$STUDENT_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);
  }
```

1. Use Bocca to create your own version of the `NonLinearOp` port specification by importing the existing definition from *$STUDENT_SRC*. This can be done using the command:

```
$ bocca create port arrayop.NonLinearOp \
  --import-sidl=arrayop.NonLinearOp@\
  $STUDENT_SRC/ports/sidl/arrayop.NonLinearOp.sidl
```

2. 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. 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 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. You can run the application using the technique you used in Chapter 2.

Appendix A

What is a Region in the Mesh

The Cartesian mesh ($N_x \times N_y$ cells in 2D) is decomposed by the Mesh object into a set ($N_p >= 1$) of rectangular non-overlapping patches which cover the mesh completely. A patch is fully defined with a set of corners and an array of integers. However, the data in a patch need to be expanded to accomodate the cell centered central difference stencil (ghost cells). That means that the data arrays of neighboring patches are overlapping in grid cell coordinates, therefore each of them must be described by a bigger "patch" called a *Region*. A Region is defined as a set of corners (Lower Corner or lbbc, Upper Corner or ubbc), a 1D data array and a pointer (reference) to the patch that corresponds to this Region. In Fig. A.1 we illustrate an example of a (3×3) patch and its ghost cells.

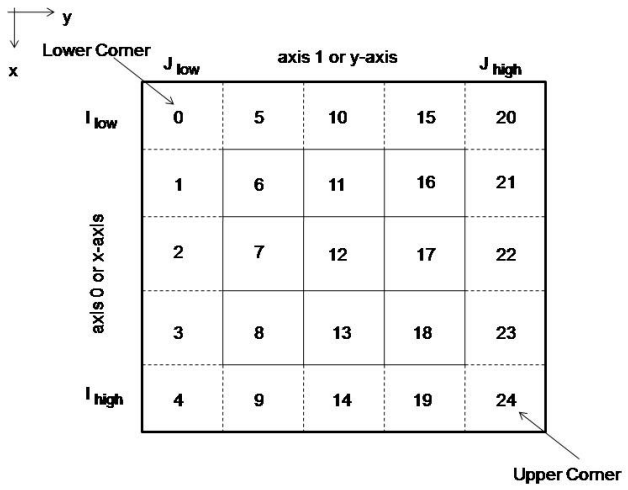

Figure A.1: An example of a Region of a (3×3) patch.

In memory the 1D data array for a Region maps to a 2D column-major array. Therefore given i, j in global 2D indexing the 1D local index is

Appendix B

Ccaffeine Script File for PDE Example 1

```
create pde.Driver               Driver
create pde.BCFactory            BCFactory
create pde.ICbzchem             ICbzchem
create pde.MeshFactory          MeshFactory
create pde.RK2                  RK2
create pde.RHSCombiner          RHSCombiner
create pde.Diffusion            Diffusion
create pde.Reaction             Reaction
create pde.viewers.MeshPrinter  viewer

connect           Driver        log             viewer          printer
connect           Driver        initCond        ICbzchem        ic
connect           Driver        bcSource        BCFactory       BCSource
connect           Driver        mesher          MeshFactory     MeshSource
connect           Driver        integrator      RK2             integrator

connect           RK2           rhs             RHSCombiner     RHS

connect           RHSCombiner   diffusion       Diffusion       diffusion
#connect           RHSCombiner   reaction        Reaction        reaction

parameter Driver userinput Nregions 1
parameter Driver userinput NX 50
parameter Driver userinput NY 50
parameter Driver userinput Nsteps 2000
parameter Driver userinput dt 0.0001
parameter Driver userinput DumpFreq 100

