

A Hands-On Guide to the Common Component Architecture

The Common Component Architecture Forum Tutorial Working Group

A Hands-On Guide to the Common Component Architecture

by The Common Component Architecture Forum Tutorial Working Group

Published 2004-08-30 21:54:59-04:00 (time this instance was generated)

Copyright © 2004 The Common Component Architecture Forum

Table of Contents

Preface	v
1. Help us Improve this Guide	v
2. Typographic Conventions	v
3. Acknowledgments	v
1. Introduction	1
1.1. The CCA Software Environment	1
1.2. The Execution Environment	1
1.3. Preparing to do the Exercises	2
2. Assembling and Running a CCA Application	6
2.1. A CCA Application in Detail	6
2.2. Running Ccaffeine Using an rc File	13
2.3. Using the GUI Front-End to Ccaffeine	15
3. The Driver Component	16
3.1. The SIDL Definition of the Driver Component	16
3.2. Implementation of the CXXDriver in C++	19
3.2.1. The setServices Implementation	20
3.2.2. The go Implementation	23
3.3. Implementation of the F90Driver in Fortran 90	25
3.3.1. The setServices Implementation	26
3.3.2. Implementing the Constructor and Destructor	30
3.3.3. The go Implementation	31
3.4. SIDL and CCA Object Orientation in Fortran	34
3.5. Using Your New Component	35
4. Creating a Component from an Existing Library	37
4.1. The legacy Fortran integrator	37
4.2. The FunctionModule wrapper.	39
4.3. Implementing the integrators.Midpoint component	40
4.4. SIDL definition of the Midpoint component	40
4.5. Fortran 90 implementation of the Midpoint integrator	42
4.5.1. The Midpoint module implementation	42
4.5.2. Defining the constructor and destructor	43
4.5.3. The setServices implementation	44
4.5.4. The integrate implementation	45
4.6. Building the Fortran 90 implementation of the integrators.Midpoint component.	47
4.7. Using your new integrators.Midpoint component	48
5. Creating a New Component from Scratch	50
5.1. SIDL Component Class Specification	50
5.2. Generating Babel Server Code for the New Component	51
5.3. Implementing the New Component	52
5.4. Using Your New Component	53
A. Installing the CCA Environment and Tutorial Source Code	54
A.1. Building the CCA Tool Chain	54
A.2. Building the Tutorial Source	55

Preface

\$Revision: 1.8 \$

\$Date: 2004/08/26 12:14:03 \$

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 eventually become complete enough that it can be used 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 [<http://www.cca-forum.org/tutorials/>] presentations before undertaking to complete the tasks in this Guide.

1. Help us Improve this Guide

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

2. Typographic Conventions

- *This font* is used for file and directory names.
- **This font** is used for commands.
- **This font** is used for input the user is expected to enter.
- *This font* is used for “replaceable” text or variables. Replaceable text is text that describes something you're supposed to type, like a *filename*, in which the word “filename” is a placeholder for the actual filename.
- *This font* is used for for the names of interfaces (i.e. CCA “ports”).
- URLs [<http://www.cca-forum.org/>] are presented in square brackets after the name of the resource they describe in the print version of the book.
- 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 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.

3. 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	Rob Armstrong, David Bernholdt (chair), Randy Bramley, Lori Freitag Diachin, Wael Elwasif, Madhusudhan Govindaraju, Ragib Hasan, Dan Katz, Jim Kohl, Gary Kumpfert, Lois Curfman McInnes, Boyana Norris, Craig Rasmussen, Jaideep Ray, Sameer Shende, Torsten Wilde, Shujia Zhou
Institutions	Argonne National Laboratory, Binghamton University - State University of New York, Indiana University, Jet Propulsion Laboratory, Los Alamos National Laboratory, Lawrence Livermore National Laboratory, NASA/Goddard, University of Illinois, Oak Ridge National Laboratory, Sandia National Laboratories, University of Oregon

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

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

Chapter 1. Introduction

\$Revision: 1.18 \$

\$Date: 2004/08/26 15:32:40 \$

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. We've intentionally chosen a very simple example 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.

In this integration example, which you've probably already seen mentioned in the tutorial presentations, we have:

- driver components, which are used like the `main` routine in a traditional program to orchestrate the overall calculation;
- a number of integrator components implementing different integration algorithms; and
- a selection of function components that can be integrated.

The exercises are laid out as follows:

- In Chapter 2, *Assembling and Running a CCA Application*, you will use pre-built components to assemble and run several different numerical integration applications.
- In Chapter 3, *Sewing CCA Components into an Application: the Driver Component*, you will construct your own driver component.
- In Chapter 4, *Creating a Component from an Existing Library*, you will wrap up an existing Fortran90 library as an integrator component.
- In Chapter 5, *Creating a New Component from Scratch*, you will create a new function component from scratch.

1.1. The CCA Software Environment

The CCA is, at its heart, just a specification. There are numerous realizations of the CCA as a software environment. In this Guide, we use the following tools to provide that software environment:

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

Babel A tool for language interoperability. It allows components written in different languages to be connected together. The Scientific Interface Definition Language (SIDL) is associated with Babel. For more information, see <http://www.llnl.gov/CASC/components/babel.html>.

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. The Execution Environment

The instructions in this guide assume you will be working on the Thor cluster at the Indiana University Computer Science Department. Thor runs Red Hat Linux 8.0. The tutorial instructors will provide you

with information you need to get logged in to the system. The following information may help you navigate around the system:

- This system has the Intel Fortran Compiler 8.0 (**ifort**) installed for Fortran90/95 code.
- Your home directory will be something like `/.automount/whale/root/san/r1a010/username`, depending on the specific disk and your username. In these exercises, you will be working primarily in your home directory tree, but you'll need to reference some files outside of it as well. If you see a relative filename in this Guide (i.e. one that does *not* begin with a "/") you should read it relative to your home directory. So that `student-src/Makefile` would refer to `/.automount/whale/root/san/r1a010/username/student-src/Makefile`
- A variety of files and tools have been installed for you to use in these exercises. The root of the installation is `/san/shared/cca/tutorial`, which once you complete the setup procedure below, you will be able to refer to as `$CCA`. We will always use the notation `$CCA/file` to refer to files in this tree except in situations where the `$CCA` would not be properly expanded.
- The CCA software tools (Ccaffeine, Babel, and related things) are installed in `$CCA/bin`.
- `$CCA/share` contains several files that you will reference or use, including the source code tree that you will start from to build your own components, and the SIDL files for the CCA specification.
- `$CCA/src` contains a pre-built version of the entire set of components for these exercises. In Chapter 2, *Assembling and Running a CCA Application*, you will use these components directly. You will also find the complete source code, including for the components you will write in the subsequent exercises. So if you're ever stuck, or unsure if you've done something correctly, you can look at the corresponding file in this tree and compare it to your own.

1.3. Preparing to do the Exercises

1. Obtain your individual username, password, and assignment of which node on the Thor cluster you will use from the tutorial staff.
2. Use an ssh client to login to the appropriate system.
 - a. If you're working on a Linux or other unix-based system, you will probably use a command like `ssh -l username thorN.cs.indiana.edu`. You will be prompted for your password.
 - b. There are a variety of ssh clients for Windows. In PuTTY, for example, you would enter the `thorN.cs.indiana.edu` in the Session->Host Name field of PuTTY Configuration window. Make sure the Protocol is set to SSH, as well. On the Connection page, set the Auto-login username field to the username you were given. Finally, return to the Session page, give these settings a name, such as `cca-tutorial` in the Saved Sessions field and click the Save button. This will allow you to quickly load these settings when you have to log back in later.

If you have trouble getting your ssh client to connect to the Thor cluster, please ask for assistance.



Note

We have written this guide so that all exercises can be performed simply using the command line tools, so it is not necessary for you to have an X11 server on your local system. If you *do* have an X11 server available, you may wish to try to use the GUI front-end for Ccaffeine in some of these exercises, and you may prefer to use the graphical version of your favorite text editor or other tools. However if the network performance between here and Indiana is poor, or systems on either end are too heavily loaded, the X11 option may be too slow for your taste. For these reasons, and depending on the defaults on your ssh client, you may need to *enable* or *disable* tunneling of the X11 protocol through the ssh session.

On unix/Linux clients, the command line switches `-x` and `-X` disable and enable X11 forwarding.

On PuTTY, there is a checkbox to Enable X11 forwarding on the Connection->SSH->Tunnels configuration page.

3. Since the CCA tools are not installed on this system in the “usual” locations (i.e. `/usr/bin` or `/usr/local/bin`), you must setup your login environment to the appropriate directories to your `PATH` and `LD_LIBRARY_PATH`. To speed things along, we've created a short script that you can just include in your login files.

Type `echo $SHELL` to determine which shell you're running. If it is a C-shell variant (i.e. `/bin/tcsh` or `/bin/csh`) then perform Step 3.a. If it is a Bourne shell variant (i.e. `/bin/bash` or `/bin/sh`) then perform Step 3.b.

- a. Edit `~/.cshrc` and at the end of the file, add the line

```
source /san/shared/cca/tutorial/share/cca.cshrc.thor.
```

- b. Edit `~/.profile` and at the end of the file, add the line

```
./san/shared/cca/tutorial/share/cca.profile.thor.
```



Warning

Note that you cannot use `$CCA/share/cca.cshrc.thor` (or `$CCA/share/cca.profile.thor`) in this case because this is the file that *defines* the `$CCA` environment variable !

You may wish to take a moment to read through this file to understand what it is doing to your environment. Each section is commented.

4. In order for these changes to affect your environment, you need to log out and log back in again. After you've done this, check that you're getting the new settings. If you type `echo $CCA`, you should get `/san/shared/cca/tutorial`. If you type `which ccafe-single`, you should get `/san/shared/cca/tutorial/bin/ccafe-single`. If you don't get these results, please ask for assistance.
5. In order to get a private copy of the tutorial source tree that you can work on, make sure you're in your home directory and enter `tar xf $CCA/share/student-src.tar`. This should give you a directory tree named `student-src`. This tree is the same as the pre-built one in `$CCA/src` except that we have removed all of the files that you'll be creating and "unmodified" the ones you'll be modifying as you work through the exercises.

The layout of the tutorial source code tree was designed to make it easy to introduce new components and ports and have everything built with **make** with minimal configuration. The tree is laid out so that much of the information the build system needs comes directly from the file and directory names. If you **cd** into `student-src`, you should see a number of subdirectories. Of primary interest are:

<code>student-src/components</code>	The source code for the various components lives in this tree. The general structure is <code>student-src/components/component_name/implementation_language</code> . Exceptions are the <code>student-src/components/sidl</code> and <code>student-src/components/examples</code> directories, which contain the SIDL definitions for the components and example scripts, respectively.
<code>student-src/legacy</code>	This is where the legacy libraries that you will componentize in Chapter 4, <i>Creating a Component from an Existing Library</i> reside.
<code>student-src/ports</code>	<code>student-src/ports/sidl</code> contains the SIDL files for the ports (SIDL interfaces) needed for the tutorial. The code generated by Babel for these interfaces will be put into directories like <code>student-src/ports/SIDL_package_name/language</code> and compiled into libraries. Both the <i>user</i> and <i>provider</i> of a port need to link against the port's library.

6. The next step is to build the tree. Although it is incomplete with respect to the code you're going to add in these exercises, everything that is there should build correctly.

Change directories to `student-src` and type **make**. This command will take several minutes to complete, and you may want to use this time to read ahead a bit. When it completes, you should see this message:

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

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

Once the build is complete, you can type **make check** to perform a basic check that the component have been built correctly. This is a convenience of the Makefile system we've put together for the tutorial that tries to instantiate each component within the Ccaffeine framework (you'll understand this better after the next chapter). This provides a basic check that the software you've built are “well-formed” CCA components. You should see a message like this, along with a couple of lines of output from **make** itself:

```
#### Testing component instantiation.

====
==== Simple tests passed, all built components were successfully \
instantiated.
====

#### Testing component connection and execution.
```

```
****
**** Some run tests did NOT succeed, go command failed (see \
      examples/ex1_rc.log).
****
```

Note that the test of component connection and execution is expected to fail at the moment, because it expects to have *all* of the components available, whereas at the moment, the ones you're going to write in these hands-on exercises are missing. The main thing to look for at this moment was that all components that are present could be instantiated (the first test).

Now you should be ready for the first exercise.



Note

If you move your `student-src` tree to another location (for example, rearranging your directory structure, or moving a copy of your `thor` directory tree to your home system to continue these exercises on your own), you will need to do a **make clean** rebuild the tree (starting with Building the tutorial source tree, above). Otherwise many of the generated files (`rc` files and the like) will contain the incorrect path.

Chapter 2. Assembling and Running a CCA Application

\$Revision: 1.17 \$

\$Date: 2004/08/26 15:59:31 \$

In this exercise, you will work with pre-built components from the integrator example to compose a CCA-based application and execute it. Specifically you will use a Monte Carlo integration algorithm on the function $4/(1+x^2)$, which gives pi as the result.

The components available are:

Drivers:	<code>drivers.CXXDriver*</code> , <code>drivers.F90Driver*</code>
Integrators:	<code>integrators.MonteCarlo</code> , <code>integrators.Midpoint*</code>
Functions:	<code>functions.PiFunction</code> , <code>functions.CubeFunction*</code>
Random Number Generators:	<code>randomgens.RandNumGenerator</code> (required by <code>integrators.MonteCarlo</code>)

Components marked with a “*” are ones that you will be creating in the subsequent exercises (you only need to do one of the two driver components), but as we have mentioned, the pre-built tree include completed examples of *all* of the components.

