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 2006-08-23 21:22:55-04:00 (time this instance was generated)

Licensing Information

Requested Attribution. CCA Forum Tutorial Working Group, *A Hands-On Guide to the Common Component Architecture, version 0.4.1_rc1*, 2006, http://www.cca-forum.org/tutorials/.

Or in BibTeX format:

```
@Manual{hog-cca:0.4.1_rc1,
  title =   {A Hands-On Guide to the Common Component Architecture},
  author =  {The Common Component Architecture Forum Tutorial
             Working Group},
  edition = {0.4.1_rc1},
  year =    2006,
  note =    {http://www.cca-forum.org/tutorials/}
}
```

Table of Contents

Preface

$Revision: 1.17 $
$Date: 2005/09/22 21:39:58 $

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

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

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

1. Help us Improve this Guide

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

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

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

3. Typographic Conventions

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

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

4. File and Directory Naming Conventions

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

Warning

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

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

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

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

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

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

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

5. Acknowledgments

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

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

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

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

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

Chapter 1. Introduction

$Revision: 1.24 $
$Date: 2005/09/22 21:39:58 $

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

Starting with Chapter 6, *Using TAU to Monitor the Performance of Components* we have some more advanced examples, including a demonstration of the *proxy component* concept using the TAU performance monitoring toolkit, an illustration of using arrays in SIDL, and a discussion of some of the detailed mechanics of argument passing in SIDL.

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. (Languages: C++ or F90)
- In Chapter 4, *Creating a Component from an Existing Library*, you will wrap up an existing Fortran90 library as an integrator component. (Language: F90)
- In Chapter 5, *Creating a New Component from Scratch*, you will create a new function component from scratch. (Languages: C++, C)
- In Chapter 6, *Using TAU to Monitor the Performance of Components*, you will use the TAU performance observation tool [http://www.cs.uoregon.edu/research/paracomp/tau/tautools/] to automatically instrument a component interface and monitor the performance of the application.
- In Chapter 7, *Understanding arrays and component state*, you will see examples of how to work with arrays in a multi-language environment, including writing your own component. (Languages: F77, F90, C)
- In Chapter 8, *Understanding objects and passing modes*, you will debug a simple unit conversion library, illustrating use of objects and argument passing modes in SIDL. (Languages: Python, C++)

You are strongly advised to at least read and understand Chapter 2, *Assembling and Running a CCA Application* before going on to later exercises. Chapter 3, *Sewing CCA Components into an Application: the Driver Component* through Chapter 6, *Using TAU to Monitor the Performance of Components* build on each other, though after completing Chapter 3, *Sewing CCA Components into an Application: the Driver Component*, you will have a sufficient set of components to assemble a working integrator application. Chapter 7, *Understanding arrays and component state* and Chapter 8, *Understanding objects and passing modes* are independent of the earlier exercises and can be done separately.

In Chapter 2, *Assembling and Running a CCA Application*, you'll be working with a complete version of the tutorial code tree. Then in Chapter 3, *Sewing CCA Components into an Application: the Driver Component* and the subsequent exercises, you'll start from your own copy of a separate stripped-down "student" version of the tutorial code tree and build up to the complete set of components as you work through the exercises. In this way, the separate complete tutorial code tree can always serve as a reference if you run into problems. Of course if you're working through this Guide as part of an organized tutorial, there should be instructors around who can help you. And if you're working on your own, you can email us for help at `<tutorial-wg@cca-forum.org>`.

1.1. The CCA Software Environment

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

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

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

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

1.2. Where to Go from Here

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

1. **Getting Connected**

    a. **Organized Tutorial Participant**

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

    b. **Self-Study User**

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

2. **Preparing the CCA Environment**

    a. **Organized Tutorial Participant**

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

b. **Self-Study User**

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

3. **Building the Tutorial Code**

a. **Organized Tutorial Participant**

Once again, the tutorial code will already have been built in a central location. (Though later on, you'll have to build your own copy of the student code tree, so you don't completely escape the work.)

b. **Self-Study User**

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

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

Chapter 2. Assembling and Running a CCA Application

$Revision: 1.44 $
$Date: 2006/08/24 00:50:08 $

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

Drivers:	`drivers.CXXDriver`*, `drivers.F90Driver`*, `drivers.PYDriver`
Integrators:	`integrators.MonteCarlo`, `integrators.Midpoint`*, `integrators.Trapezoid`, `integrators.Simpson`
Functions:	`functions.PiFunction` ($4/(1+x^2)$, which integrates to pi), `functions.CubeFunction`* (x^3, which integrates to 0.25), `functions.LinearFunction` (x, which integrates to 0.5)
Random Number Generators:	`randomgens.RandNumGenerator` (required by `integrators.MonteCarlo`)

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 `tutorial-src` tree includes completed examples of *all* of the components.

There 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.

In the interests of time, it is not necessary for you to actually *do* all three procedures before moving on to the later chapters, but you should certainly *read and understand* this chapter before moving on. In particular, you will find that Section 2.1, “A CCA Application in Detail” has the most detailed explanations of what is going on, but at the same time, it is the most tedious procedure to actually perform because it involves a lot of typing, and doesn't tolerate typing errors well. However the later sections and subsequent chapters assume that you understand this material.

Note

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

Tutorial and Student Code Trees.

Tip

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

2.1. 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 `$TUTORIAL_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.

   You should see something like this (note that some of the output lines have been folded for presentation here, indicated by “\”):

   ```
   (16251) CmdLineClientMain.cxx: MPI_Init not called in \        (1)
           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     (1)

   CCAFFEINE configured with babel.(2)

   cca>
   CmdContextCCAMPI::initRC: No rc file found. Pallet may be empty.(3)
   ```

   (1) 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 comprise the component). 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 TUTORIAL_SRC/components/lib ❶
# There are allegedly 19 classes in the component path

cca>path
pathBegin
pathElement TUTORIAL_SRC/components/lib
pathEnd
```

❶ Remember that when you see markup like "*`TUTORIAL_SRC`*" you should replace it with the appropriate directory location on the system you're using. If you're part of an organized tutorial, this will be on the handout you received.

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 `$CCA_COMPONENT_PATH` environment variable.
path prepend	Adds a directory to the beginning of the current path.
path set	Sets the path to the value provided.

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, which is based on the components (specifically, the `.cca` files) Ccaffeine finds in the **path** you set. 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*. To begin with, we're going to load a set of components that will allow us to build just one specific integration application; later, we'll add other components and show how you can “plug and play” to create a variety of distinct integration applications from the full pallette of available components.

```
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 component_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).

5. 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). In this case, we need to connect appropriate ports on the driver to the integrator, and the integrator to the function to be integrated. Since we're using the Monte Carlo method in this integrator, the integrator also needs to be connected to a random number generator.

```
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                                               ❷

cca>display chain                                                          ❸
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, not the *provider*.

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** ***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 of type `GoPort`. The purpose of this port is to initiate the execution of a component. The command **go** ***component_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 $4/(1+x^2)$.

Note

The type, or *class name* of the port must be `GoPort`, but the *instance name* of the port can be something else. Both of these are determined by the software developer who writes the code for the component. You can use the **display component** command in Ccaffeine to check both the class names and instance names of ports a component *uses* and *provides*.

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.

7. 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*.

8. 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                                          (1)

cca>disconnect integratorsMonteCarlo FunctionPort functionsPiFunction \
    FunctionPort
integratorsMonteCarlo))))FunctionPort-\ \-FunctionPort((((functionsPiFunction
connection broken successfully                                          (1)
```

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

Note

Step 7 and Step 8 could have been done in either order.

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 \                  ❶
 randomgens.RandNumGenerator

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`).

Note

The Monte Carlo algorithm is unique among the integrators we have implemented in requiring a source of random numbers. As a consequence, the mid-point and other integrators do not have a *uses* port for a random number generator, and it is not necessary, or even *possble* in the CCA context, to connect a random number generator to any of the integrators besides the Monte Carlo.

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
```

11. 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.

1. 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 $TUTORIAL_SRC/components/examples/task0_rc .`** and view the file. 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 TUTORIAL_SRC/components/lib ❸
path

# Step 3 ❷

palette
repository get-global drivers.CXXDriver
repository get-global drivers.F90Driver
repository get-global drivers.PYDriver
repository get-global functions.LinearFunction
repository get-global functions.CubeFunction
repository get-global functions.PiFunction
repository get-global integrators.Trapezoid
repository get-global integrators.Midpoint
repository get-global integrators.MonteCarlo
repository get-global integrators.Simpson
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 ❹
```

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 In your copy of the `rc` file, this should be the fully-qualified path to the *`TUTORIAL_SRC`* directory.

❹ 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 **csh** or **tcsh** shells; if you're using the **sh** or **bash** shells, the command is **`ccafe-single --ccafe-rc task0_rc > task0.out 2>&1`**)

   View the `task0.out` file and compare the results with those in the previous 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 (with “#”) 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

There is a graphical front-end for Ccaffeine (known as ccafe-gui, or "the GUI") which provides a fairly simple visual programming metaphor for the assembly of applications using CCA components. If you've been through the previous sections of this chapter, you'll already recognize that using the GUI is entirely optional. As with many environments that offer both graphical and non-graphical interfaces, we find that new/inexperienced users tend to like the GUI, while once they "get the hang" of using the CCA, they tend to prefer text-based scripting, as in Section 2.2, " Running Ccaffeine Using an `rc` File ".