go Driver GO
```

Appendix C

Details of the `Mesh` and the `FieldVar` Classes

This chapter explains some of the terms commonly used in meshes. There are no standardized definitions for these terms, so we define a few here.

Fig. C.1 shows a domain D in black of length $Lx \times Ly$. It is divided into $Nx \times Ny$ cells. We will define fields (variables) on this discretized domain. The lower left corner of the domain corresponds to the origin of a coordinate system. The grid-cell size $\Delta x \times \Delta y$ is $Lx/Nx \times Ly/Ny$.

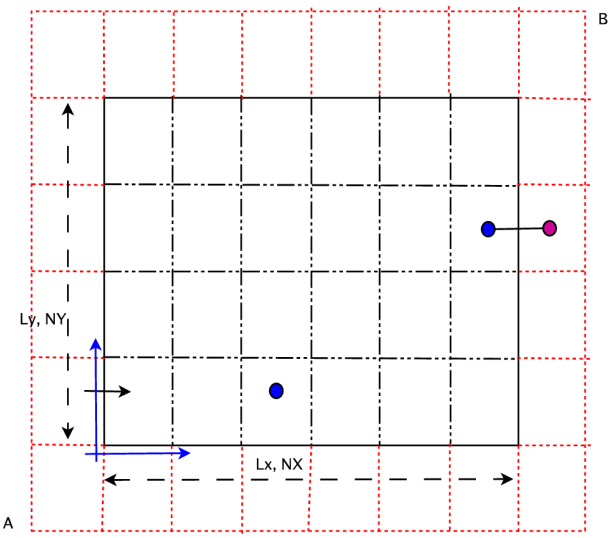

Figure C.1: A domain and its boundary halo.

Certain variables are stored at the center of cells and have a *Mesh Collocation Type* of *Cell-centered*; the blue circle shows a position where a cell-centered variable will be stored. However one may store variables at other places too e.g. a horizontal velocity stored on a face which has its face-normal in the horizontal direction. Such a variable is said to have a *FaceCenteredX* mesh collocation type. Other mesh collocation types are *FaceCenteredY, FaceCenteredZ* and *VertexCentered*, where the variables are stored at cell corners. Fig. C.2 shows these positions clearly.

Sometimes one may need to keep a *halo* of cells outside the domain D to help implement boundary conditions. The width of the boundary halo of cells, b, may be anything; it does not have to be equal to the width of any finite-difference/finite-volume stencil that you might be using. It might even be 0, for example, when implementing Dirichlet boundary conditions using second-order central difference. Fig. C.1 shows a boundary halo, 1 cell wide, in red.

Variables defined on the discretized domain are kept in arrays. However, when operating on arrays, it is frequently crucial to know the spatial locations of the domain where the variables are kept. The rectangular region is space, whose field variables are stored in an array `V`, is called the *patch* associated with the array `V`.

A patch of cells is defined by the lower and upper bounding box corners in integer cell coordi-

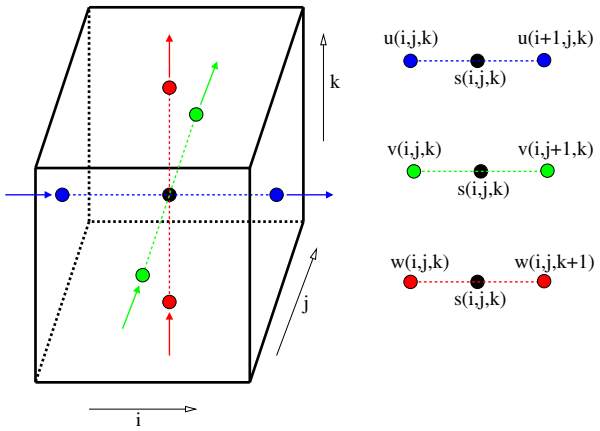

Figure C.2: Locations of variables. The black dot is the position where *CellCentered* variables are collocated, the blue dots are *FaceCenteredX*, the red one *FaceCenteredY* and the green ones *FaceCenteredZ*. The corners of the cube store *VertexCentered* variables.