Below are three different procedures for this exercise. In Section 2.1, “A CCA Application in Detail”, you interact directly with Ccaffeine on the command line to do everything. This is the best place to start to understand how to assemble and run a CCA application. In Section 2.2, “Running Ccaffeine Using an rc File”, you will see how the steps you performed manually in the first procedure can be captured in a script that Ccaffeine reads. This is the more common scenario because it gives you an easy way to represent a complete CCA application that is easy to reproduce, or to adapt to other situations, without having to re-do everything from scratch every time you want to run it. This is probably the approach you’ll want to use when testing your work in the subsequent exercises. Finally, in Section 2.3, “Using the GUI Front-End to Ccaffeine”, we use a graphical front-end to Ccaffeine, which allows you to perform the composition and execution of the application using a “visual programming” metaphor. This procedure will only work if you have an X11 windowing system installation on your machine.

2.1. A CCA Application in Detail

In this section, you will interact directly with the Ccaffeine framework to assemble and run several different numerical integration applications from pre-built components.

We will present the procedure in the form of a dialog between you and the Ccaffeine framework. Things you are supposed to type are presented **like this** and Ccaffeine's output will be presented **like this**. Note that Ccaffeine's input prompt is “cca>”. Particular features of the output will sometimes be marked and discussed in further detail below the output fragment.



Tip

The complete set of Ccaffeine commands for this procedure can be found in `CCA/src/components/examples/task0_rc`. You can use this file for reference, or to cut and paste commands into Ccaffeine.

1. Start the Ccaffeine framework with the command **ccafe-single**. **ccafe-single** is one of several ways to invoke the Ccaffeine framework, and is used for single-process (i.e. sequential) interactive sessions; **ccafe-batch** is designed for use in non-interactive situations, including parallel jobs; and **ccafe-client** is designed to interact with a front-end GUI rather than with a user at the command line interface.

Here is what you'll see (note that some of the output lines have been folded for presentation here, indicated by “\”):

```
(16251) CmdLineClientMain.cxx: MPI_Init not called in \           ❶
      ccafe-single mode.
(16251) CmdLineClientMain.cxx: Try running with ccafe-single \
      --ccafe-mpi yes , or
(16251) CmdLineClientMain.cxx: try setenv CCAFE_USE_MPI 1 to force MPI_Init.
(16251) my rank: -1, my pid: 16251
my rank: -1, my pid: 16251
my rank: -1, my pid: 16251
my rank: -1, my pid: 16251Type: One Processor Interactive       ❶

CCAFFEINE configured with babel. ❷

cca>
CmdContextCCAMPI::initRC: No rc file found. Pallet may be empty. ❸
```

- ❶ Lines between these two markers give information about the status of MPI in the Ccaffeine framework, including the processes rank if MPI is initialized. As the messages indicate, **ccafe-single** is intended for single-process use and does not normally call `MPI_Init`, but if you're running parallel and having problems with the MPI environment, this is the first place to look for signs of trouble.
- ❷ This message confirms that this Ccaffeine executable was configured and built to work with Babel. This is a useful thing to check when you're using an unfamiliar installation of Ccaffeine, or the first time you Ccaffeine after building it yourself.
- ❸ It is common to use an “rc” file with Ccaffeine to help assemble and run the application. This is the place where Ccaffeine confirms that it loaded the rc file you intended (or in this case, it confirms that we *didn't* specify one). If there is an rc file, the Ccaffeine output from the commands it contains will follow this message, so there may be a lot more text between this message and the “cca>” prompt at which you can interact with Ccaffeine.



Note

We present Ccaffeine's output with “spew” disabled (the default). If Ccaffeine is configured and built with the `--enable-spew` option, you will see a *lot* of debugging output from Ccaffeine itself in addition to what we show here.

2. 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 are needed). When it starts up, Ccaffeine's path is empty, and it has no idea where to find components. Next you will set the path that points to the pre-built components:

```
path
pathBegin
pathEnd! empty path.

cca>path set /san/shared/cca/tutorial/src/components/lib
# There are allegedly 8 classes in the component path
```

```
cca>path
pathBegin
pathElement /san/shared/cca/tutorial/src/components/lib
pathEnd
```

Path-related commands in Ccaffeine include:

path append	Adds a directory to the end of the current path.
path init	Sets the path from the value of the <code>\$CCA_COMPONENT_PATH</code> environment variable.
path prepend	Adds a directory to the beginning of the current path.
path set	Sets the path to the value provided.



Tip

Typing **help** at the Ccaffeine `cca>` prompt will provide a complete list of the commands Ccaffeine's scripting language understands.

3. Ccaffeine also has the concept of a *palette* of components from which applications can be assembled. The **palette** command will show you what is currently in the palette, and the **repository get-global *class_name*** command is used to get the component of the specified class name from the repository (path) and load it into the palette:

```
cca>palette
Components available:

cca>repository get-global drivers.CXXDriver
Loaded drivers.CXXDriver NOW GLOBAL .

cca>repository get-global functions.PiFunction
Loaded functions.PiFunction NOW GLOBAL .

cca>repository get-global integrators.MonteCarlo
Loaded integrators.MonteCarlo NOW GLOBAL .

cca>repository get-global randomgens.RandNumGenerator
Loaded randomgens.RandNumGenerator NOW GLOBAL .

cca>palette
Components available:
drivers.CXXDriver
functions.PiFunction
integrators.MonteCarlo
randomgens.RandNumGenerator
```

4. Next, you need to instantiate the components you're going to use. The **instances** command will list all the component instances in Ccaffeine's work area, or *arena*. The command **instantiate *class_name instance_name*** will create an instance of the specified class from the palette with the specified instance name and call the new component instance's `setServices` method.

```
cca>instances
FRAMEWORK of type Ccaffeine-Support
```

```
cca>instantiate drivers.CXXDriver driversCXXDriver
driversCXXDriver of type drivers.CXXDriver
successfully instantiated

cca>instantiate functions.PiFunction functionsPiFunction
functionsPiFunction of type functions.PiFunction
successfully instantiated

cca>instantiate integrators.MonteCarlo integratorsMonteCarlo
integratorsMonteCarlo of type integrators.MonteCarlo
successfully instantiated

cca>instantiate randomgens.RandNumGenerator randomgensRandNumGenerator
randomgensRandNumGenerator of type randomgens.RandNumGenerator
successfully instantiated

cca>instances
FRAMEWORK of type Ccaffeine-Support
driversCXXDriver of type drivers.CXXDriver
functionsPiFunction of type functions.PiFunction
integratorsMonteCarlo of type integrators.MonteCarlo
randomgensRandNumGenerator of type randomgens.RandNumGenerator
```



Note

When you instantiate a component, you can name it whatever you like as long as it is unique with respect to all of the components that you've instantiated in your session with the framework. It is possible to instantiate the a given component class multiple times (with different names, of course).

- Once the components you need are instantiated, you need to connect up their ports appropriately. The **display chain** command will list the component instances in Ccaffeine's arena and any connections among their ports. To make a connection, you use the command **connect *user_instance_name user_port_name provider_instance_name provider_port_name*** (note that some of the input lines have been folded with “\” to fit on the page -- you'll have to rejoin them when you type in the commands because Ccaffeine doesn't understand continuation lines):

```
cca>display chain
Component FRAMEWORK of type Ccaffeine-Support ❶
Component driversCXXDriver of type drivers.CXXDriver
Component functionsPiFunction of type functions.PiFunction
Component integratorsMonteCarlo of type integrators.MonteCarlo
Component randomgensRandNumGenerator of type randomgens.RandNumGenerator

cca>connect driversCXXDriver IntegratorPort integratorsMonteCarlo \
IntegratorPort
driversCXXDriver))))IntegratorPort---->IntegratorPort((((integratorsMonteCarlo
connection made successfully ❷

cca>connect integratorsMonteCarlo FunctionPort functionsPiFunction \
FunctionPort
integratorsMonteCarlo))))FunctionPort---->FunctionPort((((functionsPiFunction
connection made successfully ❷

cca>connect integratorsMonteCarlo RandomGeneratorPort \
randomgensRandNumGenerator RandomGeneratorPort
```

```
integratorsMonteCarlo))))RandomGeneratorPort---->\
RandomGeneratorPort((((randomgensRandNumGenerator
connection made successfully
```

②

```
cca>display chain
```

③

```
Component FRAMEWORK of type Ccaffeine-Support
Component driversCXXDriver of type drivers.CXXDriver
  is using IntegratorPort connected to Port: IntegratorPort provided by \
  component integratorsMonteCarlo
Component functionsPiFunction of type functions.PiFunction
Component integratorsMonteCarlo of type integrators.MonteCarlo
  is using FunctionPort connected to Port: FunctionPort provided by \
  component functionsPiFunction
  is using RandomGeneratorPort connected to Port: RandomGeneratorPort \
  provided by component randomgensRandNumGenerator
Component randomgensRandNumGenerator of type randomgens.RandNumGenerator
```

- ① At this point, there are no connections, so the output of **display chain** looks very much like that of **instances** -- just a simple listing of the component instances in the arena.
- ② Characteristic of the output of a **connect** command is the ASCII “cartoon” illustrating the connection, with the *user* on the left and the *provider* on the right.
- ③ Now the output of **display chain** lists the connections associated with each component instance. Note that the connection information is printed with the *using* component instance only.



Note

Port names and port types are defined by the person who implements the component. They have to be unique within the component, but not across an entire application. In order to connect a *uses* port to a *provides* port, the *types* of the port must match, but the names need not match.



Tip

In the Ccaffeine framework, you can find out what ports a particular component *uses* and *provides* with the command **display component instance_name**:

```
cca>display component integratorsMonteCarlo
-----
Instance name: integratorsMonteCarlo
Class name: integrators.MonteCarlo
-----
UsesPorts registered for integratorsMonteCarlo

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

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

- 6. At this point, you have a fully-assembled application and are ready to run it!

While most CCA ports are defined by component developers, the CCA specification includes a special port named `GoPort`. The purpose of this port is have a way of kicking off the execution of a component. The command `go instance_name go_port_name` instructs the framework to invoke the specified go port:

```
cca>go driversCXXDriver GoPort
Value = 3.141768
##specific go command successful
```

and you can see a (fairly inaccurate) value for pi computed by Monte Carlo integration of the function $4/(1+x^2)$.

At this stage, you have successfully composed and run a CCA application based on existing components. In the remainder of this procedure, we'll see how it is possible to dynamically change the application you've assembled by disconnecting components and connecting others in their place. Or you can jump straight to Step 11 to (gracefully) end this session with Ccaffeine and move on to other procedures in this chapter, or on to other tasks altogether.

- At the moment, Ccaffeine's palette contains only the components we needed for the first application. Now, we'll add some more components to the palette and instantiate them in the arena:

```
cca>repository get-global integrators.Midpoint
Loaded integrators.Midpoint NOW GLOBAL .
```

```
cca>instantiate integrators.Midpoint integratorsMidpoint
integratorsMidpoint of type integrators.Midpoint
successfully instantiated
```

```
cca>repository get-global functions.CubeFunction
Loaded functions.CubeFunction NOW GLOBAL .
```

```
cca>instantiate functions.CubeFunction functionsCubeFunction
functionsCubeFunction of type functions.CubeFunction
successfully instantiated
```



Note

There is no harm in having components you don't use in the palette, or even having instances of them in the arena.

- In order to be able to swap out components for others, we first need to disconnect them. The **disconnect** command has the same syntax as the **connect** command, with both the *uses* and *provides* end points of the connection being specified.

Let's begin by changing the Monte Carlo integrator for another. The integrator is connected to both the driver and the function. (And also to the random number generator, but since we don't need it for anything else, there is no harm in leaving that connection intact.)

```
cca>disconnect driversCXXDriver IntegratorPort integratorsMonteCarlo \
IntegratorPort
driversCXXDriver))))IntegratorPort-\ \-IntegratorPort((((integratorsMonteCarlo
connection broken successfully
```

```
cca>disconnect integratorsMonteCarlo FunctionPort functionsPiFunction \
FunctionPort
integratorsMonteCarlo))))FunctionPort-\ \-FunctionPort((((functionsPiFunction
```

connection broken successfully

❶

❶ The **disconnect** command prints an ASCII cartoon of a broken connection, similar to that printed by the **connect** command.

9. Once we connect up a new integrator (in this case, using the mid-point rule algorithm) to the driver and function, we have a new “application” that’s ready to run:

```
cca>connect driversCXXDriver IntegratorPort integratorsMidpoint \
      IntegratorPort
driversCXXDriver)))IntegratorPort---->IntegratorPort((((integratorsMidpoint
connection made successfully
```

```
cca>connect integratorsMidpoint FunctionPort functionsPiFunction \
      FunctionPort
integratorsMidpoint)))FunctionPort---->FunctionPort((((functionsPiFunction
connection made successfully
```

```
cca>display chain
Component FRAMEWORK of type Ccaffeine-Support
Component driversCXXDriver of type drivers.CXXDriver
  is using IntegratorPort connected to Port: IntegratorPort provided by \
  component integratorsMidpoint
Component functionsCubeFunction of type functions.CubeFunction ❶
Component functionsPiFunction of type functions.PiFunction
Component integratorsMidpoint of type integrators.Midpoint
  is using FunctionPort connected to Port: FunctionPort provided by \
  component functionsPiFunction
Component integratorsMonteCarlo of type integrators.MonteCarlo ❶
  is using RandomGeneratorPort connected to Port: RandomGeneratorPort \
  provided by component randomgensRandNumGenerator
Component randomgensRandNumGenerator of type \
  randomgens.RandNumGenerator ❶
```

```
cca>go driversCXXDriver GoPort
Value = 3.141553
##specific go command successful
```

❶ Observe that there are a number of component instances in the arena that we have either never used (functionsCubeFunction) or which we have disconnected from the rest of the application (integratorsMonteCarlo and randomgensRandNumGenerator).

10. Finally, we swap the pi function for an x^3 function and run a third application built from the same set of components:

```
cca>disconnect integratorsMidpoint FunctionPort functionsPiFunction \
      FunctionPort
integratorsMidpoint)))FunctionPort-\ \-FunctionPort((((functionsPiFunction
connection broken successfully
```

```
cca>connect integratorsMidpoint FunctionPort functionsCube FunctionPort
integratorsMidpoint)))FunctionPort---->FunctionPort((((functionsCubeFunction
connection made successfully
```