In this section, we'll present the same sequence of operations as in Section 2.1, "A CCA Application in Detail" and Section 2.2, " Running Ccaffeine Using an `rc` File " using the GUI. We'll focus on the mechanics of using the GUI and assume that you've work through (or at least read) the previous sections to understand what's going on in the Ccaffeine instance running behind the GUI, and in the CCA components within Ccaffeine.

2.3.1. Running Ccaffeine with the GUI

Ccaffeine and its GUI are run as two separate processes, possibly on two different machines. In any event, you'll need two separate terminal sessions to control and monitor the two processes. We will refer to these as "*Ccaffeine host*" and "*GUI host*".

In this exercise, we will invoke Ccaffeine on the *Ccaffeine host* with:

```
gui-backend.sh --port 3314 \ ❶
             --ccafe-rc rc_file \ ❷
```

❶ This tells Ccaffeine the port number to expect the GUI to connect to. Typically, it can be any port number between 1025 and 65535 *that doesn't conflict with another application wanting to use the same port*. In this Guide, we will use port 3314, but you can change this if it is problematic.

Warning

If you're working in a setting in which there may be more than one person using Ccaffeine on the same system, *you must choose different ports, or you will conflict!* Choosing anything besides the default port, the chances of a conflict are small.

If you're participating in an organized CCA tutorial, we'll assign you a port number to use for the GUI as part of your account information as a simple way to insure there are no conflicts. *Please use your assigned port number* in this case!

❷ An `rc` file is required in order to set the component search path and do the **repository get-global** commands to load the required components into the *palette*. In the tutorial, `rc` files are automatically generated for use with the GUI. They are `$TUTORIAL_SRC/components/examples/taskn_gui_rc`. You may find it convenient to copy them into your working directory rather than retyping the complete path every time you need to reference them. So for the first exercise, you should copy `$TUTORIAL_SRC/components/examples/task0_gui_rc` to your working directory and use it as the argument for the **--ccafe-rc** argument.

The default for **ccafe-client** (and **ccafe-batch**) is to direct most of their own output, as well as the output from applications within them, to files named `pOutn` (for the stdout stream) and `pErrn` (for the stderr stream), where *n* denotes the MPI rank of the process. (In this Guide, we'll be running sequentially, so we'll have only `pOut0` and `pErr0`.) The **gui-backend.sh** script invokes **ccafe-client** with the `--ccafe-io2tty` to direct its output to the terminal instead of to the files (the files will still be created, but will contain just a few startup messages relating to the MPI rank and the process id). Using `--ccafe-io2tty` is more convenient for more interactive development work, but if you're going to run an actual application, you probably want to capture the output in the files instead. With the current **gui-backend.sh**, this would require modifying the script, but for “production” computing with CCA, we expect most people to use the `rc` approach of the previous section rather than the GUI.

The Ccaffeine GUI is implemented in Java, and is available as a `jar` file that can be used with any recent version of the Java runtime or the full software development kit. Because the Java invocation is long and hard to remember, we provide convenience scripts to simplify using it. Which one you need depends on your circumstances:

gui.sh — This is the script to use if you're running the GUI on the *same machine* as the backend. This script is configured and installed as part of the CCA tools build process.

simple-gui.sh — This is the script to use if you're running the GUI on a *Linux-like machine* and want to connect to Ccaffeine running *remotely*. You will need to download this to your local machine, along with the GUI's `jar` file, following the directions in Section B.2, “The Ccaffeine GUI”.

simple-gui.bat — This is the script to use if you're running the GUI on a *Windows machine* and want to connect to Ccaffeine running *remotely*. You will need to download this to your local machine, along with the GUI's `jar` file, following the directions in Section B.2, “The Ccaffeine GUI”.

Below, we will refer to **simple-gui.sh**, but you should replace it with whichever command is appropriate for your situation.

Note

While the GUI can be run remotely, using the X11 protocol to display on your local X11 server, this is generally unacceptably slow because of the way Java handles graphics in X11. You will probably get more satisfactory performance if you can run the GUI on your local system and allow it to connect over the network to the remote host where you're running Ccaffeine. Tunneling the GUI-Ccaffeine connection over your ssh connection is a straightforward way to deal with firewalls that often prevent direct access to most ports on remote hosts. It has the added advantage, for the purposes of this Guide, that you would use the same arguments to invoke the GUI running remotely through a tunnel or locally on the same machine as the backend. For more information, see Appendix A, *Remote Access for the CCA Environment* and in particular Section A.3, “Tunneling other Connections through SSH”.

In this exercise, we will invoke the GUI on the *GUI host* with:

```
simple-gui.sh --builderPort 3314 \ ❶
        --host localhost ❷
```

❶ This tells the GUI which port to use for the connection to *Ccaffeine host*. In general, it should match the **ccafe-client** `--port` option (though when tunneling the connection through ssh, that need not be the case).

❷ This tells the GUI which host to connect to for the Ccaffeine backend. In general, it should be the *Ccaffeine host* (though when tunneling the connection through ssh, it would be `localhost`).

Note

Since **gui.sh** is designed to be used on the same machine as Ccaffeine is running on, it does not take a **--host** argument. The **simple-gui** scripts do require it.

Note

Ccaffeine should be running and ready to receive the GUI's connection before you start the GUI. If you're scripting their execution, especially on the same machine, the **sleep** command can help build in a few seconds of delay.

Note

Once the GUI displays on your screen, it may take a few more seconds before it will respond to user actions.

2.3.2. Assembling and Running an Application Using the GUI

1. Run **gui-backend.sh** on the *Ccaffeine host* using the appropriate port and the `task0_gui_rc` `rc` file. The command will look something like:

   ```
   gui-backend.sh --port 3314 --ccafe-rc $TUTORIAL_SRC/components/examples/task0_g
   ```

   depending on the port number assigned to you, and whether or not you've copied the `rc` file to your local directory.

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

   ```
   (Ccaffeine host)
   (19419) CmdLineClientMain.cxx: execName is ccafe-client
   (19419) CmdLineClientMain.cxx: runType is CLIENT
   (19419) CmdLineClientMain.cxx: If execName is unexpected, blame your MPI startu
   (19419) CmdLineClientMain.cxx: If MPI_Init is unwanted, try adding the switch '
   (19419) CmdLineClientMain.cxx: or try setenv CCAFE_USE_MPI 0 .
   (19419) CmdLineClientMain.cxx: MPI_Init being called.
   (19419) CmdLineClientMain.cxx: MPI_Init succeeded.
   (19419) my rank: 0, my pid: 19419
   (19419) MapEverythingToFile(pOut0, pErr0)
   Type: Server
   ```

which is similar to what you saw running **ccafe-single** in Step 1 of Section 2.1, “A CCA Application in Detail”, except that in **ccafe-client**, MPI is configured “on” by default. The next-to-last line is special to **ccafe-client** (and **ccafe-batch**) and serves as a reminder that by default, the stdout and stderr output streams from these executables are funneled to the indicated files. This message appears (and the files are created, but with minimal content) even though we used the `--ccafe-io2tty` option.

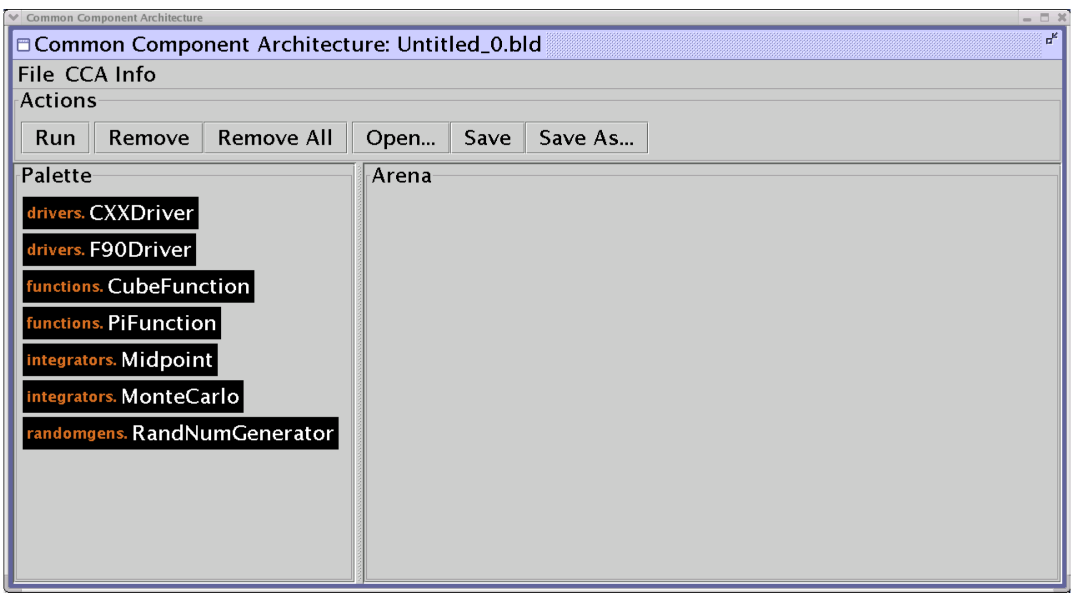
2. Run the **simple-gui.sh** on the *GUI host*.