nates. Patches are non-overlapping and completely cover the domain. The locations of these corners are described using an integer index stored in vectors `firstIndices[2]` and `lastIndices[2]`. These are 3 long for 3D meshes. For Fig. C.1, point A is the lower bounding box corner and has indices {`-1, -1`}; point B has indices {`Nx + 1, Ny + 1`}. However, note that the spatial location of A is $(-\Delta x, -\Delta y)$ and B is $(Lx + \Delta x, Ly + \Delta y)$.

However, the patch is an abstract concept and is rarely used. A more useful (and practical) concept is the *Region*. This is a consequence of parallel computing and the necessity of applying various types of boundary conditions. Consider Fig. C.4 which shows the situation in a domain-decomposed parallel computing paradigm. Subdomain $S2$ is shown with a red boundary halo (of width 1) and green ghost-cell halo of width 2. Generally, these widths are different - the ghost-cells halo width is determined by the width of spatial discretization stencils while the bounday halo width is determined by the boundary condition treatment. This is clearly seen in Fig. C.1, where the cell-centered variable in boundary halo interacts directly with the cell immediately inside the domain. On the other hand, in Fig. C.4, the discretization stencil has a radius of 2. These are specified separately when setting up the mesh. The ghost-cell halo is updated using MPI. The bounding box corners now play an important part in identifying the spatial location of an array. In Fig. C.4, F denotes the lower bounding box corner and has indices {`1, -1`} and spatial locations $(\Delta x, -\Delta y)$. The locations of all other points can be calculated with respect to F. The union of the patch (black cells in Fig. C.4), the ghost halo (green cells in Fig. C.4) and the boundary cells (red cells in the same figure) constitute the Region.

Variables are defined on Regions, not patches. A Region has a position in space, defined by its `lowerCorner` and `upperCorner`, as shown in Fig. C.3.. A real variable spread on a Region is stored as a 1-D double-precision array sized to account for the collocation type, boundary conditions, and stencils. The FieldVar interface provides access to the Regions and their corresponding data arrays. The each axis of a data array is as large or larger than the corresponding dimension in the Region (this is explained further in the next paragraph). The bounds of the data array are stored in vectors `lowerCorner[2]` and `upperCorner[2]`. These are 3 long for 3D meshes.