```
cca>display chain
Component FRAMEWORK of type Ccaffeine-Support
Component driversCXXDriver of type drivers.CXXDriver
  is using IntegratorPort connected to Port: IntegratorPort provided by \
```

```

component integratorsMidpoint
Component functionsCubeFunction of type functions.CubeFunction
Component functionsPiFunction of type functions.PiFunction
Component integratorsMidpoint of type integrators.Midpoint
  is using FunctionPort connected to Port: FunctionPort provided by \
  component functionsCubeFunction
Component integratorsMonteCarlo of type integrators.MonteCarlo
  is using RandomGeneratorPort connected to Port: RandomGeneratorPort \
  provided by component randomgensRandNumGenerator
Component randomgensRandNumGenerator of type randomgens.RandNumGenerator

cca>go driversCXXDriver GoPort
Value = 0.250010
##specific go command successful

```

- To exit Ccaffeine “politely” and allow it to cleanly shutdown and destroy all components, use the **quit** command:

```

cca>quit

bye!
exit

```

2.2. Running Ccaffeine Using an rc File

In practice, most people don't use Ccaffeine interactively on a routine basis. Like many applications, Ccaffeine can be run with a script, or “rc” file that tells it what to do. Any commands that can be entered at the `cca>` prompt can be used in an rc file, so it is possible to systematically capture the assembly and execution of an application in a reusable form. The rc also makes it easy to create a new application from an existing one by adapting the script.

In this section, you will explore the use of an rc file that captures all of the commands performed in the previous section. This is the basic approach you will want to use when testing your work in the subsequent exercises.

- For this procedure, it is best to work in your home directory. To save you a lot of additional typing, we've created an rc file with all of the commands from the previous section. Make a local copy by typing `cp $CCA/src/components/examples/task0_rc .` and open it in your text editor. Here are some of the important features to note in this file:

```

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

# Step 2 ❷

path
path set /san/shared/cca/tutorial/src/components/lib
path

# Step 3 ❷

palette

```

```

repository get-global drivers.CXXDriver
repository get-global functions.PiFunction
repository get-global integrators.MonteCarlo
repository get-global randomgens.RandNumGenerator
palette

# Step 4

instances
instantiate drivers.CXXDriver driversCXXDriver
instantiate functions.PiFunction functionsPiFunction
instantiate integrators.MonteCarlo integratorsMonteCarlo
instantiate randomgens.RandNumGenerator randomgensRandNumGenerator
instances

# Step 5

display chain
connect driversCXXDriver IntegratorPort integratorsMonteCarlo IntegratorPort
connect integratorsMonteCarlo FunctionPort functionsPiFunction FunctionPort
connect integratorsMonteCarlo RandomGeneratorPort \
    randomgensRandNumGenerator RandomGeneratorPort
display chain
display component integratorsMonteCarlo

# Step 6

go driversCXXDriver GoPort

# Step 7

repository get-global integrators.Midpoint
instantiate integrators.Midpoint integratorsMidpoint
repository get-global functions.CubeFunction
instantiate functions.CubeFunction functionsCubeFunction

# Step 8

disconnect driversCXXDriver IntegratorPort integratorsMonteCarlo \
    IntegratorPort
disconnect integratorsMonteCarlo FunctionPort functionsPiFunction \
    FunctionPort

# Step 9

connect driversCXXDriver IntegratorPort integratorsMidpoint IntegratorPort
connect integratorsMidpoint FunctionPort functionsPiFunction FunctionPort
display chain
go driversCXXDriver GoPort

# Step 10

disconnect integratorsMidpoint FunctionPort functionsPiFunction FunctionPort
connect integratorsMidpoint FunctionPort functionsCube FunctionPort
display chain
go driversCXXDriver GoPort

# Step 11

quit 

```

- ❶ Ccaffeine requires this line exactly as written to recognize this file as an input script.
 - ❷ Ccaffeine interprets “#” as the beginning of a comment and ignores the remainder of the line. (Note that we have marked only the first few comments in this file.)
 - ❸ If your script does not contain a **quit** command, Ccaffeine will run the script and leave you at the Ccaffeine prompt, “cca>”, allowing you to interact with the framework manually. For example, you can use the `rc` file just to setup the palette; or you can use it to setup the palette and instantiate the components you need in the arena; or you can use it to assemble the entire application, but type the **go** command yourself.
2. Enter the command `ccafe-single --ccafe-rc task0_rc >& task0p1.out` (assuming you're using the `cs`h or `tc`sh shells; if you're using the `sh` or `ba`sh shells, the command is `ccafe-single --ccafe-rc task0_rc > task0.out 2>&1`)

Edit the `task0.out` file and compare the results with those in the prior section. Everything should be essentially the same.

3. Experiment with changing `task0_rc` and re-running Step 2. Take a careful look at the output to make sure each change worked as you expected.

Some suggestions for things to change:

- Rearrange some of the commands so that all of the **repository get-global** commands are at the beginning of the file; you could also group all of the instantiations together. Done properly, this should have no effect your ability to execute the applications.
- Since the original script assembles and runs three distinct applications, you might modify the script so that it does only one by commenting out the lines that aren't needed.
- Make use of the `drivers.F90Driver`, which has not been used at all so far. (This means you will have to add **repository get-global** and **instantiate** commands for it.)



Tip

You can copy the original `task0_rc` to other filenames if you want to preserve the different variations you try. If you're just eliminating lines (for example to run only a single application), it may be convenient to just comment them out instead of actually removing them.



Warning

If you remove the **quit** command from the `rc` file, Ccaffeine will leave you in interactive mode rather than terminating and returning you to the shell prompt. In this case, you should *not* capture Ccaffeine's output into a file, as instructed in Step 2 because you won't be able to see the `cca>` prompt and it will *appear* that Ccaffeine has hung (in reality it is just waiting for your input). If you make this mistake a **Control-c** will interrupt Ccaffeine and return you to the shell prompt.

2.3. Using the GUI Front-End to Ccaffeine

Not yet documented.

Chapter 3. Sewing CCA Components into an Application: the Driver Component

\$Revision: 1.31 \$

\$Date: 2004/08/26 15:17:54 \$

In this exercise, you will create a new Driver component. This component is very simple, and basically only *uses* other components (it also *provides* a GoPort). If you're working in an environment in which components are already available that do *most* of what you need, it is often sufficient to create a component, which we refer to generically as a *driver*, that orchestrates these existing components to perform your computation.

Unlike other component models (e.g. Cactus [<http://citeseer.nj.nec.com/allen00cactus.html>] or ESMF [http://sdcd.gsfc.nasa.gov/ESS/esmf_tasc/]) CCA does not impose a built-in execution model. CCA allows the user to determine how the components are to be used. The driver component, in essence, takes the place of the main program in a normal application.

In this section we will walk through the construction of a driver component, either in Fortran (SIDL name `drivers.F90Driver`) or C++ (SIDL name `drivers.CXXDriver`) Regardless of language, our driver component will *use* an `integrator.IntegratorPort`. It will also *provide* a `gov.cca.ports.GoPort` that allows an outside entity (a user or script) to start execution of the component. (These ports should be familiar from Chapter 2, *Assembling and Running a CCA Application*.)

3.1. The SIDL Definition of the Driver Component

The first step in creating a new component is to create its `.sidl` file. In SIDL, a component is a class that implements several SIDL interfaces. All CCA components must implement the `gov.cca.Component` interface, which is defined as part of the CCA specification (the CCA specification uses the `gov.cca` namespace). In addition, components must implement the interfaces corresponding to any CCA ports they wish to *provide*. The CCA specification defines a few ports, such as `gov.cca.ports.GoPort`, but mostly, ports are defined by the people who write components, or by communities that get together to agree on a “standard” interface.

In order to better understand what is required to implement a given interface, you need to find the SIDL specification for it. First, we'll look in the SIDL file for the CCA specification to see what the `gov.cca.Component` interface looks like.

1. Edit `$CCA/share/cca-spec-babel-0_7_0-babel-0.9.4/cca.sidl`. First, notice the package declarations at the beginning of the file:

```
package gov {
package cca version 0.7.0 {
...

```

which declare the `gov.cca` namespace for everything in the file.

2. Now, search for “interface Component”:

```

...
/**
 * All components must implement this interface.
 */
interface Component {
    ... Comments elided ...
    void setServices(in Services services) throws CCAException;
}
...

```

Which tells us that our driver will have to implement a `setServices`. This is the key method that allows a piece of code to become a CCA component. The component's `setServices` method is invoked by the CCA framework when the component is instantiated, and advertises to the framework the ports the component will *provide* and *use*.

3. Since the port this component *provides* is also part of the CCA specification, this is the place to look for the definition of the `GoPort`. Search for “interface `GoPort`”:

```

...
package ports {

    /**
     * Go, component, go!
     */
    interface GoPort extends Port {
        ... Comments elided ...
        int go();
    }

}
...

```

First, notice that there is an additional package declaration here, making the full name of this interface `gov.cca.ports.GoPort`. This definition tells us that our driver component must also implement a `go` method.

4. Now you have enough information to write the SIDL declaration for your driver component. At this point, you should choose whether you want to implement your driver component in C++ or Fortran 90. (Once you get one done, you can implement the other too, if you wish.)

Edit the file `student-src/components/sidl/drivers.sidl` and type in one of the two following SIDL declarations, according to your choice of language:

a.

```

package drivers version 1.0 {
    class F90Driver implements gov.cca.ports.GoPort,
                                gov.cca.Component
    {
        int go();
        void setServices(in gov.cca.Services services)
                        throws gov.cca.CCAException;
    }
}

```

b.

```

package drivers version 1.0 {

```

```

class CXXDriver implements gov.cca.ports.GoPort,
                           gov.cca.Component
{
    int go();
    void setServices(in gov.cca.Services services)
                    throws gov.cca.CCAException;
}

```

First, notice that the two declarations are identical except for the name, and in reality, you could choose anything you wanted for the name. The only reason we put an indication of the implementation language into the class name of this component was pedagogical: to avoid a name collision if you want to eventually implement both versions, and identify what distinguishes them. Normally, you might want different implementations of a component if they do things differently (i.e. use different algorithms), or in the case of a driver, solve different problems. Under normal circumstances, there is no reason to have more than one implementation of a component that does precisely the same thing.

Second, notice that the class definition references both `gov.cca.ports.GoPort` and `gov.cca.Component`, and declares all of the methods that we saw in those interface definitions, with precisely the same signatures.

5. Now you need to modify the Makefile system so that it is aware of the new `drivers.sidl` file and the component you're adding.

Edit `student-src/component/MakeIncl.components` and make the following additions:

```

# SIDL files containing component declarations
# For example:
# SIDL_FILES = sidl/drivers.sidl
SIDL_FILES = sidl/functions.sidl sidl/integrators.sidl sidl/randomgens.sidl \
             sidl/drivers.sidl

# The COMPONENTS list contains the fully-qualified names of the component
# classes, augmented with -LANGUAGE, where LANGUAGE is the language
# in which the component is implemented, e.g., c, c++, f90.
# For example:
# COMPONENTS = drivers.F90Driver-f90 drivers.CXXDriver-c++
COMPONENTS = functions.PiFunction-c++ \
             integrators.MonteCarlo-f90 randomgens.RandNumGenerator-c++ \
             drivers.CXXDriver-c++

```

Of course if you've chose to create the Fortran 90 driver, you should add **drivers.F90Driver-f90** to the definition of COMPONENTS instead. In both cases, notice the backslash (“\”) used to continue definition on to the next line. **make** will accept long lines, but the files are easier to read if they're nicely formatted.

6. In the `student-src/components` directory, type **make .repository** to make Babel process the `.sidl` files and update the XML repository. The output should look something like this:

```

touch .sidl

### Generating XML for SIDL packages containing component declarations
/san/shared/cca/tutorial/bin/babel -t xml -R../xml_repository \

```

```

-R/san/shared/cca/tutorial/share/cca-spec-babel-0_7_0-babel-0.9.4/xml \
-o ../xml_repository sidl/functions.sidl sidl/integrators.sidl \
sidl/randomgens.sidl sidl/drivers.sidl
Babel: Parsing URL "file:./automount/whale/root/san/r1a010/bernhold/\
student-src/components/sidl/functions.sidl"...
Babel: Warning: Symbol exists in XML repository: \
functions.LinearFunction-v1.0
Babel: Warning: Symbol exists in XML repository: \
functions.NonlinearFunction-v1.0
Babel: Warning: Symbol exists in XML repository: \
functions.PiFunction-v1.0
Babel: Parsing URL "file:./automount/whale/root/san/r1a010/bernhold/\
student-src/components/sidl/integrators.sidl"...
Babel: Warning: Symbol exists in XML repository: \
integrators.MonteCarlo-v1.0
Babel: Parsing URL "file:./automount/whale/root/san/r1a010/bernhold/\
student-src/components/sidl/randomgens.sidl"...
Babel: Warning: Symbol exists in XML repository: \
randomgens.RandNumGenerator-v1.0
Babel: Parsing URL "file:./automount/whale/root/san/r1a010/bernhold/\
student-src/components/sidl/drivers.sidl"...
touch .repository

```

The next step is to implement the internals of the component, which are obviously dependent on the implementation language you've chosen. For C++, continue directly on with Section 3.2, “Implementation of the CXXDriver in C++”. For Fortran 90, please jump to Section 3.3, “Implementation of the F90Driver in Fortran 90”.

3.2. Implementation of the CXXDriver in C++

1. The next step is to get Babel to generate the skeleton code that we will fill in with the component's implementation. In the `student-src/components` directory, type **make .drivers.CXXDriver-c++**. The output should look something like this:

```

### Generating a c++ implementation for the drivers.CXXDriver component.
/san/shared/cca/tutorial/bin/babel -s c++ -R../xml_repository \
-R/san/shared/cca/tutorial/share/cca-spec-babel-0_7_0-babel-0.9.4/xml \
-g -u -E -l -m drivers.CXXDriver. --suppress-timestamp drivers.CXXDriver
Babel: Resolved symbol "drivers.CXXDriver"...
touch .drivers.CXXDriver-c++

```

and in the `student-src/components/drivers/c++` directory, you should see the following files:

```

drivers.CXXDriver.babel.make
drivers_CXXDriver_Impl.cc
drivers_CXXDriver_Impl.hh
glue

```

all of which were generated by Babel. (`glue` is actually a directory that contains a large number of generated files that Babel needs to do its job, but which you never need to modify.) The source code files that you will need to modify in order to implement the component are the so-called Im-

pl files. For C++, both a source file (.cc) and the corresponding header file (.hh) are generated.

2. In your editor, take a look through both `student-src/components/drivers/c++/drivers_CXXDriver_Impl.cc` and `student-src/components/drivers/c++/drivers_CXXDriver_Impl.hh` to familiarize yourself with their structure before you make any changes.
 - a. Near the top of `drivers_CXXDriver_Impl.hh`, you will see a group of include directives:

```
...
//
// Includes for all method dependencies.
//
#ifdef included_drivers_CXXDriver_hh
#include "drivers_CXXDriver.hh"
#endif
...
```

Babel generates include directives for header files that are necessary to resolve the types used in the SIDL definition of the class you're implementing (in this case, in the `student-src/components/sidl/drivers.sidl` file). It does not automatically generate include directives for interfaces you implement. You will have to add those and any other header files your implementation requires as part of the implementation process.

When an automatically generated file is manually modified, there is always a danger that the modifications will be overwritten the next time the file is generated. Babel solves this with a concept called *splicer blocks*. These structured comments that appear to the compiler as regular comments, but are interpreted by Babel as having a special meaning. Babel will preserve code *within* a splicer block when the file is regenerated. Code outside splicer blocks will be overwritten. Most Babel-generated files contain numerous splicer blocks -- everywhere you might need to add something to the generated skeleton. Here is an example:

```
...
// DO-NOT-DELETE splicer.begin(drivers.CXXDriver._includes)
// Put additional includes or other arbitrary code here...
// DO-NOT-DELETE splicer.end(drivers.CXXDriver._includes)
...
```

Note that each splicer block has a name that is unique within the file, and has explicit beginning and end markers. In this case, the leading comment syntax is appropriate to C++, but of course files generated for other languages will have different ways of denoting comments.

- b. In the `drivers_CXXDriver_Impl.cc`, You will see that Babel has already generated the signatures for all of the methods you need to implement, giving them appropriate C++-ized names, and has provided splicer blocks ready for you to fill in. This includes both the `go` method inherited from the `gov.cca.ports.GoPort` definition, and the `setServices` method inherited from the `gov.cca.Component` definition.

3.2.1. The `setServices` Implementation

1. We'll begin by implementing the `setServices` method in `drivers_CXXDriver_Impl.cc`. Here is what the routine should look like (you'll need to type in the stuff marked up **like this**), along with some comments about different sections.

```

...
/**
 * Method:  setServices[]
 */
void
drivers::CXXDriver_impl::setServices (
    /*in*/ ::gov::cca::Services services )
throw (
    ::gov::cca::CCAException
){
    // DO-NOT-DELETE splicer.begin(drivers.CXXDriver.setServices)
    // insert implementation here

    frameworkServices = services; ❶

    // Provide a Go port
    gov::cca::ports::GoPort gp = self; ❷

    frameworkServices.addProvidesPort(gp, ❸
                                     "GoPort",
                                     "gov.cca.ports.GoPort",
                                     frameworkServices.createTypeMap());

    // Use an IntegratorPort port
    frameworkServices.registerUsesPort ("IntegratorPort", ❹
                                       "integrator.IntegratorPort",
                                       frameworkServices.createTypeMap());

    // DO-NOT-DELETE splicer.end(drivers.CXXDriver.setServices)
}
...

```

- ❶ When the framework calls `setServices`, it passes in a `gov.cca.Services` object (in C++ `gov::cca::Services`) that we need to keep a copy of. Note that `frameworkServices` is not declared here. We will add a declaration for it to the `.hh` file in the next step.
- ❷ In order to register the ports that our component will provide with the framework, we use the `addProvidesPort` method of the `gov.cca.Services` interface. You can find this interface in the `cca.sidl` file (where you previously looked up `gov.cca.Component` and `gov.cca.ports.GoPort`) in order to check its signature, which is:

```

...
void addProvidesPort(in gov.cca.Port inPort,
                    in string portName,
                    in string type,
                    in gov.cca.TypeMap properties )
    throws gov.cca.CCAException ;
...

```

(Of course we're actually calling the C++ version of the interface.)

The first argument is the object that actually provides the port. The way we wrote the SIDL, the `drivers.CXXDriver` class provides the port, and since we're writing a method within this class, C++ allows the enclosing object to be referred to as `self` (cast to the appropriate

type).

The second and third arguments are a local name for the port, which must be unique within the component, and a type, which *should* be globally unique. If the actual types of the ports don't match between *user* and *provider*, it will cause a failed cast or possibly a segmentation fault. The string type here is a convenience to the user, giving a human-readable way to identify the type of the port that can be presented in the framework's user interface. By convention, the SIDL interface name for the port is used for the type.

The final argument is a `gov.cca.TypeMap`. This is a CCA-defined type that provides a simple hash table that can be used to associate properties with a *provides* port. In practice, it is rarely used, but must be present.

- ④ We must also tell the framework which ports we expect to *use* from other components. Looking in `cca.sidl`, we find that the method's signature is:

```
...
void registerUsesPort(in string portName,
                    in string type,
                    in gov.cca.TypeMap properties )
    throws gov.cca.CCAException ;
...
```

The first and second arguments are a local name for the port, following the same rules and conventions as in the `addProvidesPort` invocation above. The final argument is, once again, a `gov.cca.TypeMap`, again like `addProvidesPort`.

2. The header file also requires a couple of additions. First, let's take care of declaring `frameworkServices` as a private variable belonging to the `drivers::CXXDriver` class.

Edit `student-src/components/drivers/c++/drivers_CXXDriver_Impl.hh` and add the following:

```
...
/**
 * Symbol "drivers.CXXDriver" (version 1.0)
 */
class CXXDriver_impl
// DO-NOT-DELETE splicer.begin(drivers.CXXDriver._inherits)
// Put additional inheritance here...
// DO-NOT-DELETE splicer.end(drivers.CXXDriver._inherits)
{

private:
    // Pointer back to IOR.
    // Use this to dispatch back through IOR vtable.
    CXXDriver self;

    // DO-NOT-DELETE splicer.begin(drivers.CXXDriver._implementation)
    // Put additional implementation details here...

    ::gov::cca::Services frameworkServices;

    // DO-NOT-DELETE splicer.end(drivers.CXXDriver._implementation)
...

```

3. We also need to add the include directives for the header files for the classes we inherit from. (For technical reasons, Babel does not insert these automatically when it generates the file.)

```

...
// DO-NOT-DELETE splicer.begin(drivers.CXXDriver._includes)
// Put additional includes or other arbitrary code here...

#include "integrator_IntegratorPort.hh"
#include "gov_cca_ports_GoPort.hh"

// DO-NOT-DELETE splicer.end(drivers.CXXDriver._includes)
...

```

Note that in naming files, Babel translates periods (“.”) in the SIDL to underscores (“_”).

4. Now, although the component is not complete, it is a good idea to check that it compiles correctly with the code you've added so far.

First, change directories to `student-src/components` and run **make drivers**. This will install `Makefile` and `MakeIncl.user` files in `student-src/components/drivers/c++`.

Then, change directories to `student-src/components/drivers/c++` and run **make**. If you get any compiler errors, you should fix them before going on.

3.2.2. The go Implementation

1. Once again, edit `student-src/components/drivers/c++/drivers_CXXDriver_Impl.cc` and add the implementation of the go method:

```

...
/**
 * Method: go[]
 */
int32_t
drivers::CXXDriver_impl::go ()
throw ()

{
  // DO-NOT-DELETE splicer.begin(drivers.CXXDriver.go)
  // insert implementation here

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

  ::integrator::IntegratorPort integrator; ❷

  // get the port ...
  integrator = frameworkServices.getPort("IntegratorPort"); ❸

  if(integrator._is_nil()) { ❹
    fprintf(stdout, "drivers.CXXDriver not connected\n");
    frameworkServices.releasePort("IntegratorPort");
  }
}

```

```

    return -1;
}
// operate on the port
value = integrator.integrate (lowerBound, upperBound, count);
④

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

// release the port.
frameworkServices.releasePort ("IntegratorPort"); ⑤
return 0; ⑥

// DO-NOT-DELETE splicer.end(drivers.CXXDriver.go)
}
...

```

- ① Setup the parameters with which to call the integrator.
- ② In this section we get a handle to the particular `IntegratorPort` that the driver's `uses` port has been connected to. First, we have to declare a variable of the appropriate type (`::integrator::IntegratorPort` is the C++ translation of the SIDL `integrator.IntegratorPort`, defined in `student-src/ports/sidl/integrator.sidl`). Then, we invoke the `getPort` on our `frameworkServices` object. The argument to this method is the local name we used in the `registerUsesPort` invocation.
- ③ This code checks that the `getPort` worked, and returned a valid port. If the `getPort` fails, or if the driver's `uses` port has not been connected to an appropriate *provider*, then `getPort` will return a nil port object. The `_is_nil` method is automatically available on all SIDL objects. Because the driver can't do anything without being properly connected to an integrator, the response to `getPort` failing is to abort by returning a non-zero value.



Note

`getPort` returning nil need not be treated as a fatal error in all cases. For example, a component may be designed so that certain ports are optional -- to be used if present, but to be ignored if not. Another possibility is that the component may be able to accomplish the same thing through several different ports, so that only one of a given group needs to be connected.

- ④ Here we actually call the `integrate` method on the `integrator` port we just got a handle for. The signature of the `integrate` method is defined in `student-src/ports/sidl/integrator.sidl`.
 - ⑤ Finally, once we're done using the port, we call `releasePort`.
 - ⑥ It is considered impolite for a component to call `exit` because it will bring down the entire application, and possibly crash the framework. Instead, components should simply return.
2. Congratulations, you have completed the implementation of the `CXXDriver`! To check your work, run **make** in `student-src/components/drivers/c++`. If you get any compiler errors, you should fix them before going on.
 3. At this point, it is a good idea to go up to `student-src` and run **make** to insure that anything else which might depend on the existence of the new `drivers.CXXDriver` component gets built too.

The next step is to test your new driver component, in Section 3.5, "Using Your New Component".

3.3. Implementation of the F90Driver in Fortran 90

Before we begin the implementation, it is important to understand that, regardless of language, both the CCA and especially Babel/SIDL impose an object-oriented model on any of its supported languages, including Fortran. Most importantly, this means that each Fortran component has *state* and *methods*. State means that variables are associated with a particular instance component and that these *state* variables (sometimes referred to as private data) can take on different values for different instances. A *method* is a subroutine that is associated with the component. A short introduction to the way CCA/Babel deal with imposing an object model on Fortran is given in Section 3.4, “SIDL and CCA Object Orientation in Fortran” and can be read at your leisure. You should also read the Fortran 90 section of the Babel Users' Guide [http://www.llnl.gov/CASC/components/docs/users_guide/users_guide.html].

There are other limitations of the Fortran 90 standard that Babel deals with by adhering to certain conventions:

- Fortran doesn't offer the hierarchical structures for routine and type names in the way that most OO languages do, so SIDL's hierarchical dot-separated notation is translated into a flat namespace using underscores in Fortran. For example, `gov.cca.Services` is translated to `gov_cca_Services`. A reference to that SIDL interface would be defined as a variable in this fashion:

```
type(gov_cca_Services_t) :: services
```

- Because of the requirement that all symbols in Fortran 90 be at most 32 characters, sometimes long names common in OO programming styles need to be abbreviated. Babel keeps the most significant portion of the name (the base name) and truncates the rest, adding a hash to make it unique if necessary. For example, our own F90Driver component's `setServices()` subroutine declaration looks like:

```
recursive subroutine F90Dri_setServices4khxt4z7ds_mi(self, services, &
                                                    exception)
```

1. The next step in implementing the driver is to get Babel to generate the skeleton code that we will fill in with the component's implementation. In the `student-src/components` directory, type **make .drivers.F90Driver-f90**. The output should look something like this:

```
### Generating a f90 implementation for the drivers.F90Driver component.
/san/shared/cca/tutorial/bin/babel -s f90 -R../xml_repository \
  -R/san/shared/cca/tutorial/share/cca-spec-babel-0_7_0-babel-0.9.4/xml \
  -g -u -E -l -m drivers.F90Driver. --suppress-timestamp drivers.F90Driver
Babel: Resolved symbol "drivers.F90Driver"...
touch .drivers.F90Driver-f90
```

and in the `student-src/components/drivers/f90` directory, you should see the following files:

```
drivers.F90Driver.babel.make
drivers_F90Driver_Impl.F90
drivers_F90Driver_Mod.F90
```

glue

all of which were generated by Babel. (glue is actually a directory that contains a large number of generated files that Babel needs to do its job, but which you never need to modify.) The source code files that you will need to modify in order to implement the component are the so-called `Impl` files. For Fortran 90, both a source file (`_Impl.F90`) and the corresponding module file (`_Mod.F90`) are generated.

2. In your editor, take a look through both `student-src/components/drivers/f90/drivers_F90Driver_Impl.F90` and `student-src/components/drivers/c++/drivers_F90Driver_Mod.F90` to familiarize yourself with their structure before you make any changes.
 - a. When an automatically generated file is manually modified, there is always a danger that the modifications will be overwritten the next time the file is generated. Babel solves this with a concept called *splicer blocks*. These structured comments that appear to the compiler as regular comments, but are interpreted by Babel as having a special meaning. Babel will preserve code *within* a splicer block when the file is regenerated. Code outside splicer blocks will be overwritten. Most Babel-generated files contain numerous splicer blocks -- everywhere you might need to add something to the generated skeleton. Here is an example:

```
...
! DO-NOT-DELETE splicer.begin(drivers.F90Driver.use)
! Insert use statements here...
! DO-NOT-DELETE splicer.end(drivers.F90Driver.use)
...
```

Note that each splicer block has a name that is unique within the file, and has explicit beginning and end markers. In this case, the leading comment syntax is appropriate to Fortran 90, but of course files generated for other languages will have different ways of denoting comments.

- b. In the `drivers_F90Driver_Impl.F90`, You will see that Babel has already generated the signatures for all of the methods you need to implement, giving them appropriate names that conform to the Fortran 90 standard (including being hashed to remain within the 32 character limit if necessary), however it should be fairly easy to match them up with corresponding SIDL names. In this case, both the `go` method inherited from the `gov.cca.ports.GoPort` definition, and the `setServices` method inherited from the `gov.cca.Component` definition are there, along with several others associated with Babel.

3.3.1. The `setServices` Implementation

1. We'll begin by implementing the `setServices` method in `drivers_F90Driver_Impl.F90`. Here is what the routine should look like (you'll need to type in the stuff marked up **like this**), along with some comments about different sections.