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

```
(Ccaffeine host)
CCAFFEINE configured with babel.
CmdLineClient parsing ...

CmdContextCCAMPI::initRC: Found /san/homedirs/bernhold/task0_gui_rc.
# There are allegedly 31 classes in the component path
```

   Finally, you should see a “`gui>`” prompt in the *GUI host* terminal window, and the GUI itself should have appeared on your display, looking something like this:



Tip

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

As mentioned above, the `task0_gui_rc` sets up the **path** and performs the appropriate **repository get-global** commands. (Note that in this case, all of the available components have been included, for convenience, whereas in the previous command-line based procedures, the **repository get-global** commands appeared in two different places.)

3. At this point, you've completed the equivalent of the first three steps in Section 2.1, “A CCA Application in Detail” (as well as the additional **repository get-global** commands in Step 7) or in the `rc` file described in Section 2.2, “ Running Ccaffeine Using an `rc` File ”.

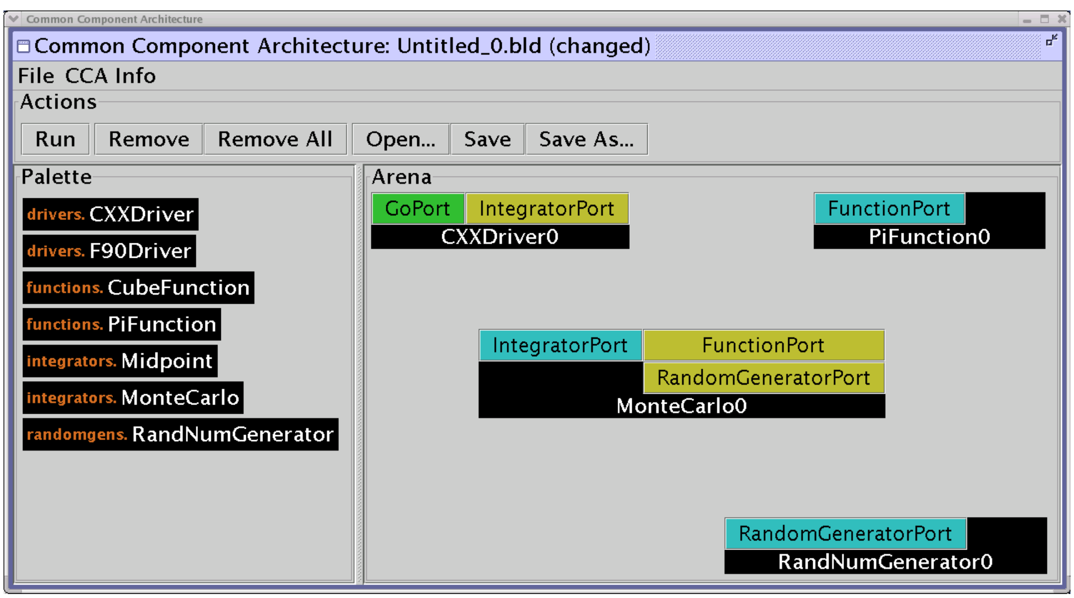
   For the remainder of this procedure, each step will be functionally equivalent to the matching step of the command-line based procedures described above. Please refer to those sections for more detailed explanations of what is happening “behind the scenes”.

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

   For the first application, you should instantiate

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

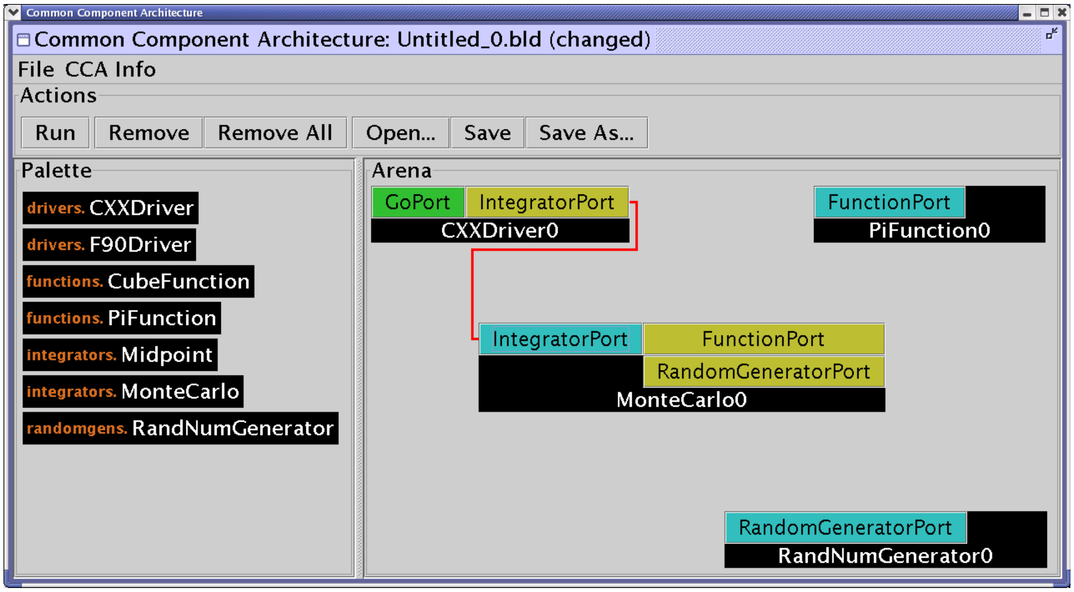
   (you may notice some debugging messages in the *GUI host* terminal window as you do this), and your GUI should look something like this:



Tip

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

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



Tip

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

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

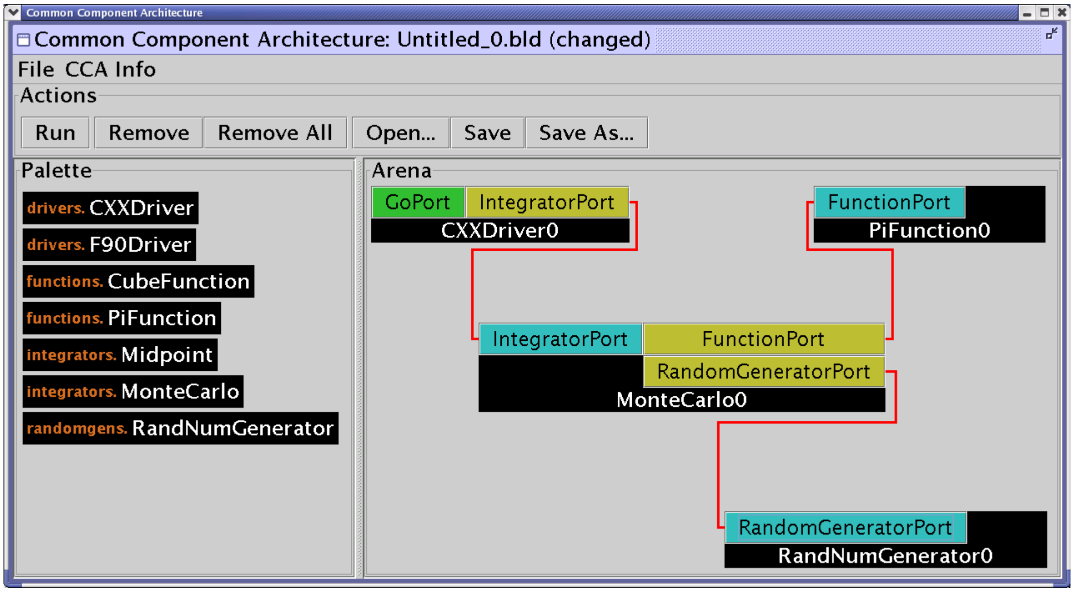
Note

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

Complete the first application by making the following connections:

- `CXXDriver0`'s `IntegratorPort` to `MonteCarlo0`'s `IntegratorPort`
- `MonteCarlo0`'s `FunctionPort` to `PiFunction0`'s `FunctionPort`
- `MonteCarlo0`'s `RandomGeneratorPort` to `RandNumGenerator0`'s `RandomGeneratorPort`.