Consider a cell-centered variable `C` and a FaceCenteredX variable `U` defined on the patch in

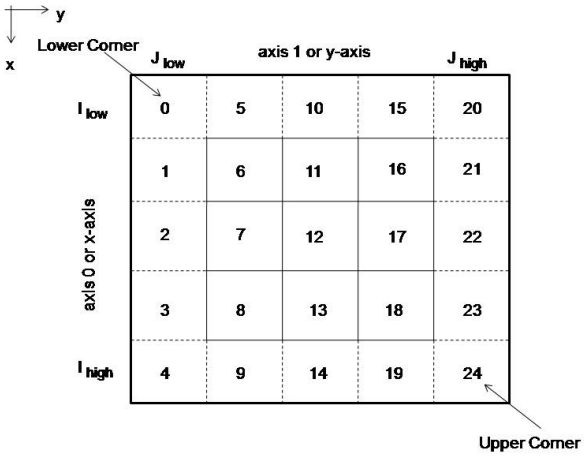

Figure C.3: A region showing the indexing of cells (and data arrays).

red Fig. C.1. Note that the dimensions of the arrays (in our terminology, their *shape*) containing the two variables are `C[Nx+2][Ny+2]` and `U[Nx+3][Ny+2]`. Whenever one operates on a variable (stored in an array), one also fetches the shape of the array, though one can calculate these based on the shape of the Region and the collocation type of the variable. However, it is a good programming practive to also fetch the corners of the Region and check whether the size of the Region and that of the array are consistent.

C.1 Codes

Below, we list some of the parameters defined above and means of obtaining their values from the mesh and field-variable methods

1. boundary width : `Mesh::getBoundaryWidth()`
2. stencil width (alternatively, width of the ghost-cell halo) : `Mesh::getStecilWidth()`
3. size of the domain i.e Lx, Ly : `Mesh::getDistances()`
4. resolution of the domain i.e. Nx, Ny : `Mesh::getShape()`
5. lower bounding box corner of a patch : `FieldVar::getLowerCorner()`
6. upper bounding box corner of a patch :`FieldVar::getUpperCorner()`
7. shape of the array living on that patch : `FieldVar::getShape()`
8. number of different field variables in that array : `FieldVar::getNVars()`
9. the pointer to the data: `FieldVar::get_data_raw(in int time, in int patchID, )`

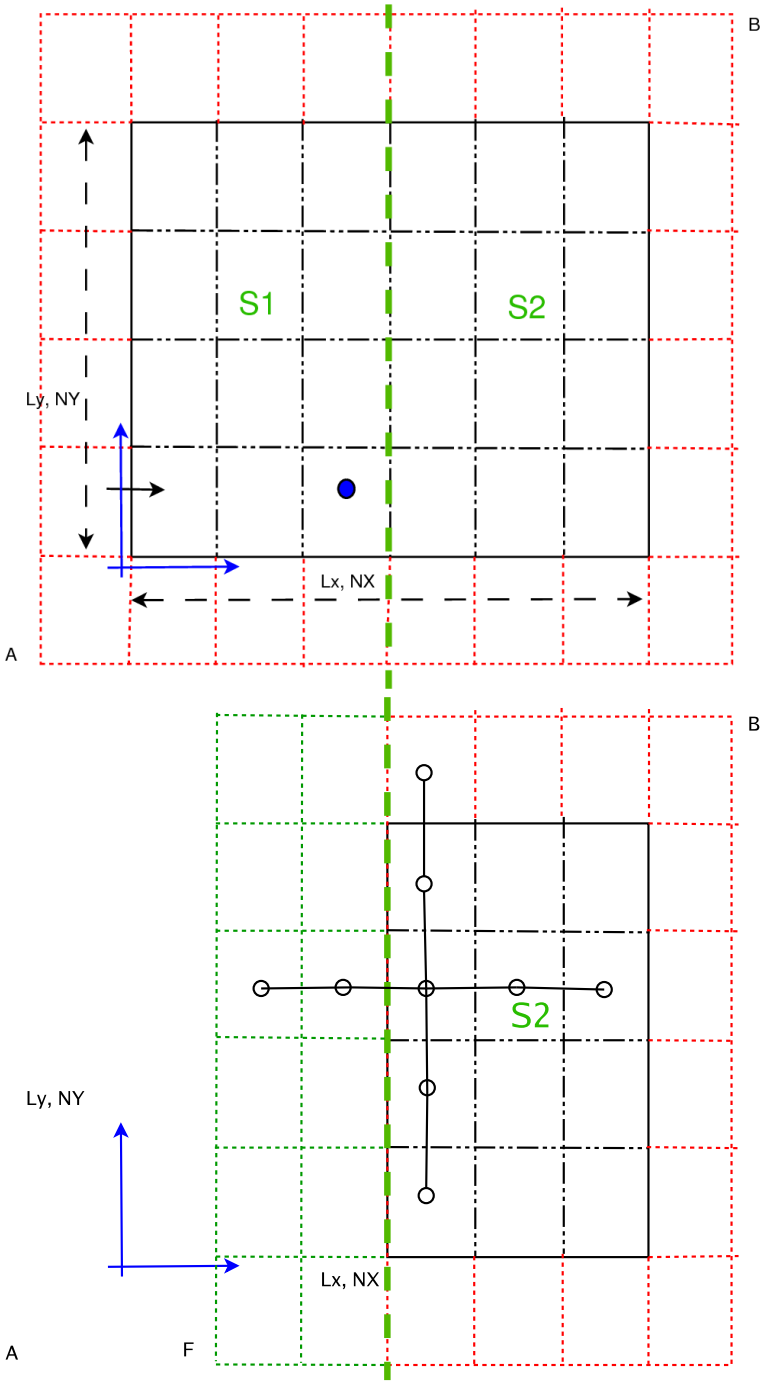

Figure C.4: Above: The domain D divided into 2 subdomains $S1$ and $S2$ along the thick green line. Below: The subdomain $S2$, showing the boundary halo in red and the ghost cells in green.

C.2 An Example

In the following I provide a C++ code snippet that initializes a `FieldVar`. References to an `FieldVar` and a `Mesh` are passed in. The `FieldVar` is assumed to hold 3 fields e.g temperature, pressure and concentration, which are all cell-centered and initialized using analytical functions.