```
...
!
! Method:  setServices[]
```

```

!
recursive subroutine F90Dri_setServices4khxt4z7ds_mi(self, services, &
    exception)
    use sidl_BaseInterface
    use drivers_F90Driver
    use gov_cca_Services
    use gov_cca_CCAException
    use drivers_F90Driver_impl
    ! DO-NOT-DELETE splicer.begin(drivers.F90Driver.setServices.use)
    ! Insert use statements here...

    use gov_cca_TypeMap      ! A CCA catch-all properties list (empty for us)
    use gov_cca_Port        ! needed to use a gov.cca.Port (we do)
    use gov_cca_ports_GoPort ! need to export our implementation of GoPort

    ! DO-NOT-DELETE splicer.end(drivers.F90Driver.setServices.use)
    implicit none
    type(drivers_F90Driver_t) :: self ! in
    type(gov_cca_Services_t) :: services ! in
    type(sidl_BaseInterface_t) :: exception ! out

! DO-NOT-DELETE splicer.begin(drivers.F90Driver.setServices)
! Insert the implementation here...

    type(gov_cca_TypeMap_t)      :: myTypeMap ❶
    type(gov_cca_Port_t)        :: myPort
    type(SIDL_BaseInterface_t) :: excpt
    type(drivers_F90Driver_wrap) :: dp

    call drivers_F90Driver__get_data_m(self, dp) ❷

    ! Set my reference to the services handle
    dp%d_private_data%frameworkServices = services ❸

    call addRef(services)

    ! Create an empty TypeMap
    call createTypeMap(dp%d_private_data%frameworkServices, & ❹
        myTypeMap, excpt)
    call checkExceptionDriver(excpt, 'setServices createTypeMap call')

    ! Provide a GoPort
    call cast(self, myPort) ❷

    call addProvidesPort(dp%d_private_data%frameworkServices, & ❷
        myPort, 'GoPort', 'gov.cca.GoPort', &
        myTypeMap, excpt)
    call checkExceptionDriver(excpt, 'setServices addProvidesPort: GoPort' )

    ! Register to use an integrator port
    call registerUsesPort(dp%d_private_data%frameworkServices, & ❺
        'IntegratorPort', &
        'integrator.Integrator', &
        myTypeMap, excpt)
    call checkExceptionDriver(excpt, &
        'setServices registerUsesPort: IntegratorPort')

    call deleteRef(myTypeMap) ❹

! DO-NOT-DELETE splicer.end(drivers.F90Driver.setServices)

```

```
end subroutine F90Dri_setServices4khxt4z7ds_mi
...
```

- ❶ Declaration of variables that will be needed below. The types are defined in various modules used above. The `drivers_F90Driver_wrap` type is a Babel idiom for the private data associated with the particular *instance* of this component, in an object-oriented sense.
- ❷ When the framework calls `setServices`, it passes in a `gov.cca.Services` object (in C++ `gov::cca::Services`) that we need to keep a copy of in the private data associated with this instance of our component. Babel uses “reference counting” to track usage of objects in order to know when it is safe to delete them. Because Fortran has no native mechanism for reference counting, we must use Babel's `addRef` method to indicate that we're storing a reference to the `services` object that the framework passed in to `setServices`
- ❸ The `services` methods to register *uses* and *provides* ports requires a `gov.cca.TypeMap` (in Fortran `TypeMap`), which we create here.

In SIDL, methods can throw exceptions. In languages like Fortran, which don't have native support for exceptions (if you're not familiar with exceptions, it is sufficient to think of them as error codes), they are translated into an additional subroutine argument (in this case `excpt`) which then should be checked (“caught”). We'll add the `checkException-Driver` method in Step 2.

When Babel creates `myTypeMap`, it will (internally) add a reference to it. Once we're done using it, we can tell Babel that by calling Babel's `deleteRef` method, which you can see at the end of the routine. When the reference count goes to zero, Babel will destroy the `myTypeRef` object and reclaim the memory associated with it.



Caution

Failure to follow proper reference counting procedures in Babel/Fortran (or other non-OO languages, such as C) code will lead to “memory leaks” in your application. See the Babel Users' Guide [http://www.llnl.gov/CASC/components/docs/users_guide/users_guide.html] for more detailed information.

- ❷ In order to register the ports that our component will provide with the framework, we use the `addProvidesPort` method of the `gov.cca.Services` interface. You can find this interface in the `cca.sidl` file (where you previously looked up `gov.cca.Component` and `gov.cca.ports.GoPort`) in order to check its signature, which is:

```
...
void addProvidesPort(in gov.cca.Port inPort,
                    in string portName,
                    in string type,
                    in gov.cca.TypeMap properties )
    throws gov.cca.CCAException ;
...
```

(Of course we're actually calling the Fortran 90 version of the interface.)

The first argument is the object that actually provides the port. The way we wrote the SIDL, the `drivers.F90Driver` class provides the port, and since we're writing a method within this class, we use Babel's `cast` method to cast our self pointer to `type gov.cca.Port`.

The second and third arguments are a local name for the port, which must be unique within the component, and a type, which *should* be globally unique. If the actual types of the ports don't match between *user* and *provider*, it will cause a failed cast or possibly a segmentation

fault. The string type here is a convenience to the user, giving a human-readable way to identify the type of the port that can be presented in the framework's user interface. By convention, the SIDL interface name for the port is used for the type.

The final argument is a `gov.cca.TypeMap`. This is a CCA-defined type that provides a simple hash table that can be used to associate properties with a *provides* port. In practice, it is rarely used, but must be present.

- ⑤ We must also tell the framework which ports we expect to *use* from other components. Looking in `cca.sidl`, we find that the method's signature is:

```
...
void registerUsesPort(in string portName,
                     in string type,
                     in gov.cca.TypeMap properties )
    throws gov.cca.CCAException ;
...
```

The first and second arguments are a local name for the port, following the same rules and conventions as in the `addProvidesPort` invocation above. The final argument is, once again, a `gov.cca.TypeMap`, again like `addProvidesPort`.

2. The module file also requires a couple of additions. First, let's take care of declaring `frameworkServices` as part of the module's private data.

Edit `student-src/components/drivers/f90/drivers_F90Driver_Mod.F90` and add the following:

```
...
type drivers_F90Driver_priv
    sequence
    ! DO-NOT-DELETE splicer.begin(drivers.F90Driver.private_data)

    ! Handle to framework Services object
    type(gov_cca_Services_t) :: frameworkServices

    ! DO-NOT-DELETE splicer.end(drivers.F90Driver.private_data)
end type drivers_F90Driver_priv
...
```

3. We also need to add the use directives for the module for `gov.cca.Services`.

```
...
! DO-NOT-DELETE splicer.begin(drivers.F90Driver.use)
! Insert use statements here...

! CCA framework services module
use gov_cca_Services

! DO-NOT-DELETE splicer.end(drivers.F90Driver.use)
...
```

4. Now, although the component is not complete, it is a good idea to check that it compiles correctly with the code you've added so far.

First, change directories to `student-src/components` and run **make drivers**. This will

```
install Makefile and MakeIncl.user files in student-  
src/components/drivers/f90.
```

Then, change directories to `student-src/components/drivers/f90` and run **make**. If you get any compiler errors, you should fix them before going on.

3.3.2. Implementing the Constructor and Destructor

Constructor and *destructor* are concepts from object-oriented programming. Specifically, they are the routines that are called to create an instance of an object, and when it is being destroyed. When using most OO languages in the CCA/Babel environment, the constructor and destructor are handled pretty much automatically. In a non-OO language, like Fortran or C, we have to do a little more work. Specifically, we have to allocate and deallocate the data needed to maintain the private state of the component instance.

1. Edit `student-src/components/drivers/f90/drivers_F90Driver_Impl.F90` and find the constructor method, which Babel abbreviates `ctor`.

The constructor must allocate the space for the private data, initialize the private data as appropriate (in this case, we set `frameworkServices` to `null`), and Babel has to be told about the private data. In this component, the only private data we need to store is a pointer to the `services` object passed into `setServices`.

```
...  
!  
! Class constructor called when the class is created.  
!  
  
recursive subroutine drivers_F90Driver__ctor_mi(self)  
  use drivers_F90Driver  
  use drivers_F90Driver_impl  
  ! DO-NOT-DELETE splicer.begin(drivers.F90Driver._ctor.use)  
  ! Insert use statements here...  
  ! DO-NOT-DELETE splicer.end(drivers.F90Driver._ctor.use)  
  implicit none  
  type(drivers_F90Driver_t) :: self ! in  
  
  ! DO-NOT-DELETE splicer.begin(drivers.F90Driver._ctor)  
  ! Insert the implementation here...  
  
  ! Access private data  
  type(drivers_F90Driver_wrap) :: dp  
  ! Allocate memory and initialize  
  allocate(dp%d_private_data)  
  call set_null(dp%d_private_data%frameworkServices)  
  call drivers_F90Driver__set_data_m(self, dp)  
  
  ! DO-NOT-DELETE splicer.end(drivers.F90Driver._ctor)  
end subroutine drivers_F90Driver__ctor_mi  
...
```

2. Find the destructor method, which Babel abbreviates `dtor`. The destructor's job is to undo what the constructor did.

```

...
!
! Class destructor called when the class is deleted.
!

recursive subroutine drivers_F90Driver__dtor_mi(self)
  use drivers_F90Driver
  use drivers_F90Driver_impl
  ! DO-NOT-DELETE splicer.begin(drivers.F90Driver.__dtor.use)
  ! Insert use statements here...
  ! DO-NOT-DELETE splicer.end(drivers.F90Driver.__dtor.use)
  implicit none
  type(drivers_F90Driver_t) :: self ! in

  ! DO-NOT-DELETE splicer.begin(drivers.F90Driver.__dtor)
  ! Insert the implementation here...

  ! Access private data and deallocate storage
  type(drivers_F90Driver_wrap) :: dp
  call drivers_F90Driver__get_data_m(self, dp)
  deallocate(dp%d_private_data)

  ! DO-NOT-DELETE splicer.end(drivers.F90Driver.__dtor)
end subroutine drivers_F90Driver__dtor_mi
...

```

3. Now, although the component is not complete, it is a good idea to check that it compiles correctly with the code you've added so far. Run **make** in `student-src/components/drivers/f90`. If you get any compiler errors, you should fix them before going on.

3.3.3. The go Implementation

1. Once again, edit `student-src/components/drivers/f90/drivers_F90Driver_Impl.F90` and add the implementation of the go method:

```

...
!
! Method: go[]
!

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

  use sidl_BaseInterface ❶
  use gov_cca_Services
  use gov_cca_Port
  use integrator_IntegratorPort

```

```

! DO-NOT-DELETE splicer.end(drivers.F90Driver.go.use)
implicit none
type(drivers_F90Driver_t) :: self ! in
integer (selected_int_kind(9)) :: retval ! out

! DO-NOT-DELETE splicer.begin(drivers.F90Driver.go)
! Insert the implementation here...

type(gov_cca_Port_t) :: generalPort ②
type(SIDL_BaseInterface_t) :: excpt
type(integrator_IntegratorPort_t) :: integratorPort ②

! Private data reference
type(drivers_F90Driver_wrap) :: dp

! local variables for integration
real (selected_real_kind(15, 307)) :: lowBound ③
real (selected_real_kind(15, 307)) :: upBound
integer (selected_int_kind(9)) :: count
real (selected_real_kind(15, 307)) :: value

! Initialize local variables
count = 100000 ④
lowBound = 0.0
upBound = 1.0

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

! get the port ...
call getPort(dp%d_private_data%frameworkServices, & ②
             'IntegratorPort', generalPort, excpt)
call checkExceptionDriver(excpt, & ④
                          'getPort(''IntegratorPort'')')
if(is_null(generalPort)) then
    write(*,*) 'drivers.F90Driver not connected'
    return
endif

! Get an IntegratorPort reference from the general port one
call cast(generalPort, integratorPort) ②

if (not_null(integratorPort)) then ④
    value = -1.0 ! nonsense number to confirm it is set

    ! operate on the port
    call integrate(integratorPort, lowBound, upBound, count, & ⑤
                  value)
    write(*,*) 'Value = ', value
else ! integratorPort is null
    write(*,*) 'DriverF90: incompatible IntegratorPort'
endif

! release the port
call releasePort(dp%d_private_data%frameworkServices, & ⑥
                'IntegratorPort', excpt)
call checkExceptionDriver(excpt, 'releasePort(''IntegratorPort'')')

```