At this point, your GUI should look something like:

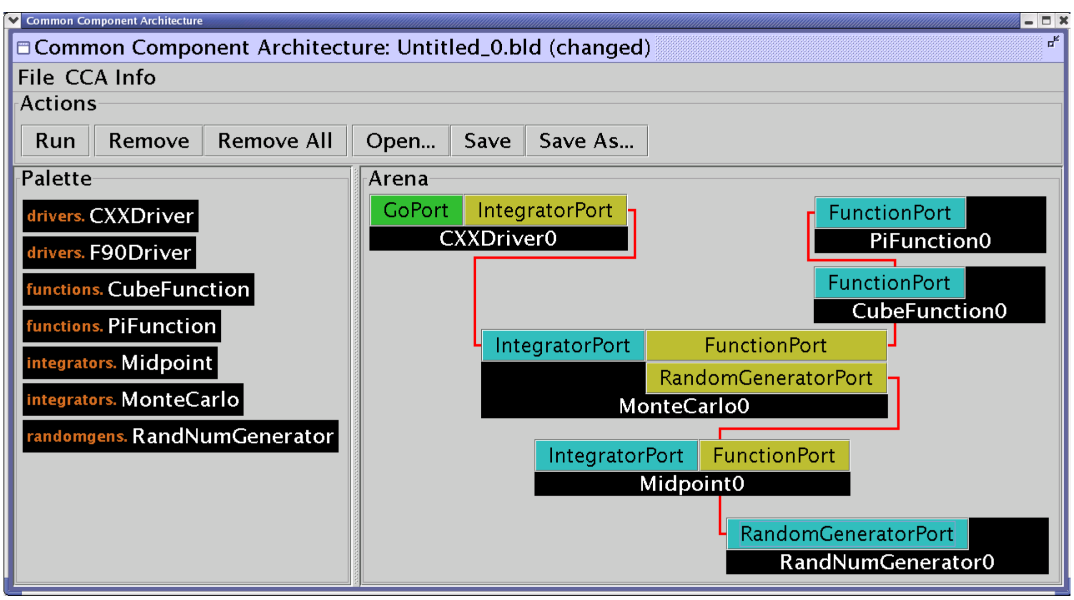


6. The application is now fully assembled and is ready to run. If you click-and-release the `GoPort` button on the `CXXDriver0` component, you should see the result appear in the *Ccaffeine host* terminal, "`Value = 3.141449`" and the message "`##specific go command successful`" in the *GUI host* terminal.

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

   - `integrators.Midpoint` and
   - `functions.CubeFunction`

   components. Since they're already in the *palette*, you can instantiate them in the same way as Step 4.



Tip

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

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

8. Next, we break the port connections we don't need so we can reconnect to the new components. Right-click on the `IntegratorPort` (either the *user* or the *provider*) and a dialog box will pop up asking you to confirm that you want to break the connection. You will need to break the following connections:

   - `CXXDriver0`'s `IntegratorPort` to `MonteCarlo0`'s `IntegratorPort`
   - `MonteCarlo0`'s `FunctionPort` to `PiFunction0`'s `FunctionPort`

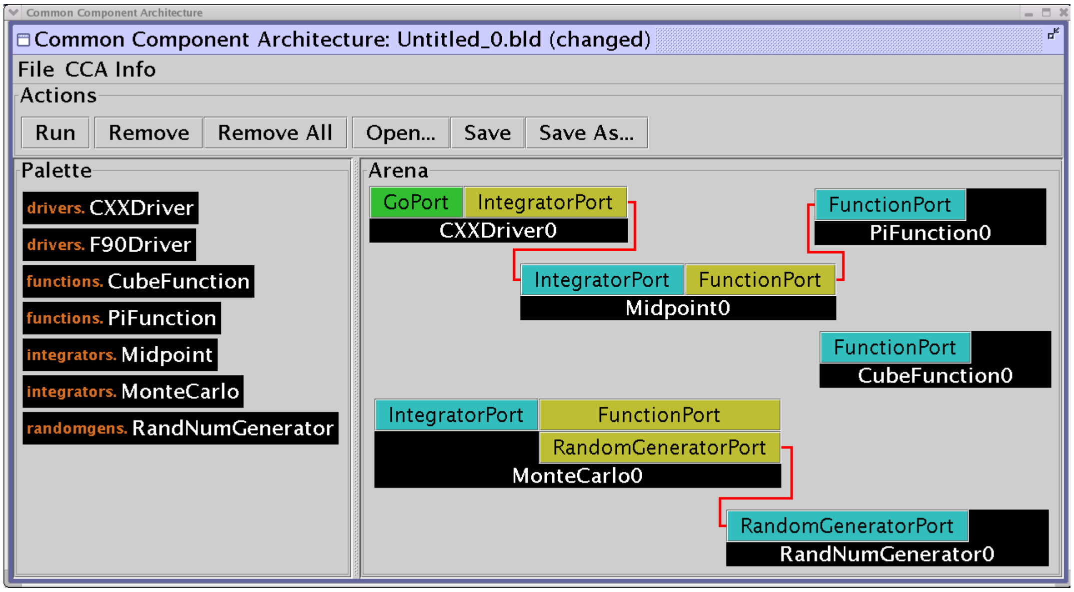
   The fact that `MonteCarlo0` remains connected to `RandNumGenerator0` is immaterial because neither component will be used in the remainder of this exercise.

   ### Note

   Step 7 and Step 8 could have been done in either order.

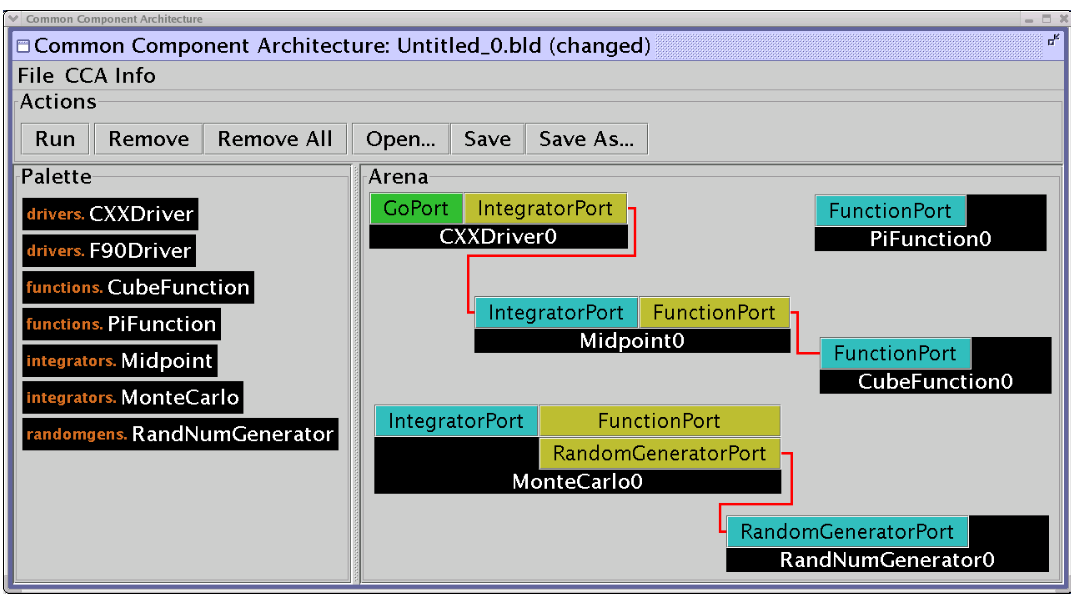
9. Assemble the new application by making the following connections:

   - `CXXDriver0`'s `IntegratorPort` to `Midpoint0`'s `IntegratorPort`
   - `Midpoint0`'s `FunctionPort` to `PiFunction0`'s `FunctionPort`



Click-and-release the `GoPort` button on the `CXXDriver0` component, you should see the result appear in the *Ccaffeine host* terminal, "`Value = 3.141553`" and the message "`##specific go command successful`" in the *GUI host* terminal.

10. Finally, create a third application by replacing `PiFunction0` with `CubeFunction0`. When you click on the `GoPort` you should get "`Value = 0.250010`" in the *Ccaffeine host* terminal.



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

Tip

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

2.3.3. Notes on More Advanced Usage of the GUI

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

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

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

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

$Revision: 1.45 $
$Date: 2006/08/22 22:09:49 $

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` (defined in `$STUDENT_SRC/ports/sidl/integrator.sidl`). 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*.)

Important

This and subsequent exercises use the `student-src` code tree instead of the `tutorial-src` code tree. The difference is that in the `student-src` we have deleted a bunch of components which you will recreate in these exercises. You can always refer to the completed versions in the `tutorial-src` to see what the final results should look like.

You'll need to *build* the `student-src` code tree in your own directory before you can proceed, by following the instructions in Appendix C, *Building the Tutorial and Student Code Trees*.

If you are participating in an organized tutorial your account information handout will tell you where you can obtain the tar file on the system you're using instead of having to download it.

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 “standard” interfaces.

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. View *CCA_TOOLS_ROOT*/`share/cca-spec-babel-0_8_0-babel-1.0.0/cca.sidl`. First, notice the package declarations at the beginning of the file:

   ```
   package gov {
   package cca version 0.8.0 {
   ...
   ```

   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 (though it is common to have multiple implementations that do things in somewhat different ways, but with the same result).

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 component you're adding (the `drivers.sidl` is already listed there along with other `.sidl` files).