```
#include <cassert>
#include ``math.h''

void initializeFieldVar( FieldVar &fv, Mesh &mesh )
{

  double Pi = 3.1416 ;

  // Make sure we're working in 2D
  assert( mesh.getDimension() == 2 );

  // get deltax, deltay
  double dx = mesh.getDistances()[0] / mesh.getShape()[0];
  double dy = mesh.getDistances()[1] / mesh.getShape()[1];

  // make sure that the incoming fieldvariable has 3 fields
  assert( fv.getNVars() == 3 );

  // make sure it is cell-centered
  assert( fv.getMeshColl() == CENTERS ) ;

  /*
    Time to start looping over all the patches on this processor
    stored on this proc. Figure out how many patches exist.
   */
   int npatches = fv.getPatchCount();

   // loop over patches
   for( int ipatch = 0; ipatch < npatches; ipatch++ )
   {
     // get the spatial starting locations of the cell-centered arrays.
     double startX = fv.getLowerCorner(ipatch)[0] * dx + 0.5*dx ;
     double startY = fv.getLowerCorner(ipatch)[1] * dy + 0.5*dy ;

     // get the dimensions of the patch
     int nx = fv.getShape(ipatch)[0] ;
     int ny = fv.getShape(ipatch)[1] ;

     /*
       get the current time and fetch the data pointer for the array
       which is supposed to hold the data for the current time
      */
     int itime = fv.getCurrentTime() ;
     double *data = (double *) fv.get_data_raw(itime, ipatch)

     // OK, all done. time to initialize. loop over all fields in
     // the FieldVar.
     for( int ifield = 0; ifield < fv.getNVars(); ifield++ )
```

```
    {
      // loop over all points in this patch
      for ( int j = 0; j < ny; j++ )
        for ( int i = 0; i < nx; i++ )
        {
          // figure out the x and y of this cell
          double x = startX + i*dx ;
          double y = startY + j*dy ;

          // figure out the stride for this cell and field;
          int index = i + j*nx + ifield*nx*ny ;

          if ( ifield == 0 )         // Temperature field
             data[index] = cos(Pi*x) ;
          else if ( ifield == 1 )       // pressure field
             data[index] = sin(Pi*x) ;
          else                          // concentration
             data[index] = tanh(x) ;
        }  // End of loop over points

     } // end of loop over fields

  } // End of loop over patches

  // all done; return

  return ;
}
```

Appendix D

Remote Access for the CCA Environment

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.

D.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.

D.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.)

D.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.

D.2.2 PuTTY

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

D.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.

D.4 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
```

```
$ simple-gui.sh --port 3314 --host localhost
```

- 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.

D.5 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 D.4), 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 E

Building the CCA Tools and TAU and Setting Up Your Environment

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 measurement 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 as illustrated in Chapter 5 (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.

E.1 The CCA Tools

E.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 cca-tutorial@cca-forum.org [`mailto:cca-tutorial@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` commands must be in your execution path.

Note

We have on occasion observed problems with the Ccaffeine 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.

- Gnome XML C Parser Library (libxml2) - most recent Linux distro's already have it, regardless of whether Gnome is installed. Make sure you have the development package, e.g., `libxml2-devel`.
- 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 and OpenMPI 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.
- 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 and later	Intel
Intel v7	Intel_7
Lahey	Lahey
NAG	NAG
SGI MIPS Pro	MPISpro
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 .

E.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` [`http://www.cca-forum.org/tutorials/#sources`] with a filename starting with `cca-tools-installer-`.
2. Untar the cca-tools tar ball some place that is convenient to build and follow the instructions in the README to configure and build it.

Caution

Remember that you need to have internet connectivity during the build process! During the installation process the appropriate versions of required software are downloaded automatically.

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 an error occurs, all logs are automatically archived in your home directory, in a file named `cca-tools-logs.tar.bz2`. You will be prompted to send an email as soon as the error occurs; you can also contact us at cca-tutorial@cca-forum.org [`mailto:cca-tutorial@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.

E.2.1 Local 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.

E.3 Downloading and Installing TAU

Note

Note that TAU is only needed for Chapter 5. If you're not planning to do that exercise, or want to delay installing TAU until then, everything else should work fine without it.

1. The latest version of the TAU Portable Profiling package can be found at http://www.cs.uoregon.edu/resea [`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/ [`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`**

E.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`*.

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.

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 `./install` expanded to be complete paths, rather than relative paths)

`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).

`TAU_VERSION` The version number of the TAU package you built.