```

    retval = 0 ⑦
    return

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

```

- ① Declarations for modules we need to use in this routine.
- ③ Setup the variables and parameters with which to call the integrator.
- ② These portions of the code are associated with getting a handle to the particular `integrator.IntegratorPort` that the driver's `uses` port has been connected to.

First, we have to declare variables of the appropriate type to hold the port. Because of the way OO programming works in CCA/Babel, we first get the port as a generic `gov.cca.Port` (`gov_cca_Port_t` in Fortran 90) and then cast it to the specific port we need to use, `integrator.IntegratorPort` (`integrator_IntegratorPort_t` in Fortran 90). Recall that `integrator.IntegratorPort` is defined in `student-src/ports/sidl/integrator.sidl`.

Then, we invoke the `getPort` on our `frameworkServices` object. The argument to this method is the local name we used in the `registerUsesPort` invocation, and it returns a `gov.cca.Port` (and an exception).

Finally, we use Babel's `cast` method to cast the generic port to the specific integrator port that we need.

- ④ This code checks that the `getPort` worked, and returned a valid port. If the `getPort` fails, or if the driver's `uses` port has not been connected to an appropriate *provider*, then `getPort` will return a null port object. The `is_null` method is automatically available on the Fortran 90 binding of any SIDL object. Because the driver can't do anything without being properly connected to an integrator, the response to `getPort` failing is to abort by returning a non-zero value.

It is also possible that a valid `gov.cca.Port` would be returned, but it might not be the `integrator.IntegratorPort` we expect. If this is the case, the `cast` will return a null value. The proper action in this case is also to fail gracefully by returning a non-zero result.



Note

`getPort` returning `nil` need not be treated as a fatal error in all cases. For example, a component may be designed so that certain ports are optional -- to be used if present, but to be ignored if not. Another possibility is that the component may be able to accomplish the same thing through several different ports, so that only one of a given group needs to be connected.

- ⑤ Here we actually call the `integrate` method on the `integrator` port we just got a handle for. The signature of the `integrate` method is defined in `student-src/ports/sidl/integrator.sidl`. Notice that while the SIDL definition of `integrate` shows it as a function, returning a double precision result, in Fortran 90, Babel translates this into a subroutine with the return value as an extra argument. This is because Fortran does not support functions returning all types (arrays, for example).
- ⑥ Finally, once we're done using the port, we call `releasePort`.
- ⑦ It is considered impolite for a component to call `exit` because it will bring down the entire application, and possibly crash the framework. Instead, components should simply return.

2. There's one other bit of code we have to provide before we can declare this component complete. In numerous places, we've seen exceptions being returned, and we've been using a routine `checkEx-`

ceptionDriver to deal with them. This is a method that we have to write.

Exceptions are a potentially powerful and sophisticated way of handling errors in software. But for the purposes of this exercise, we're going to take a very simple approach. Our *exception handler* routine simply test whether or not the exception is a null object, and if it is print a message and tell Babel that as far as we're concerned it can delete the `excpt` object. Notice that this routine does *not* exit or abort. As we've noted, it is not considered polite behavior for a component to exit, even in the event of an exception.

In `student-src/components/drivers/f90/drivers_F90Driver_Impl.F90` locate the splicer blocks for miscellaneous code, at the very end of the file, and enter the following:

```
...
! DO-NOT-DELETE splicer.begin(_miscellaneous_code_end)
! Insert extra code here...!
! Small routine (not part of the SIDL interface) for
! checking the exception and printing the message passed as
! and argument
!
subroutine checkExceptionDriver(excpt, msg)
  use SIDL_BaseInterface
  use gov_cca_CCAException
  implicit none
  type(sidl_BaseInterface_t), intent(inout) :: excpt
  character (len=*) :: msg ! in
  if (not_null(excpt)) then
    write(*, *) 'drivers.F90Driver Exception: ', msg
    call deleteRef(excpt)
  end if
end subroutine checkExceptionDriver

! DO-NOT-DELETE splicer.end(_miscellaneous_code_end)
...
```

3. Congratulations, you have completed the implementation of the `F90Driver`! To check your work, run **make** in `student-src/components/drivers/f90`. If you get any compiler errors, you should fix them before going on.
4. At this point, it is a good idea to go up to `student-src` and run **make** to insure that anything else which might depend on the existence of the new `drivers.CXXDriver` component gets built too.

The next step is to test your new driver component, in Section 3.5, "Using Your New Component".

3.4. SIDL and CCA Object Orientation in Fortran

There will be a few artifacts of CCA's(and Babel's) insistence on an object model. Generally the object oriented style of programming groups *state* data and subroutines (or methods) into "objects". Because CCA requires an object model for its components, Fortran programmers will have to become a little familiar with how CCA/Babel implements this in the language. A broad exposition on object oriented concepts is beyond the scope of this tutorial document, more and better information can be found elsewhere [http://en.wikipedia.org/wiki/Object_oriented_programming].

The first thing objects need is a constructor and destructor to initialize state data. For Fortran, the methods ending in `_ctor` and `_dtor` are the constructor and destructor for the component (see listing above). This allows the programmer to create (in the constructor) and delete (in the destructor) state data associated with the component. One thing that almost all components want to store is the `gov_cca_Services` handle that is passed in through the `setServices()`. A complex component may wish to store parameters associated with its function as well.

Looking at the `cca.sidl` specification, Babel maps each CCA SIDL type (e.g. `gov.cca.Port`) to a Fortran type (e.g. `type(gov_cca_Port_t)`).

Because return values cannot accept all Babel types and because Fortran does not provide either an object model or a mechanism for exceptions, these features are placed in the argument list:

- A handle that represents the component and holds the state (or private) data for the component is prepended to the *front* of the argument list for every subroutine method: it is usually called `self`.
- The return value is appended to the *end* of the argument list.
- If there is an exception specified in the `.sidl` file, then the exception (of type `SIDL_BaseInterface_t`) is appended *after* the return value.

As an example, if a user specifies a SIDL snippet such as:

```
file: ./cca-spec-babel/cca.sidl line:108
package gov {
package cca version 0.7.0 {
...
    Port getPort(in string portName) throws CCAException;
...
} // end of package cca
} // end package gov
```

In Fortran translates into:

```
...
type(gov_cca_Port_t) :: port
type(SIDL_BaseInterface_t) :: excpt
type(gov_cca_Services) :: frameworkServices
...
port = getPort(frameworkServices, port, excpt)
```

3.5. Using Your New Component

1. Change directories to `student-src/components/examples` and edit `task1_rc`. This file will assemble and run an application using the new driver component you've created. However it includes lines for both versions of the driver component, and probably you've only implemented one. So you will need to comment out all of the lines which refer to the driver component you did *not* implement.
2. Run the script with `ccafe-single --ccafe-rc task1_rc`. It should run without errors and give you a result like `Value = 3.140347` (since we're using a Monte Carlo integration algorithm, results will vary).

3. Feel free to modify `task3_rc` to assemble applications with different components. The beginning of the `rc` file loads the palette with all of the available components and creates an instance of each. See Chapter 2, *Assembling and Running a CCA Application* for further information and ideas for other “applications” you can construct.

Chapter 4. Creating a Component from an Existing Library

\$Revision: 1.34 \$

\$Date: 2004/08/26 16:41:26 \$

In this exercise, you will wrap an existing (“legacy”) software library as a CCA component (i.e. “componentize” it). The CCA is designed to make it as easy as possible to componentize existing software, and a significant fraction of CCA components are created in this way. While this specific example is minimal, the techniques used to produce a component that uses an existing library with minimal or no modifications to legacy code is applicable for large legacy codes.

The integrator components are Fortran90 wrappers over an existing legacy integrator library. For the purposes of this exercise, the legacy library is located in the `student-src/legacy/f90` directory. The `Integrator.f90` code implements a midpoint rule integration approach. Our goal is to create an integrator component that uses the legacy implementation to compute the integral of a function.

4.1. The legacy Fortran integrator

Our Fortran legacy library (in `student-src/legacy/f90`) contains an integration algorithm, which can be invoked as follows:

```
call integrate_mp(functionParams, lowBound, upBound, count)
```

where `functionParams` is a variable of type `FunctionParams_t`. This type is used to store various function-specific attributes, such as the constant coefficients. The definition of this type is in the `FunctionModule` module, in the `LegacyFunctionModule.f90` file:

```
file: student-src/legacy/f90/LegacyFunctionModule.f90
```

```
module FunctionModule
  implicit none

  type FunctionParams_t
    private
    real, dimension(3) :: coef
  end type FunctionParams_t

contains

  subroutine init(params, coefficients)
    !!INPUT PARAMETERS:
    type(FunctionParams_t), intent(INOUT) :: params
    real, dimension(:), intent(IN) :: coefficients

    integer :: i

    do i = 1 ,3
      params%coef(i) = coefficients(i)
    end do

  end subroutine init

  real function eval(params, x)

    !!INPUT PARAMETERS:
    type(FunctionParams_t), intent(IN) :: params
```

```

real, intent(IN)          :: x

eval = 2 * x

end function eval

end module FunctionModule

```

The legacy integrator (in `Integrator.f90`) uses the midpoint integration algorithm to integrate an arbitrary function that has an `eval` function and uses `FunctionParams_t` to store its state. The complete code for the legacy integrator follows.

```

file: student-src/legacy/f90/Integrator.f90
module Integrator
  use FunctionModule ❶
  implicit none

contains

  real function integrate_mp(functionParams, lowBound, upBound, count)
    implicit none

    !!INPUT PARAMETERS:

    type(FunctionParams_t), intent(IN) :: functionParams ❷
    real, intent(IN) :: lowBound
    real, intent(IN) :: upBound
    integer, intent(IN) :: count

    !!LOCAL VARIABLES:
    real :: sum, h, x, dcount, func_val
    integer :: i

    integrate_mp = -1

    ! Compute integral
    sum = 0.0
    h = (upBound - lowBound) / count

    do i = 0, count
      x = lowBound + h * (i + 0.5)
      func_val = eval(functionParams, x) ❸
      sum = sum + func_val
    end do

    integrate_mp = sum * h

  end function integrate_mp

end module Integrator

```

Notes on the `Integrator.f90` file

- ❶ The `Integrator` module uses the `FunctionModule`, which means that the integrator can only evaluate functions defined in this `FunctionModule`, or other Fortran modules that "extend" it.
- ❷ The `functionParams` argument of the integrator is the only way function parameters can be passed through to the function being evaluated.

Notes on the `FunctionModuleWrapper.f90` file

- 1 The `FunctionModuleWrapper` module uses (includes) the `FunctionPort_type` and `FunctionPort` modules (in `student-src/ports/function/f90`, whose definitions were automatically generated by Babel from the SIDL definition of `function.FunctionPort(student-src/ports/sidl/function.sidl)`).
- 2 The `FunctionParams_t` type that was originally defined in `LegacyFunctionModule.f90`.
- 3 The legacy `FunctionModule` contained the `eval` function; in our wrapper implementation, we create an `eval` interface that contains the new evaluation function, `evalFunction`.
- 4 This is the call to the `evaluate` subroutine of the `FunctionPort`, using the parameters passed to the `evalFunction`. Note that the `params%funcPort` is supposed to have already been set by the caller by using the `setFunctionPort` subroutine defined in this module.



Note

In one of the first steps of this tutorial (see `Building the tutorial source tree`), the entire tutorial tree was built, including the sources in the `student-src/legacy/f90` directory and its subdirectories. Two distinct libraries were created, one containing only legacy codes (`lib/libLegacyIntegrator.a`), and another one (`lib/libWrappedLegacyIntegrator.a`) containing the `FunctionModule` definition in `FunctionModuleWrapper.f90` instead of the `FunctionModule` definition contained in `LegacyFunctionModule.f90`. Also, the compiled modules for each version (legacy and wrapped) are put in separate include directories: `include` for the legacy code, and `include_w` for the wrapped version. While the simple application example (in `simpleApp/Main.f90`) uses only the legacy codes, the `include_w` directory and the `lib/libWrappedLegacyIntegrator.a` are used in the compilation of the Midpoint integrator component that you will write in the steps that follow.

4.3. Implementing the `integrators.Midpoint` component

The `integrator.IntegratorPort` definition

The file `student-src/ports/sidl/integrator.sidl` already contains the `integrator.IntegratorPort` SIDL declaration:

```
package integrator version 1.0 {  
    interface IntegratorPort extends gov.cca.Port  
    {  
        double integrate(in double lowBound, in double upBound,  
                        in int count);  
    }  
}
```

The `integrator.IntegratorPort` SIDL interface extends the `gov.cca.Port` interface, which does not have any methods. Thus, the only method in the `integrator.IntegratorPort` is `integrate`, which takes several arguments that determine the region of integration and the number of points at which the function is evaluated.

4.4. SIDL definition of the Midpoint component

1. We will write a SIDL-based component that implements the port defined in previous steps and calls the `integrate_mp` method implemented in the legacy code described in Section 4.1, “The legacy Fortran integrator” to integrate a function, using function components that implement the `function.FunctionPort` port described in The `integrator.IntegratorPort` definition.

Edit the file, `student-src/components/sidl/integrators.sidl` to define the class for the new integrator component, `integrators.Midpoint`:

```
package integrators version 1.0 {

    // The following components implement all methods of the
    // integrator.IntegratorPort and gov.cca.Component interfaces.
    // Since they use the SIDL 'implements-all' keyword, the
    // methods do not need to (but optionally can) be listed explicitly.

    class Midpoint implements-all integrator.IntegratorPort,
                                   gov.cca.Component
    {
    }

    class MonteCarlo implements-all integrator.IntegratorPort,
                                     gov.cca.Component
                                     gov.cca.ComponentRelease
    {
        // integrator.IntegratorPort methods:
        double integrate(in double lowBound, in double upBound,
                        in int count);

        // gov.cca.Component methods:
        void setServices(in gov.cca.Services services)
            throws gov.cca.CCAException;

        // gov.cca.ComponentRelease methods:
        void releaseServices(in gov.cca.Services services)
            throws gov.cca.CCAException;
    }
}
```

Note that the `Midpoint` class, unlike the `MonteCarlo` class does not implement the `gov.cca.ComponentRelease` interface, which is optional.

2. Edit the file `student-src/components/MakeIncl.components` to add a new component description in the `COMPONENTS` variable, which contains the list of components in this directory. Each value consists of the fully-qualified name of the component (including packages), to which we append “-language”, where language is one of `c`, `c++`, or `f90`. In this case, the name is `integrators.Midpoint`, and the language is `f90`, so you need to add **`integrators.Midpoint-f90`**. The updated value of `COMPONENTS` should look like something like this:

```
COMPONENTS = functions.PiFunction-c++ \
              integrators.MonteCarlo-f90 randomgens.RandNumGenerator-c++ \
              drivers.F90Driver-f90 drivers.CXXDriver-c++ \
              integrators.Midpoint-f90
```

Note the backslash (“\”) that has to be added in order to extend the entry to the next line.

3. In the `student-src/components` directory, run `make .repository`. This will generate the XML representation of the `integrator.Midpoint` SIDL class and store it in the stu-

dent-src/xml_repository directory.

4. In the student-src/components directory, run `make .integrators.Midpoint-f90`. This will generate Fortran 90 server code for the `integrators.Midpoint` component class.

4.5. Fortran 90 implementation of the Midpoint integrator

4.5.1. The Midpoint module implementation

- After the Fortran 90 code has been generated by Babel, in `student-src/components/integrators/f90`, edit the Fortran module definition to define data that will be stored in each instance of this component:

```
file: student-src/components/integrators/f90/integrators_Midpoint_Mod.F90
#include "integrators_Midpoint_fAbbrev.h"
module integrators_Midpoint_impl

! DO-NOT-DELETE splicer.begin(integrators.Midpoint.use)
! Insert use statements here...