   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 sidl/unitdrivers.sidl sidl/library.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, cxx, f90.
# For example:
# COMPONENTS = drivers.F90Driver-f90 drivers.CXXDriver-cxx
COMPONENTS = drivers.PYDriver-python functions.PiFunction-cxx \
```

```
library.CxxUnitsLibraryComp-cxx library.PyUnitsLibraryComp-python \
undrivers.PyDriver-python \
functions.LinearFunction-c integrators.MonteCarlo-f90 \
randomgens.RandNumGenerator-cxx  integrators.Trapezoid-cxx \
        integrators.Simpson-f77\
        drivers.CXXDriver-cxx
```

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.

Important

Before proceeding, you need to be sure that you have done the initial build of the `student-src`, following the directions in Appendix C, *Building the Tutorial and Student Code Trees*.

6. When it is processing a `.sidl` file, Babel needs to be able to resolve external references contained within the file (for example, to `gov.cca.Component`, or to other ports, etc.). The simplest way to do this is to have Babel collect all of the information it needs in a *repository*. The build system for the tutorial is designed to do this, so at this point, we need to add the new `drivers.sidl` file to the repository.

   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/cca/cca-tools_gcc_intelF90_PIC/bin/babel -t xml -R../xml_repository \
    -R/san/cca/cca-tools_gcc_intelF90_PIC/share/ \
     cca-spec-babel-0_8_0-babel-1.0.0/xml  \
     -o ../xml_repository sidl/functions.sidl sidl/integrators.sidl \
     sidl/randomgens.sidl sidl/drivers.sidl sidl/unitdrivers.sidl \
     sidl/library.sidl
Babel: Parsing URL "file:/san/homedirs/bernhold/student-src/ \
        components/sidl/functions.sidl".
Babel: Parsing URL "file:/san/homedirs/bernhold/student-src/ \
        components/sidl/integrators.sidl".
Babel: Parsing URL "file:/san/homedirs/bernhold/student-src/ \
        components/sidl/randomgens.sidl".
Babel: Parsing URL "file:/san/homedirs/bernhold/student-src/ \
        components/sidl/drivers.sidl".
Babel: Parsing URL "file:/san/homedirs/bernhold/student-src/ \
        components/sidl/unitdrivers.sidl".
Babel: Parsing URL "file:/san/homedirs/bernhold/student-src/ \
        components/sidl/library.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-cxx`**. The output should look something like this:

   ```
   ### Generating a cxx implementation for the drivers.CXXDriver component.
   /san/cca/cca-tools_gcc_intelF90_PIC/bin/babel -s cxx -R../xml_repository \
      -R/san/cca/cca-tools_gcc_intelF90_PIC/share/cca-spec-babel-0_7_8-babel-0.10.
       -g -u -E -l -m drivers.CXXDriver. --suppress-timestamp drivers.CXXDriver
   Babel: Resolved symbol "drivers.CXXDriver"...
   touch .drivers.CXXDriver-cxx
   ```

   and in the `$STUDENT_SRC/components/drivers/cxx` directory, you should see the following files:

   ```
   drivers.CXXDriver.babel.make
   drivers_CXXDriver_Impl.cxx
   drivers_CXXDriver_Impl.hxx
   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 C++, both a source file (`.cxx`) and the corresponding header file (`.hxx`) are generated.

2. In your editor, take a look through both `$STUDENT_SRC/components/drivers/cxx/drivers_CXXDriver_Impl.cxx` and `$STUDENT_SRC/components/drivers/cxx/drivers_CXXDriver_Impl.hxx` to familiarize yourself with their structure before you make any changes.

   a. Near the top of `drivers_CXXDriver_Impl.hxx`, you will see a group of include directives:

      ```
      ...
      //
      // Includes for all method dependencies.
      //
      #ifndef included_drivers_CXXDriver_hxx
      #include "drivers_CXXDriver.hxx"
      #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.cxx`, 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 (with a default method body that throws a "method not implemented" exception). 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. You will obviously need to delete the babel-generated code that throws the exception (or comment it out), and replace it with the code that actually implements the method under consideration.

3.2.1. The `setServices` Implementation

1. We'll begin by implementing the `setServices` method in `drivers_CXXDriver_Impl.cxx`. 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 = (*this); ❷

  frameworkServices.addProvidesPort(gp, ❷
                                     "GoPort",
```

```
                                                  "gov.cca.ports.GoPort",
                                                  frameworkServices.createTypeMap());

  // Use an IntegratorPort port
  frameworkServices.registerUsesPort ("IntegratorPort", ❸
                                                    "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 `.hxx` 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, the babel C++ binding allows the enclosing object to be accessed through the standard C++ *this mechanism (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.

❸ 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`, 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/cxx/drivers_CXXDriver_Impl.hxx` and add the following:

```
...
  /**
   * Symbol "drivers.CXXDriver" (version 1.0)
   */
  class CXXDriver_impl : public virtual ::drivers::CXXDriver
  // DO-NOT-DELETE splicer.begin(drivers.CXXDriver._inherits)
  // Insert-Code-Here {drivers.CXXDriver._inherits} (optional inheritance here)
  // DO-NOT-DELETE splicer.end(drivers.CXXDriver._inherits)
  {
  // All data marked protected will be accessable by
  // descendant Impl classes
  protected:

    bool _wrapped;


    // DO-NOT-DELETE splicer.begin(drivers.CXXDriver._implementation)
    // Insert-Code-Here {drivers.CXXDriver._implementation} (additional details

    ::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.hxx"
#include "gov_cca_ports_GoPort.hxx"

// 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/cxx`.

   Then, change directories to `$STUDENT_SRC/components/drivers/cxx` 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/cxx/drivers_CXXDriver_Impl.cxx` 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; (1)
  int count = 100000;
  double lowerBound = 0.0, upperBound = 1.0;

  ::integrator::IntegratorPort integrator; (2)

  // get the port ...
  gov::cca::Port port = frameworkServices.getPort("IntegratorPort");
  integrator = babel_cast< ::integrator::IntegratorPort >(port);
(2)

  if(integrator._is_nil()) { (3)
    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 `integrator.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 `re-`

`gisterUsesPort` invocation. Note the use of the `babel_cast` method to cast from the `gov::cca::Port` to its sub-type `::integrator::IntegratorPort`.

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/cxx`. 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. 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/software.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 31 characters, the 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/cca/cca-tools_gcc_intelF90_PIC/bin/babel -s f90 -R../xml_repository \
 -R/san/cca/cca-tools_gcc_intelF90_PIC/share/cca-spec-babel-0_8_0-babel-1.0.0/x
 -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/cxx/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 31 character limit if necessary), however it should be fairly easy to match them up with corresponding SIDL names. Furthermore, Babel also generates a "default" method implementation that throws a "method not implemented" exception. You will need to delete the exception-throwing code (or comment it out) before adding code that actually implements the method under consideration. 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
  use sidl_NotImplementedException
  use gov_cca_CCAException
  use gov_cca_Services
  use sidl_BaseInterface
  use sidl_RuntimeException
  use drivers_F90Driver
  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, excpt)

  ! 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, excpt) 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, excpt) 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.

❸ 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`

❹ 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 `checkExceptionDriver` method in Step 2. Note also that for SIDL methods that do not explicitely throw an exception, Babel adds an extra argument `exception` of type sidl_BaseInterface to the method's argument list.

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.

❺ 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 F90 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, exception)
  use sidl
  use sidl_NotImplementedException
  use sidl_BaseInterface
  use sidl_RuntimeException
  use drivers_F90Driver
  use drivers_F90Driver_impl
  ! DO-NOT-DELETE splicer.begin(drivers.F90Driver.go.use)
  ! Insert use statements here...

  use sidl_BaseInterface (1)

  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
  type(sidl_BaseInterface_t) :: exception ! 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 (kind=sidl_double) :: lowBound ❸
  real (kind=sidl_double) :: upBound
  integer (kind=sidl_int) :: count
  real (kind=sidl_double) :: value

  ! Initialize local variables
  count = 100000 ❸
  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, & ❷
       'IntegratorPort', generalPort, excpt)
  call checkExceptionDriver(excpt, & ❹
       '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, excpt) ❷

  call checkExceptionDriver(excpt, 'cast(generalPort, integratorPort, excpt)')
  if (not_null(integratorPort)) then ❹
     value = -1.0 ! nonsense number to confirm it is set

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

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

  retval = 0 ❼
  return

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

❶ 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 `checkExceptionDriver` 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
type(sidl_BaseInterface_t) :: throwaway
if (not_null(excpt)) then
write(*, *) 'drivers.F90Driver Exception: ', msg
call deleteRef(excpt, throwaway)
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 specification `cca.sidl`, 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 0.8.0 {
...
        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 `task1_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: 1.47 $
$Date: 2006/08/03 19:45:32 $

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 small, the techniques can be 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` to store its state. The complete code for the legacy integrator follows.

```
file: $STUDENT_SRC/legacy/f90/Integrator.f90
module Integrator
  use FunctionModule                                              (1)
  implicit none

contains

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

    ! !INPUT PARAMETERS:

    type(FunctionParams_t),intent(IN) :: functionParams          (2)
    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)                        (3)
       sum = sum + func_val
    end do

    integrate_mp = sum * h

  end function integrate_mp

end module Integrator
```

Notes on the `Integrator.f90` file

1. 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.
2. The `functionParams` argument of the integrator is the only way function parameters can be passed through to the function being evaluated.

❸ This evaluates the function given the parameters passed into the `Integrator`.

4.2. The `FunctionModule` wrapper.

To enable the legacy integrator to evaluate functions that are not defined in the same fashion as the `FunctionModule` above (i.e., such that they define the `eval` method or equivalent interface that takes a FunctionParams_t argument and a real argument) is to create another `FunctionModule` that allows a `FunctionPort` to be used for the function evaluation.

file: $STUDENT_SRC/legacy/f90/FunctionModuleWrapper.f90

```
module FunctionModule

  ! This module replaces the FunctionModule used by the legacy integrator.
  ! Thus, we need to makes sure that this module is first in the module
  ! search path when building the integrator component.