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

csh, tcsh, and Related Shells

```
set path=(CCA_TOOLS_ROOT TAU_CMPT_ROOT $path)
setenv LD_LIBRARY_PATH CCA_TOOLS_ROOT/lib\
:TAU_COMPT_ROOT/components/TauPerformance-TAU_VERSION\
:$LD_LIBRARY_PATH
```

bash, ksh, sh, and Related Shells

```
export PATH=CCA_TOOLS_ROOT:TAU_CMPT_ROOT:$PATH
export LD_LIBRARY_PATH=CCA_TOOLS_ROOT/lib\
:TAU_COMPT_ROOT/components/TauPerformance-TAU_VERSION\
:$LD_LIBRARY_PATH
```

Warning

Note that *LD_LIBRARY_PATH* values are colon-separated lists, without white space. We show the variable definition folded onto multiple lines for presentation purposes, but in practice you would usually want to make it a single long line. It is very easy to inadvertently introduce extraneous white space, causing errors.

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.

Appendix F

Building the Tutorial Code Tree

The file `tutorial-src-0.7.1-0.tar.gz` (or a mode recent version) at `http://www.cca-forum.org/tutorials/#sources` has the full code for all of the components discussed in this Guide. If you're part of an organized tutorial, there will be a copy of the source code on the system you're using, so you can just copy it instead of downloading it over the network.

Note

In this release of the tutorial source code, we're experimenting with a new approach to organizing and building the code. It is likely there are some rough edges, so please don't hesitate to contact us at cca-tutorial@cca-forum.org [`mailto:cca-tutorial@cca-forum.org`] if you're having problems building it.

1. Make sure you have installed the CCA tools, and configured your environment according to Appendix E. *This critical, since the build for the tutorial source tree keys off of the environment variables setup in Appendix E.4.*

2. ```
   $ cd $WORKDIR
   ```

3. Download or copy the `tutorial-src-0.7.1-0.tar.gz` tarball to your *$WORKDIR*.

4. Unpack the tarball:

   ```
   $ tar zxf tutorial-src-0.7.1-0.tar.gz
   ```

5. ```
   $ cd tutorial-src
   ```

6. To build the "PDE" portion of the code tree, type

   ```
   $ ./pde-make all
   ```

The build will take some time.

Caution

After the build is complete, it tries to perform several basic tests on the components. Some of these tests are currently written to launch the Ccaffeine GUI, which may not work if you don't have an X11 connection to the machine you're building on. This is *not a problem* as long as the main build process completed successfully (scroll back through the `make` output a bit to check.) If you have problems, as for assistance (from tutorial instructors or at cca-tutorial@cca-forum.org [`mailto:cca-tutorial@cca-forum.org`]). We're working on improving this.

7. To build the "ODE" portion of the code tree, type

```
$ ./ode-make all
```

The build will take some time.

Caution

After the build is complete, it tries to perform several basic tests on the components. Some of these tests are currently written to launch the Ccaffeine GUI, which may not work if you don't have an X11 connection to the machine you're building on. This is *not a problem* as long as the main build process completed successfully (scroll back through the `make` output a bit to check.) If you have problems, as for assistance (from tutorial instructors or at cca-tutorial@cca-forum.org [`mailto:cca-tutorial@cca-forum.org`]). We're working on improving this.

If (a) you're part of an organized tutorial, and (b) you use the provided shell configuration fragments to setup your environment, and (c) you follow the directions above as to where to unpack and build the tutorial code tree, you should find that the environment variables *$STUDENT_SRC* and *$PDE_STUDENT_SRC* are already pointing to the tops of the Bocca project trees, in the directories `obj/ode` and `obj/pde` below the build directory (which should be you current directory).

If you're working through this Guide on your own, you should be sure to add to your environment the appropriate variable definitions: *$TUTORIAL_SRC* (interchangable with *$STUDENT_SRC* in this case) and *$PDE_SRC* (interchangable with *$PDE_STUDENT_SRC* in this case).

Note

If you want to do a clean build of either code tree, you can use (Substitute "ode" for "pde" as appropriate): **`./pde-make clean`** and then **`./pde-make all`**. The "clean" operation actually copies the `obj/pde` tree to a backup copy with a time-stamped directory name. If you're running out of disk space, you might need to clean up some of these backups.