! CCA framework services module
use gov_cca_Services

! Use a "wrapper" module for the legacy FunctionModule module
use FunctionModule 1

! Use legacy Integrator module
use Integrator 2

! DO-NOT-DELETE splicer.end(integrators.Midpoint.use)

type integrators_Midpoint_priv
sequence
! DO-NOT-DELETE splicer.begin(integrators.Midpoint.private_data)

! Handle to framework Services object
type(gov_cca_Services_t) :: frameworkServices 3

! Function parameters (required by legacy integrator)
type(FunctionParams_t) :: funcParams 4

! DO-NOT-DELETE splicer.end(integrators.Midpoint.private_data)
end type integrators_Midpoint_priv

type integrators_Midpoint_wrap
sequence
type(integrators_Midpoint_priv), pointer :: d_private_data
end type integrators_Midpoint_wrap

end module integrators_Midpoint_impl
```

Notes on the `integrators_Midpoint_Mod.F90` file

- ❶ The `integrators_Midpoint` module uses the `FunctionModule`, which means that the integrator can only evaluate functions defined in this `FunctionModule`, or other Fortran modules that "extend" it.
- ❷ This component stores a handle to the framework's `Services` object, equivalently to the way the `Driver` component was implemented in Step 2.
- ❸ The legacy `Integrator` module is included.
- ❹ The `integrators.Midpoint` component, like the legacy integrator (see `Integrator.f90`) requires that the function whose integral is to be computed provides its state via the `FunctionParams_t` type.

4.5.2. Defining the constructor and destructor

In the same directory (`student-src/components/integrators/f90`), edit the `integrators_Midpoint_Impl.F90` and insert the code between splicer blocks of the `integrators_Midpoint__ctor_mi`, `integrators_Midpoint__dtor_mi`, and `setServices` subroutines:

```
file: student-src/components/integrators/f90/integrators_Midpoint_Impl.F90
...
!
! Class constructor called when the class is created.
!
recursive subroutine integrators_Midpoint__ctor_mi(self)
  use integrators_Midpoint
  use integrators_Midpoint_impl
  ! DO-NOT-DELETE splicer.begin(integrators.Midpoint.__ctor.use)
  ! Insert use statements here...
  ! DO-NOT-DELETE splicer.end(integrators.Midpoint.__ctor.use)
  implicit none
  type(integrators_Midpoint_t) :: self ! in

! DO-NOT-DELETE splicer.begin(integrators.Midpoint.__ctor)
! Insert the implementation here...

  ! Access private data
  type(integrators_Midpoint_wrap) :: dp
  ! Allocate memory and initialize
  allocate(dp%d_private_data)
  call set_null(dp%d_private_data%frameworkServices)
  call integrators_Midpoint__set_data_m(self, dp)

! DO-NOT-DELETE splicer.end(integrators.Midpoint.__ctor)
end subroutine integrators_Midpoint__ctor_mi

!
! Class destructor called when the class is deleted.
!
recursive subroutine integrators_Midpoint__dtor_mi(self)
  use integrators_Midpoint
```

```

use integrators_Midpoint_impl
! DO-NOT-DELETE splicer.begin(integrators.Midpoint._dtor.use)
! Insert use statements here...
! DO-NOT-DELETE splicer.end(integrators.Midpoint._dtor.use)
implicit none
type(integrators_Midpoint_t) :: self ! in

! DO-NOT-DELETE splicer.begin(integrators.Midpoint._dtor)
! Insert the implementation here...

! Access private data and deallocate storage
type(integrators_Midpoint_wrap) :: dp
call integrators_Midpoint__get_data_m(self, dp)

! Decrement reference count for framework services handle
if (not_null(dp%d_private_data%frameworkServices)) then
    call deleteRef(dp%d_private_data%frameworkServices)
end if

deallocate(dp%d_private_data)

! DO-NOT-DELETE splicer.end(integrators.Midpoint._dtor)
end subroutine integrators_Midpoint__dtor_mi

```

4.5.3. The setServices implementation

In this step we continue to edit the student-src/components/integrators/f90/integrators_Midpoint_Impl.F90 file, adding the implementation of the setServices subroutine, which is part of the gov.cca.Component. Note that in order to accommodate identifier length restriction in Fortran (31 characters), the name of the subroutine was automatically shortened by Babel. The unmangled name is always visible in the comment preceding the subroutine in the Fortran generated code.

```

...
recursive subroutine Midpoi_setServices6_m9htaw4m_mi(self, services, &
    exception)
    use sidl_BaseInterface
    use integrators_Midpoint
    use gov_cca_Services
    use gov_cca_CCAException
    use integrators_Midpoint_impl
    ! DO-NOT-DELETE splicer.begin(integrators.Midpoint.setServices.use)
    ! Insert use statements here...

    use gov_cca_TypeMap
    use gov_cca_Port
    use SIDL_BaseInterface

    ! DO-NOT-DELETE splicer.end(integrators.Midpoint.setServices.use)
    implicit none
    type(integrators_Midpoint_t) :: self ! in
    type(gov_cca_Services_t) :: services ! in
    type(sidl_BaseInterface_t) :: exception ! out

! DO-NOT-DELETE splicer.begin(integrators.Midpoint.setServices)
! Insert the implementation here...

type(gov_cca_TypeMap_t)      :: myTypeMap
type(gov_cca_Port_t)        :: integratorPort
type(SIDL_BaseInterface_t) :: excpt
! Access private data

```

```

type(integrators_Midpoint_wrap) :: dp
call integrators_Midpoint__get_data_m(self, dp)

! Set my reference to the services handle
dp%d_private_data%frameworkServices = services

call addRef(services)

! Create a TypeMap with my properties
call createTypeMap(dp%d_private_data%frameworkServices, myTypeMap, excpt)
call checkExceptionMid(excpt, 'setServices createTypeMap call')

call cast(self, integratorPort)

! Register my provides port
call addProvidesPort(dp%d_private_data%frameworkServices, integratorPort, &
    'IntegratorPort', 'integrator.IntegratorPort', &
    myTypeMap, excpt)
call checkExceptionMid(excpt, 'setServices addProvidesPort: IntegratorPort')

! The ports I use
call registerUsesPort(dp%d_private_data%frameworkServices, &
    'FunctionPort', 'function.FunctionPort', &
    myTypeMap, excpt)
call checkExceptionMid(excpt, 'setServices registerUsesPort: FunctionPort')

call deleteRef(myTypeMap)

! DO-NOT-DELETE splicer.end(integrators.Midpoint.setServices)
end subroutine Midpoi_setServices6_m9htaw4m_mi

```

4.5.4. The integrate implementation

Continuing your edits in the `integrators_Midpoint_Impl.F90` file, fill in the implementation of the `integrator.IntegratorPort` interface component, inserting the call to the legacy integrator in the `integrate` method.

```

file: student-src/components/integrators/f90/integrators_Midpoint_Impl.F90
recursive subroutine Midpoint_integrateekg4n6wqha_mi(self, lowBound, upBound, &
    count, retval)
    use integrators_Midpoint
    use integrators_Midpoint_impl
    ! DO-NOT-DELETE splicer.begin(integrators.Midpoint.integrate.use)
    ! Insert use statements here...

    use function_FunctionPort
    use randomgen_RandomGeneratorPort
    use gov_cca_Services
    use gov_cca_Port
    use sidl_BaseInterface

    use Integrator          ! Legacy integrator module
    use FunctionModule     ! Legacy function module wrapper

    ! DO-NOT-DELETE splicer.end(integrators.Midpoint.integrate.use)
    implicit none
    type(integrators_Midpoint_t) :: self ! in
    real (selected_real_kind(15, 307)) :: lowBound ! in
    real (selected_real_kind(15, 307)) :: upBound ! in
    integer (selected_int_kind(9)) :: count ! in
    real (selected_real_kind(15, 307)) :: retval ! out

```

```

! DO-NOT-DELETE splicer.begin(integrators.Midpoint.integrate)
! Insert the implementation here...

type(gov_cca_Port_t) :: generalPort
type(function_FunctionPort_t) :: functionPort
type(randomgen_RandomGeneratorPort_t) :: randomPort
type(SIDL_BaseInterface_t) :: excpt

! Legacy types and wrappers:
type(FunctionParams_t) :: funParams

! Private data reference
type(integrators_Midpoint_wrap) :: dp

! Copies of base type arguments to the integrate method
real :: lbnd, ubnd
integer :: cnt

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

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

if (not_null(dp%d_private_data%frameworkServices)) then

    ! Obtain a handle to a FunctionPort
    call getPort(dp%d_private_data%frameworkServices, &
        'FunctionPort', generalPort, excpt)

    if (is_null(excpt)) then

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

            ! Set the function port in the FunctionModule wrapper
            call setFunctionPort(funParams, functionPort)

            ! Invoke legacy integrator algorithm to compute integral
            lbnd = lowBound
            ubnd = upBound
            cnt = count
            retval = integrate_mp(funParams, lbnd, ubnd, cnt)

        else ! functionPort is null
            write(*,*) 'Exception: Midpoint: incompatible FunctionPort'
        endif

        ! Free ports
        call releasePort(dp%d_private_data%frameworkServices, &
            'FunctionPort', excpt)
        call checkExceptionMid(excpt, 'releasePort(''FunctionPort'')')

    else ! excpt is not null

        call checkExceptionMid(excpt, 'getPort(''FunctionPort'')')

    endif

else ! frameworkServices is null
    write(*,*) 'Error: Midpoint: integrate called before setServices'
endif

! DO-NOT-DELETE splicer.end(integrators.Midpoint.integrate)
end subroutine Midpoint_integrateekg4n6wqha_mi

```

Finally, in the `integrators_Midpoint_Impl.F90` file, find the very last splicer block (labeled `_miscellaneous_code_end`) and add the following helper subroutine:

```
file: student-src/components/integrators/f90/integrators_Midpoint_Impl.F90
!
! Small routine (not part of the SIDL interface) for
! checking the exception and printing the message passed as
! and argument
!
subroutine checkExceptionMid(excpt, msg)
  use SIDL_BaseInterface
  use gov_cca_CCAException
  implicit none
  type(sidl_BaseInterface_t), intent(inout) :: excpt
  character (len=*) :: msg ! in
  if (not_null(excpt)) then
    write(*, *) 'integrators.Midpoint Exception: ', msg
    call deleteRef(excpt)
  end if
end subroutine checkExceptionMid
```

4.6. Building the Fortran 90 implementation of the `integrators.Midpoint` component.

1. In the `student-src/components/integrators/f90` directory, edit the user-defined settings in `MakeIncl.user` file to specify the include paths and library location of the legacy integrator library.

```
file: student-src/components/integrators/f90/MakeIncl.user
# Include path directives, including paths to Fortran modules
INCLUDES = \
  $(CCASPEC_BABEL_F90MFLAG)$(COMPONENT_TOP_DIR)/../legacy/f90/include_w

# Library paths and names
LIBS = \
  -L$(COMPONENT_TOP_DIR)/../legacy/f90/lib \
  -lWrappedLegacyIntegrator
```

Note that the `INCLUDES` variable is used by the Fortran compiler to locate compiled module information; since the flag used to specify the search path for modules is not the same in all compilers, we use the variable `CCASPEC_BABEL_F90MFLAG`, which was set during the configuration and installation of Babel and CCA tools. The `COMPONENT_TOP_DIR` variable is set automatically when the component's Makefile is generated from the `student-src/components/Makefile_template.server` makefile template.

Also note that the library specified in the definition of the `LIBS` variable is not the original legacy library, which contained the original definition of `FunctionModule` and `FunctionParams_t`. The only difference between the legacy library and `libWrappedLegacyIntegrator.a` is that the original `FunctionModule` has been replaced with a new definition of `FunctionModule` in `FunctionModuleWrapper.f90` as described in Section 4.2, “The `FunctionModule` wrapper.”

2. In `student-src/components/integrators/f90`, run **make**. This will build the dynamic component libraries and generate the `*.cca` files needed to load these libraries and instantiate the

components in the Ccaffeine framework. After a successful build, you should be able to see the `libintegratorsMidpoint-f90.so` and `libintegratorsMidpoint-f90.so.cca` files in the `student-src/components/lib` directory.



Note

In this step, the makefile automatically generated the `.cca` file needed by the Ccaffeine and Babel runtime systems to identify and locate babel components. This file can also be generated manually by executing the following command in the directory `student-src/components/lib`:

```
$CCA/bin/genSCLCCA.sh cca \  
  `pwd`/libintegratorsMidpoint-f90.so integrators.Midpoint \  
  integratorsMidpoint dynamic private now \  
> integrators.Midpoint.cca
```

4.7. Using your new `integrators.Midpoint` component

To see the new Midpoint integrator component in action, in `student-src/components`, run