  ! We need to include the function port definitions
  use function_FunctionPort_type                                    ❶
  use function_FunctionPort

  implicit none

  type FunctionParams_t
     sequence              ! required for component version
     type(function_FunctionPort_t) funcPort                        ❷
  end type FunctionParams_t

  interface eval                                                    ❸
     ! This is the one called by the legacy Integrator
     module procedure evalFunction
  end interface

contains

  subroutine setFunctionPort(params, port)
    type(FunctionParams_t),        intent(OUT) :: params
    type(function_FunctionPort_t), intent(IN)  :: port

    params%funcPort = port
  end subroutine setFunctionPort

  real function evalFunction(params, x)
    use function_FunctionPort
    ! input parameters:
    type(FunctionParams_t), intent(IN) :: params
    real,                   intent(IN) :: x

    ! local variablesreal
    real (selected_real_kind(15, 307)) :: xx
    real (selected_real_kind(15, 307)) :: retval

    ! Compute value by calling the function evaluation in FunctionModule
    xx = x
    call evaluate(params%funcPort, xx, retval)                      ❹
    evalFunction = retval

  end function evalFunction

end module FunctionModule
```

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, the entire tutorial tree was built (see Appendix C, *Building the Tutorial and Student Code Trees*), 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 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. 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 Section 4.3, “The `integrator.IntegratorPort` Definition” (and defined in `$STUDENT_SRC/ports/sidl/function.sidl`).

   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, cxx, 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-cxx \
        integrators.MonteCarlo-f90 randomgens.RandNumGenerator-cxx \
        drivers.F90Driver-f90 drivers.CXXDriver-cxx  \
         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 `$STUDENT_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)

private :: wrapObj_s

interface wrapObj
module procedure wrapObj_s
end interface
                                                                      (3)
type integrators_Midpoint_priv
  sequence
  ! DO-NOT-DELETE splicer.begin(integrators.Midpoint.private_data)    (4)

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

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

  ! 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

1. 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.
3. This component stores a handle to the framework's Services object, equivalently to the way the Driver component was implemented in Step 2.
2. The legacy `Integrator` module is included.
4. 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 sidl
  use sidl_NotImplementedException
  use sidl_BaseInterface
  use sidl_RuntimeException
  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 sidl
  use sidl_NotImplementedException
  use sidl_BaseInterface
  use sidl_RuntimeException
  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...

  type(SIDL_BaseInterface_t) :: throwaway
  ! 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, throwaway)
  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 name mangling algorithm attempts to preserve the most significant part of the name to help identifying it.

```
...
recursive subroutine Midpoi_setServices6_m9htaw4m_mi(self, services,          &
  exception)
  use sidl
  use sidl_NotImplementedException
  use gov_cca_CCAException
  use gov_cca_Services
  use sidl_BaseInterface
  use sidl_RuntimeException
  use integrators_Midpoint
  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
  type(SIDL_BaseInterface_t) :: throwaway
  ! 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, throwaway)

  ! 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, throwaway)

  ! 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, throwaway)

! 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 sidl
```

```
  use sidl_NotImplementedException
  use sidl_BaseInterface
  use sidl_RuntimeException
  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 (kind=sidl_double) :: lowBound ! in
  real (kind=sidl_double) :: upBound ! in
  integer (kind=sidl_int) :: count ! in
  real (kind=sidl_double) :: retval ! out
  type(sidl_BaseInterface_t) :: exception ! out

! DO-NOT-DELETE splicer.begin(integrators.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, excpt)
        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
  type(SIDL_BaseInterface_t) :: throwaway
  if (not_null(excpt)) then
    write(*, *) 'integrators.Midpoint Exception: ', msg
    call deleteRef(excpt, throwaway)
  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`:

```
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: 1.26 $
$Date: 2006/08/23 16:21:28 $

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 (defined in `$STUDENT_SRC/ports/sidl/function.sidl`) 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. If you're looking for a C language example, try the `functions.LinearFunction`.

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, cxx, or f90. In this case, the name is functions.CubeFunction, and the language is cxx. The updated value of `COMPONENTS` should look

like this:

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

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

3. In the `$STUDENT_SRC/components` directory, run **make .repository**. This will re-generate the XML representation of the SIDL component class definitions (including the newly added class `CubeFunction` and store them in the `$STUDENT_SRC/xml_repository` directory so that all references can be easily resolved.

   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/cca/cca-tools_gcc_intelF90_PIC/share/cc
Babel: Parsing URL "file:/san/homedirs/bernhold/student-src/components/sidl/dri
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:/san/homedirs/bernhold/student-src/components/sidl/fun
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:/san/homedirs/bernhold/student-src/components/sidl/int
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:/san/homedirs/bernhold/student-src/components/sidl/ran
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-cxx** to generate the C++ server-side binding for the component class `functions.CubeFunction`. The output from this step should look something like this:

```
### Generating a cxx implementation for the CubeFunction component
babel -s cxx -R../xml_repository -R/san/cca/cca-tools_gcc_intelF90_PIC/share/cc
-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/cxx` should contain two additional files, `functions_CubeFunction_Impl.cxx` and `functions_CubeFunction_Impl.hxx` 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.hxx` in the directory `$STUDENT_SRC/components/functions/cxx`. 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 `frameworkServices`. Upon completion, this splicer block should look like this:

```
...
    // DO-NOT-DELETE splicer.begin(functions.CubeFunction._implementation)
    // Insert-Code-Here {functions.CubeFunction._implementation} (additional de
    gov::cca::Services    frameworkServices;
    // DO-NOT-DELETE splicer.end(functions.CubeFunction._implementation)
...
```

2. Edit the file `functions_CubeFunction_Impl.cxx` in the directory `$STUDENT_SRC/components/functions/cxx` 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-Code-Here {functions.CubeFunction.setServices} (setServices method)


   frameworkServices = 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 = (*this);      //  Babel required casting
   if(p._is_nil()) {
      fprintf(stderr,
              "Error:: %s:%d: Error casting (*this) to gov::cca::Port \n",
              __FILE__, __LINE__);
      exit(1);
   }

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

   //  Babel required casting
   gov::cca::ComponentRelease cr =
             ::babel_cast< gov::cca::ComponentRelease >(*this);
```

```
     services.registerForRelease(cr);
     return;

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

3. 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-Code-Here {functions.CubeFunction.evaluate} (evaluate method)
     return x*x*x;
   // DO-NOT-DELETE splicer.end(functions.CubeFunction.evaluate)
...
```

4. To build the newly written component into a usable library, type **make** in the directory `$STUDENT_SRC/components/functions/cxx`. This will compile, link, and install the new component into a library that is installed in the directory `$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`:

```
genSCLCCA.sh cca `pwd`/libfunctionsCubeFunction-cxx.so \
   functions.CubeFunction cubeFunction dynamic private now \
   > functions.CubeFunction.cca
```

5.4. Using Your New Component

1. 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.

2. Run the script with **`ccafe-single --ccafe-rc task3_rc`**. It should run without errors and give you a result of `Value = 0.250010`.

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 6. Using TAU to Monitor the Performance of Components

$Revision: 1.12 $
$Date: 2006/08/20 21:51:15 $

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

We will start by create a proxy component for the `integrator.IntegratorPort`. Note that you only need to have completed Chapter 3, *Sewing CCA Components into an Application: the Driver Component* in order to follow these instructions. Though the proxy will be implemented in C++, it can proxy for components implemented in any language.

Warning

The following instructions assume that you chose to implement the `drivers.CXXDriver` rather than the `drivers.F90Driver`. If you implemented the `drivers.F90Driver`, you will need to edit `task4_rc` to reflect this.

6.1. Creating the Proxy Component

1. Edit the file `$STUDENT_SRC/components/sidl/integrators.sidl` and make the following addition:

```
package integrators version 1.0 {

 class MonteCarlo implements 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.CCAExcept

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

  class Trapezoid implements-all integrator.IntegratorPort,
                gov.cca.Component,
                gov.cca.ComponentRelease
  {
  }
  class Simpson implements-all integrator.IntegratorPort,
                gov.cca.Component,
                gov.cca.ComponentRelease
  {
```

```
  }

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

This will give us a new component, called `IntegratorProxy` that implements the `integrator.IntegratorPort`.

2. Edit `$STUDENT_SRC/components/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, cxx, f90.
# For example:
# COMPONENTS = drivers.F90Driver-f90 drivers.CXXDriver-cxx
COMPONENTS = functions.PiFunction-cxx \
        integrators.MonteCarlo-f90 randomgens.RandNumGenerator-cxx \
        drivers.CXXDriver-cxx integrators.IntegratorProxy-cxx
```

3. In the `$STUDENT_SRC/components` directory, type **`make .integrators.IntegratorProxy-cxx`** to rebuild the repository. The output should look something like this:

```
### Generating XML for SIDL packages containing component declarations
/san/cca/cca-tools_gcc_intelF90_PIC/bin/babel -t xml -R../xml_repository \
    -R/san/cca/cca-tools_gcc_intelF90_PIC/share/cca-spec-babel-0_7_8-babel-0.10
    -o ../xml_repository sidl/functions.sidl sidl/integrators.sidl \
    sidl/randomgens.sidl sidl/drivers.sidl
Babel: Parsing URL "file:/san/homedirs/bernhold/\
        student-src/components/sidl/functions.sidl"...
Babel: Parsing URL "file:/san/homedirs/bernhold/\
        student-src/components/sidl/integrators.sidl"...
Babel: Parsing URL "file:/san/homedirs/bernhold/\
        student-src/components/sidl/randomgens.sidl"...
Babel: Parsing URL "file:/san/homedirs/bernhold/\
        student-src/components/sidl/drivers.sidl"...
Babel: Parsing URL "file:/san/homedirs/bernhold/\
        student-src/components/sidl/unitdrivers.sidl".
Babel: Parsing URL "file:/san/homedirs/bernhold/\
        student-src/components/sidl/library.sidl".

touch .repository

### Generating a cxx implementation for the integrators.IntegratorProxy \
    component.
```

```
    /san/cca/cca-tools_gcc_intelF90_PIC/bin/babel -s cxx -R../xml_repository \
    -R/san/cca/cca-tools_gcc_intelF90_PIC/share/cca-spec-babel-0_7_8-babel-0.10
    -g -u -E -l -m integrators.IntegratorProxy. --suppress-timestamp \
    integrators.IntegratorProxy
Babel: Resolved symbol "integrators.IntegratorProxy"...
touch .integrators.IntegratorProxy-cxx
```

6.2. Using the proxy generator

1. In the `$STUDENT_SRC/components/integrators/cxx` directory, type **`tau_babel_pg -f integrators_IntegratorProxy_Impl.cxx -h integrator_IntegratorPort.hxx -p IntegratorPort -t integrator.IntegratorPort`**

   The usage of the proxy generator is as follows:

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

   The `-h` option specifies the header file that needs to be included to use the port. Note that this is the same header file that was added to the `drivers.CXXDriver` component in order to use the `integrator.IntegratorPort`.

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

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

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

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

3. Now build the proxy by going to `$STUDENT_SRC` and running **`make`**.

6.3. Using the new proxy component

1. Change directories to `$STUDENT_SRC/components/examples` and edit `task4_rc`. This file will assemble and run an application using the new proxy component you've created.

   ### Note

   If you installed the cca tools yourself, you will need to modify `task4_rc` to reflect the location of the performance component.

2. Run the script with **`ccafe-single --ccafe-rc task4_rc`**. It should run without errors and give you a result like `Value = 3.140347` as before.

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

```
Reading Profile files in profile.*

NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------
100.0           32           32           1           0      32043 integrate \
                                              double (double, double, int32_t)
```

Further exercises: Try swapping in a different integrator. Try generating a proxy for the Function port.

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

Chapter 7. Understanding arrays and component state

$Revision: 1.9 $
$Date: 2006/08/03 19:45:32 $

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

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

Note

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

7.1. Introduction

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

```
   package arrayop version 1.0{

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

 /** Evaluate Acc = Acc + alpha A x,  where
   *   Acc     The internal accumulator maintained by implementors
   *           of this interface
   * return the result  in vector y (of size m)
   */
         int mulMatVec (in double               alpha,
                        in rarray<double, 2>    A(m, n),
                        in rarray<double, 1>    x(n),
```

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

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

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

     }
  ...
```

Note

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

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

The student source contains fully implemented three components that *provide* the `LinearOp` port. The components `F90ArrayOp`, `F77ArrayOp`, and `CArrayOp` can be found at `$STUDENT_SRC/components/arrayOps/{f90,f77,c}`. In addition, a driver component that *uses* the `LinearOp` port can be found at `$STUDENT_SRC/components/arrayDrivers/c`.

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

7.2. The `CDriver` Component

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

7.2.1. Using SIDL Raw Arrays

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

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

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

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

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

7.2.2. Using SIDL Normal Arrays

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

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

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

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

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

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

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

Direct access to underlying SIDL array memory is also available in the Babel SIDL array binding in F77, F90, and C++. Example of such use is available in the discussion in Section 7.3, “Linear Array Operations Components”.

7.3. Linear Array Operations Components

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

7.3.1. The `CArrayOp` Component

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

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

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

```
struct arrayOps_CArrayOp__data *pd = (struct arrayOps_CArrayOp__data*)
              calloc(1, sizeof(struct arrayOps_CArrayOp__data));
arrayOps_CArrayOp__set_data(self, pd);
arrayop_LinearOp_init(self);
return;
```

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

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

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

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

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

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

Note

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

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

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

7.3.2. The `F77ArrayOp` Component

Code for the `F77ArrayOp` component can be found in the directory `$STUDENT_SRC/components/arrayOps/f77`, in `Impl` file `arrayOps_f77ArrayOp_Impl.f`. Private component state is represented by entries an an array of SIDL `opaque` types. It is the responsibility of the programmer to ensure consistency of the treatment of entries in this array across method calls (this is similar to the way entries into `common blocks` are manipulated). Code for the creation and initialization of the private component state can be found in the component constructor method `arrayOps_F77ArrayOp__ctor_fi`.

```
integer *8  stateArray, intArray, tmp
tmp = 0
call sidl_opaque__array_create1d_f(3, stateArray)
call sidl_int__array_create1d_f(2, intArray)
if ((statearray .ne. 0) .and. (intArray .ne. 0)) then
  call sidl_opaque__array_set1_f(statearray, 0, tmp)
  call sidl_opaque__array_set1_f(statearray, 1, intArray)
  call sidl_opaque__array_set1_f(statearray, 2, tmp)
else
   . . .
```

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

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

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

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

```
if (accVector .eq. 0) then
   call sidl_double__array_create1d_f(m, accVector)
   call sidl_int__array_set1_f(intArray, 0, m)
   call sidl_opaque__array_set1_f(stateArray, 2, accVector)
   dblTmp = 0.0
   do i = 0, m-1
      call sidl_double__array_set1_f(accVector, i, dblTmp)
   end do
else
   . . .
```

Note

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

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

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

7.3.3. The `F90ArrayOp` Component

Code for the `F90ArrayOp` component can be found in the directory `$STUDENT_SRC/components/arrayOps/f90`, in the `Impl` files `arrayOps_F90ArrayOp_Impl.F90`and `arrayOps_F90ArrayOp_Mod.F90`. Private component state is represented by the `type arrayOps_F90ArrayOp_priv` in the file `arrayOps_F90ArrayOp_Mod.F90`

```
type arrayOps_F90ArrayOp_priv
  sequence
  ! DO-NOT-DELETE splicer.begin(arrayOps.F90ArrayOp.private_data)
  ! Handle to framework Services object
  type(gov_cca_Services_t) :: frameworkServices
  real (selected_real_kind(15, 307)), dimension(:), pointer :: myVectorP
  integer (selected_int_kind(9)) :: myVecLen
  ! DO-NOT-DELETE splicer.end(arrayOps.F90ArrayOp.private_data)
end type arrayOps_F90ArrayOp_priv
```

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

```
   type(arrayOps_F90ArrayOp_wrap) :: dataWrap
   type(arrayOps_F90ArrayOp_priv), pointer :: pd

   allocate(dataWrap%d_private_data)
   pd => dataWrap%d_private_data
   ! Allocate memory and initialize
   call set_null(pd%frameworkServices)
   pd%myVectorP => NULL()
   pd%myVecLen = 0
   call arrayOps_F90ArrayOp__set_data_m(self, dataWrap)
```

Note that private data is accessed through the pointer `pd`, accessed through the variable `dataWrap` of type `arrayOps_F90ArrayOp_wrap`. The call to the *built-in* method `arrayOps_F90ArrayOp__set_data_m` associates the newly created structure pointed to via `pd` with this instance of the component. The corresponding method `arrayOps_F90ArrayOp__get_data_m` is used to retrieve this private data for further processing.

The subroutine that implements the `mulMatVec` method uses SIDL raw arrays (note that the name of this subroutine is altered by Babel to accomodate F90 identifier length restrictions as outlined in Section 3.3, “Implementation of the `F90Driver` in Fortran 90”). SIDL raw arrays manifest themselves in F90 implementations as regular F90 arrays that use zero-based indexing.

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

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

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

Note

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

7.4. Assignment: `NonLinearOp` Component and Driver

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

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

/** Evaluate Acc = Acc + alpha log(A) where
  *   log(A)  Elementwise log (base 10) of array A
  *   Acc     The internal accumulator maintained by implementors
  *           of this interafce
  * return the result  in array R
  */
```

```
    int logMat (in double                alpha,
                in rarray<double, 2>     A(m, n),
                inout rarray<double, 2>  R(m, n),
                in int                   m,
                in int                   n);

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

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

1. **Adding SIDL Specification**

   a. Edit the file `$STUDENT_SRC/components/sidl/arraydrivers.sidl` to add specification for the nonlinear matrix operations driver.

   ```
   class NLinearDriver implements-all gov.cca.ports.GoPort,
                                      gov.cca.Component,
                                      gov.cca.ComponentRelease
   {}
   ```

   b. Edit the file `$STUDENT_SRC/components/sidl/arrayops.sidl` to add specification for the nonlinear matrix operations component.

   ```
   class NLinearOp implements-all arrayop.NonLinearOp,
                                  gov.cca.Component,
                                  gov.cca.ComponentRelease
   {}
   ```

2. **Adding your new components to the build system**

   Edit the file `$STUDENT_SRC/components/MakeIncl.components` to add the specification of the two new components to the list of components in the `COMPONENTS` macro. The new entries will be of the form **`arrayDrivers.NLinearDriver-XX`** and **`arrayOps.NLinearOp-YY`**, where **`XX`** and **`YY`** are the language(s) you will use to implement the components (lower case “c”, “cxx”, “f77”, “f90”, or “python”).