```
ccafe-single --ccafe-rc examples/task2_rc
```

Feel free to modify `task2_rc` to assemble applications with different components. The beginning of the `rc` file loads the palette with all of the available components and creates an instance of each. See Chapter 2, *Assembling and Running a CCA Application* for further information and ideas for other “applications” you can construct.

The output should look something like this:

```
(3587) CmdLineClientMain.cxx: MPI_Init not called in ccafe-single mode.  
(3587) CmdLineClientMain.cxx: Try running with ccafe-single --ccafe-mpi yes , or  
(3587) CmdLineClientMain.cxx: try setenv CCAFE_USE_MPI 1 to force MPI_Init.  
(3587) my rank: -1, my pid: 3587  
my rank: -1, my pid: 3587  
my rank: -1, my pid: 3587  
CCAFFEINE configured with babel.  
my rank: -1, my pid: 3587  
Type: One Processor Interactive
```

```
cca>  
CmdContextCCAMPI::initRC: Found task2_rc.  
  
cca># There are allegedly 8 classes in the component path  
  
cca>  
cca>Loaded drivers.CXXDriver NOW GLOBAL .  
  
cca>Loaded functions.PiFunction NOW GLOBAL .  
  
cca>Loaded integrators.Midpoint NOW GLOBAL .  
  
cca>  
cca>driver of type drivers.CXXDriver
```

```
successfully instantiated

cca>pifunc of type functions.PiFunction
successfully instantiated

cca>midpoint of type integrators.Midpoint
successfully instantiated

cca>
cca>driver))))IntegratorPort---->IntegratorPort(((midpoint
connection made successfully

cca>midpoint))))FunctionPort---->FunctionPort(((pifunc
connection made successfully

cca>
cca>Value = 3.141553
##specific go command successful

cca>
cca>
bye!
exit
```

Chapter 5. Creating a New Component from Scratch

\$Revision: 1.14 \$

\$Date: 2004/08/26 16:28:35 \$

In this exercise, you will put together what you've learned in the previous tasks to create a complete component from scratch. We will add to the list of `function` components by creating one that returns the cube of the argument. The new component class will be named `functions.CubeFunction`, and it will implement the `function.FunctionPort` interface, just as the other function components do. The following procedures will guide you through writing the component in C++, though very little would change for if you wanted to implement it in another Babel-supported language.

5.1. SIDL Component Class Specification

In this step, we will define the `function.CubeFunction` SIDL class and build its xml repository representation

1. Edit the file `student-src/components/sidl/functions.sidl`, and add the definition of the class `CubeFunction` to the package `functions`

```
package functions version 1.0 {  
  
    class LinearFunction implements function.FunctionPort,  
                                   gov.cca.Component  
    {  
        // function.FunctionPort methods:  
        double evaluate(in double x);  
  
        // gov.cca.Component methods:  
        void setServices(in gov.cca.Services servicesHandle)  
                    throws gov.cca.CCAException;  
    }  
  
    ... some definitions skipped ...  
  
    class PiFunction implements-all function.FunctionPort,  
                                   gov.cca.Component  
    {  
    }  
    class CubeFunction implements-all function.FunctionPort,  
                                   gov.cca.Component  
    {  
    }  
}
```

2. Edit the file `student-src/components/MakeIncl.components` to add a new component description in the `COMPONENTS` variable, which contains the list of components in this directory. Each value consists of the fully-qualified name of the component (including packages), to which we append "-language", where language is one of `c`, `c++`, or `f90`. In this case, the name is `functions.CubeFunction`, and the language is `c++`. The updated value of `COMPONENTS` should look like this:

```
COMPONENTS = functions.PiFunction-c++ \  
            integrators.MonteCarlo-f90 randomgens.RandNumGenerator-c++ \  
            drivers.CXXDriver-c++ integrators.Midpoint-f90 \  
            functions.CubeFunction-c++
```

Note the backslash (“\”) that has to be added in order to extend the entry to the next line.

3. In the `student-src/components` directory, run **make .repository**. This will regenerate the XML representation of the SIDL component class definitions (including the newly added class `CubeFunction` and store them in the `student-src/xml_repository` directory.

The output from this step should look something like this:

```
touch .sidl  
  
### Generate XML for SIDL packages containing component declarations  
babel -t xml -R../xml_repository -R/san/shared/cca/tutorial/share/cca-spec-babel  
Babel: Parsing URL "file:/.automount/whale/root/san/r1a010/elwasifw/handson/com  
Babel: Warning: Symbol exists in XML repository: drivers.F90Driver-v1.0  
Babel: Warning: Symbol exists in XML repository: drivers.CXXDriver-v1.0  
Babel: Parsing URL "file:/.automount/whale/root/san/r1a010/elwasifw/handson/com  
Babel: Warning: Symbol exists in XML repository: functions.LinearFunction-v1.0  
Babel: Warning: Symbol exists in XML repository: functions.NonlinearFunction-v1  
Babel: Warning: Symbol exists in XML repository: functions.PiFunction-v1.0  
Babel: Parsing URL "file:/.automount/whale/root/san/r1a010/elwasifw/handson/com  
Babel: Warning: Symbol exists in XML repository: integrators.MonteCarlo-v1.0  
Babel: Warning: Symbol exists in XML repository: integrators.Midpoint-v1.0  
Babel: Warning: Symbol exists in XML repository: integrators.ParallelMid-v1.0  
Babel: Parsing URL "file:/.automount/whale/root/san/r1a010/elwasifw/handson/com  
Babel: Warning: Symbol exists in XML repository: randomgens.RandNumGenerator-v1  
touch .repository
```

5.2. Generating Babel Server Code for the New Component

- In the `student-src/components` directory run **make .functions.CubeFunction-c++** to generate the C++ server-side binding for the component class `functions.CubeFunction`. The output from this step should look something like this:

```
### Generate a C++ implementation for the CubeFunction component  
babel -s c++ -R../xml_repository -R/home/elwasif/CCA/cca-spec-babel-cvs/share/c  
-g -u -E -l -m "functions.CubeFunction." --suppress-timestamp functions.CubeFun  
Babel: Resolved symbol "functions.CubeFunction" ...  
touch .functions.CubeFunction
```

Upon completion of this step, the directory `student-src/components/functions/c++` should contain two additional files, `functions_CubeFunction_Impl.cc` and `functions_CubeFunction_Impl.hh` which will be edited to provide the implementation of the

newly defined component.

5.3. Implementing the New Component

1. Edit the file `functions_CubeFunction_Impl.hh` in the directory `student-src/components/functions/c++`. You will need to add the declaration for the `gov::cca::Services` object to the private object state. This will be done inside the Babel splicer block `functions.CubeFunction._implementation`. We will call this variable `myServices`. Upon completion, this splicer block should look like this:

```
...
// DO-NOT-DELETE splicer.begin(functions.CubeFunction._implementation)
// Put additional implementation details here...
gov::cca::Services myServices;
// DO-NOT-DELETE splicer.end(functions.CubeFunction._implementation)
...
```

2. Edit the file `functions_CubeFunction_Impl.cc` in the directory `student-src/components/functions/c++` to provide the implementation details. First, you'll need to edit the body of the `setServices` method (between the Babel splicer blocks `functions.CubeFunction.setServices`). Upon completion, this part of the file should look like this:

```
...
// DO-NOT-DELETE splicer.begin(functions.CubeFunction.setServices)
// insert implementation here

myServices = services;
gov::cca::TypeMap tm = services.createTypeMap();
if(tm._is_nil()) {
    fprintf(stderr, "Error:: %s:%d: gov::cca::TypeMap is nil\n",
        __FILE__, __LINE__);
    exit(1);
}
gov::cca::Port p = self; // Babel required casting
if(p._is_nil()) {
    fprintf(stderr, "Error:: %s:%d: Error casting self to gov::cca::Port \n",
        __FILE__, __LINE__);
    exit(1);
}

services.addProvidesPort(p,
    "FunctionPort",
    "function.FunctionPort", tm);

gov::cca::ComponentRelease cr = self; // Babel required casting
services.registerForRelease(cr);
return;

// DO-NOT-DELETE splicer.end(functions.CubeFunction.setServices)
...
```

- Next you will need to edit the implementation for the method `evaluate` inside the Babel splicer block `functions.CubeFunction.evaluate`. After adding the implementation for this method, the body should look like this

```
...
// DO-NOT-DELETE splicer.begin(functions.CubeFunction.evaluate)
// insert implementation here
return x*x*x;
// DO-NOT-DELETE splicer.end(functions.CubeFunction.evaluate)
...
```

- To build the newly written component into a usable library, type **make** in the directory `student-src/components/functions/c++`. This will compile, link, and install the new component into a library that is installed in the directory `student-src/components/lib`.



Note

In this step, the makefile automatically generated the `.cca` file needed by the Ccaffeine and Babel runtime systems to identify and locate babel components. This file can also be generated manually by executing the following command in the directory `student-src/components/lib`:

```
$CCA/bin/genSCLCCA.sh cca \
  `pwd`/libfunctionsCubeFunction-c++.so functions.CubeFunction \
  cubeFunction dynamic private now \
  > functions.CubeFunction.cca
```

5.4. Using Your New Component

- Change directories to `student-src/components/examples` and edit `task3_rc`. This file will assemble and run an application using all of the new components you've created. However it includes lines for both versions of the driver component, and probably you've only implemented one. So you will need to comment out all of the lines which refer to the driver component you did *not* implement.
- Run the script with `ccafe-single --ccafe-rc task3_rc`. It should run without errors and give you a result of `Value = 0.250010`.
- Feel free to modify `task3_rc` to assemble applications with different components. The beginning of the `rc` file loads the palette with all of the available components and creates an instance of each. See Chapter 2, *Assembling and Running a CCA Application* for further information and ideas for other “applications” you can construct.

Appendix A. Installing the CCA Environment and Tutorial Source Code

\$Revision: 1.8 \$

\$Date: 2004/08/27 04:25:26 \$

There are two different tar balls that you will have to install to get the tutorial code to work on your own machine:

- *The CCA tool chain.* This source tree contains all of the tools developed under the CCA and is needed by the tutorial source to build and run components. You can download the source [<http://cca-forum.org/download/cca-tools>] from the net.
- *The tutorial source code.* The source for this tutorial must be built against the CCA tool chain. You should be careful to rebuild the tutorial code every time you change the build for the tool chain. You can download the latest source [<http://cca-forum.org/download/tutorial>] from the net. You should be careful to get the latest versions of both (or just use the two url's above) to make sure you have compatible versions.

A.1. Building the CCA Tool Chain

The build requirements vary from platform to platform, but here we will stick with x86 Linux. The requirements for this environment are:

- gcc >= 3.2
- Java Software Development Kit >= 1.4
- Gnome XML C Parser (libxml2) -- most recent Linux distro's already have it, regardless of whether Gnome is installed.
- GNU autobuild tools: anything recent.
- A connection to the internet.

Untar the cca-build tools tar ball some place that is convenient to build. Look at the README for up to the minute information.

There are a panoply of options to choose from here. First are the language choices: C,C++,F77 *only*, F90/95, Python (version 2.2 or better must be present), Java (version 1.4 or better). Much of what dictates the language support is the information given to the configure script. C,C++, and Java are pretty much automatic, and if the appropriate version is present, Python will be automatic as well.

Fortran 90/95 is a different story. Because there are no non-proprietary versions and because the Fortran 90/95 standards eschew language interoperability (even between different F90/95 compilers), special instructions will need to be given to the **configure** script. There is also a "failsafe" mode that can be used that will hopefully give you the tools just enabled for C,C++, Java and, if present, Python. Just type:

```
$ cd $INSTALL/cca-tools-XXXX
$ ./configure --with-localsrc
```

If you desire to configure Fortran90/95 support you must identify your compiler and the absolute path to the compiler executable. This is done with two options to configure: **--with-F90-vendor=VENDOR** (for the brand of compiler) and **--with-F90=/full/path/to/compiler** (for the absolute path). The possible options for **VENDOR** are:

- **Absoft** (Absoft)
- **Alpha** (Hp Compaq Fortran)
- **Cray** (Cray Fortran)
- **IBMXL** (IBM XL Fortran)
- **Intel** (Intel v8)
- **Intel7** (Intel v7)
- **Lahey** (Lahey)
- **NAG** (NAG)
- **MIPSpro** (SGI MIPS Pro)
- **SUNWsprow** (SUN Solaris)

An example using the Intel (version 8) compiler:

```
$ configure --with-F90-vendor=Intel --with-F90=/opt/intel_fc_80/bin/ifort
```

Generally the configure script will find MPI if it is installed in a reasonable place. Currently only MPICH is supported automatically. LAM is supported with special options. If MPI is getting in the way use **--with-mpi=no**. If the configure script is finding the wrong MPI use **--with-mpi=/wherever/mpich/is**.

Before you run **configure** make sure that:

- **java.jvac**, etc. are in your path.
- If you configured for it, make sure the Fortran90 compiler is in your path.
- Make sure that ".so" libraries for the Fortran libraries are in the `LD_LIBRARY_PATH` so that they can be found at configure time.

Now it is safe to run configure with the options you have selected. Now type:

```
$ make
```

and everything should build. In the exceedingly rare case (:-) that it doesn't, contact someone of authority [mailto:tutorial-wg@cca-forum.org]. There is no need to type **make install** everything will install in `$INSTALL/cca-tools-XXXXX/local`.

A.2. Building the Tutorial Source

Most of the configuration information for building the tutorial source will be obtained from the previous build of the CCA tool chain. This build requires the modifications to the `PATH` and `LD_LIBRARY_PATH` in the previous section as well as two other modifications:

- Add the absolute path to the CCA tool chain binaries to your path, here: `$INSTALL/src/cca-tools-XXXXX/local/bin`.
- Add the absolute library path for the CCA tool chain to your `LD_LIBRARY_PATH`, here: `$INSTALL/src/cca-tools-XXXXX/local/lib`.

Now just type `cd $INSTALL/tutorial-src-XXXX` and `make` and `make check` to test the build.