3. **Generating code for the new components**

   Run **`make`** in the directory `$STUDENT_SRC/components` to generate `Impl` files and Babel glue code for the newly added components. Note that this code may be generated in the same language subdirectory that contains code for the `LinearOp` driver or components, if you choose the same language(s) for your new components.

4. **Editing Implementation Files**

   Edit the newly generated `Impl` files to implement the methods in the driver and the `NonLinearOp` component. Build the new components (by running **`make`** in the directory where the `Impl` files are generated).

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

   You can run the application using one of the techniques outlined in Chapter 2, *Assembling and Running a CCA Application*. Note that you will need to assign matching port types in the driver and the component.

Chapter 8. Understanding objects and passing modes

$Revision: 1.7 $
$Date: 2006/08/03 19:45:32 $

This exercise focuses on the use of objects in a component interface and the subtleties of parameter passing modes. You will be working with a units of measurement library component that can perform unit conversions, for example, convert a quantity in meters to an equivalent quantity in inches. The component's design and implementation are simplified for the sake of this example; it is not intended for real use.

Note

This exercise is self-contained. Components and ports utilized in this example do not rely on components or ports from other exercises.

8.1. The unit library

The unit library involves the following SIDL types:

```
package units version 1.0 {
  interface UnknownUnitException extends sidl.BaseException { }

  interface Unit {
    string name();
    void conversionFactors(out double slope, out double offset);
  }

  interface Conversion {
    Unit convertFrom();
    Unit convertTo();
    double convert(in double orig);
  }

  interface UnitsLibrary extends gov.cca.Port
  {
    Conversion lookupConversion(in Unit src, in Unit dest);
    void defineUnit(in string name,
                    in double slope,
                    in double offset,
                    in  Unit  knownUnit,
                    out Unit  newUnit);
    void invertConversion(inout Conversion convert);
    Unit lookupUnit(in string name) throws UnknownUnitException;
  }
}
```

This SIDL defines one CCA port and three helper interfaces. CCA ports should only reference SIDL interfaces — not classes; otherwise, the client code becomes tightly coupled to the helper class implementation.

The `conversionFactors` in the interface `units.Unit` is an excellent example of when *`out`* is appropriate. Because two values must be returned, *`out`* is the appropriate mode for *`slope`* and *`off-`*

set. The use of *out* for *newUnit* in `defineUnit` is somewhat contrived for the sake of example. Normally, one would use a return value instead of using a single *out* argument.

Now an imaginary, overworked, hapless scientific software developer saw this port definition and thought this component would solve his problem of linking together a code from the US where all the distances were calculated in inches and a code from the EU where all the distances were calculated in meters. He decided to whip up a test code to verify that the component did what he expected, and this is what he wrote in the `go` method of his driver component's `GoPort`.

```
# DO-NOT-DELETE splicer.begin(go)
genericPort = self.d_services.getPort("UnitLibrary")
library = units.UnitsLibrary.UnitsLibrary(genericPort)
meter = library.lookupUnit("meter")
inch = library.lookupUnit("inch")
converter = library.lookupConversion(meter,inch)
reverseConverter = library.invertConversion(converter)
inches = converter.convert(1)  # convert 1 meter to inches
meters = reverseConverter.convert(1) # convert 1 inch to meters
print "1 meter =  " + str(inches) + " inch"
print "1 inch  =  " + str(meters) + " meter"
self.d_services.releasePort("UnitLibrary")
return 0
# DO-NOT-DELETE splicer.end(go)
```

The software developer expected that `converter` would convert quantities from meters to inches, and `reverseConverter` would convert quantities from inches to meters. However, this is what the driver component produced when run:

```
cca>go drive GoPort
Loading main: ok
1 meter =  0.0254 inch
1 inch  =  0.0254 meter
##specific go command successful
```

8.2. Exercises debugging the units library

1. First, verify that the components are installed and giving the same, incorrect answers by running **`ccafe-single --ccafe-rc $STUDENT_SRC/components/examples/object_rc`**. This script will try using a Python and C++ implementations of `units.UnitsLibrary` called `library.PyUnitsLibraryComp` and `library.CxxUnitsLibraryComp`, respectively. Alternatively, you can do this graphically using **`gui-backend.sh --port`** ***`3314`*** **`--ccafe-rc $STUDENT_SRC/components/examples/object_gui_rc`** and **`simple-gui.sh --builderPort`** ***`3314`*** **`--host`** ***`localhost`***. Either approach should yield the following output.

   ```
   1 meter =  0.0254 inch
   1 inch  =  0.0254 meter
   ##specific go command successful
   ```

2. Modify the Python driver to output some information to help us figure out what is happening. Edit `$STUDENT_SRC/components/undrivers/python/undrivers/PyDriver_Impl.py`. Leading white space is

significant in Python, so ensure that the `p' in `print` starts in the same column as the statements around it.

```
# DO-NOT-DELETE splicer.begin(go)
genericPort = self.d_services.getPort("UnitLibrary")
library = units.UnitsLibrary.UnitsLibrary(genericPort)
meter = library.lookupUnit("meter")
inch = library.lookupUnit("inch")
converter = library.lookupConversion(meter,inch)
reverseConverter = library.invertConversion(converter)
print "converter converts", converter.convertFrom().name(), \
      "to", converter.convertTo().name()
print "reverseConverter converts", \
       reverseConverter.convertFrom().name(), "to", \
       reverseConverter.convertTo().name()
inches = converter.convert(1)  # convert 1 meter to inches
meters = reverseConverter.convert(1) # convert 1 inch to meters
print "1 meter =  " + str(inches) + " inch"
print "1 inch  =  " + str(meters) + " meter"
self.d_services.releasePort("UnitLibrary")
return 0
# DO-NOT-DELETE splicer.end(go)
```

3. Rebuild the Python driver with the following commands:

```
cd $STUDENT_SRC/components/undrivers/python
rm -f .lib .lib-nolibtool
make
```

4. Rerun the example using one of the approaches detailed above. Now the output should read.

```
converter converts inch to meter
reverseConverter converts inch to meter
1 meter =  0.0254 inch
1 inch  =  0.0254 meter
##specific go command successful
```

Eureka! It appears that `lookupConversion` modifies its arguments. Hence, `converter` and `reverseConverter` both end up being conversions from inches into meters; in fact, they are both references to the same `Conversion` object. Because the argument to `lookupConversion` is declared as *`inout`*, our imaginary, hapless, overworked scientific software developer assumed that `lookupConversion` would leave the incoming `Conversion` unchanged and return a distinct `Conversion` to perform the reverse conversion. The actual implementation modifies the incoming `Conversion` and returns it.

The implementation of `lookupConversion` is not illegal. However, it is confusing, and it is easy to make the mistake shown here. There are two approaches we can take to make this component better. Frist, we will modify the argument's mode to clarify the intent. Second, we will modify the implementation to leave the incoming value unchanged and return a new, distinct `Conversion` object.

5. If for some reason we want or need to use the initial implementation, we can clarify the component's interface by using *`in`* instead of *`inout`*. The implementation is free to modify the state of the incoming `Conversion` object, and the new method signature makes it clear that the incoming value is modified.

Note

Modifications to simple data types (i.e., int, long, double, ...) passed as *in* parameters do not propogate to the caller. However, objects and interfaces are always passed by reference. Hence, you can make calls on an *in* object that will change its state. Some day, Babel will support a *copy in* mode which will copy the incoming object/interface, provided it is serializable, and prevent the subroutine from being able to change the state of the incoming argument.

Edit `$STUDENT_SRC/ports/sidl/units.sidl` and change the mode of `invertConversion` to *in*.

```
/**
 * Transform a Conversion interface to convert values in the
 * opposite direction. Note this can modify or replace the
 * incoming parameter.
 */
void invertConversion(in Conversion convert);
```

Next, rebuild the port library with the following commands:

```
cd $STUDENT_SRC/ports
rm ../xml_repository/units*.xml ../xml_repository/library*.xml \
   .repository
make
```

6. Edit the Python driver you edited in Step 2. The new version should look like this.

```
    # DO-NOT-DELETE splicer.begin(go)
    genericPort = self.d_services.getPort("UnitLibrary")
    library = units.UnitsLibrary.UnitsLibrary(genericPort)
    meter = library.lookupUnit("meter")
    inch = library.lookupUnit("inch")
    converter = library.lookupConversion(meter,inch)
    reverseConverter = library.lookupConversion(meter,inch)
    library.invertConversion(reverseConverter)
#    reverseConverter = library.invertConversion(converter)
    print "converter converts", converter.convertFrom().name(), \
          "to", converter.convertTo().name()
    print "reverseConverter converts", \
           reverseConverter.convertFrom().name(), "to", \
           reverseConverter.convertTo().name()
    inches = converter.convert(1)  # convert 1 meter to inches
    meters = reverseConverter.convert(1) # convert 1 inch to meters
    print "1 meter =  " + str(inches) + " inch"
    print "1 inch  =  " + str(meters) + " meter"
    self.d_services.releasePort("UnitLibrary")
    return 0
    # DO-NOT-DELETE splicer.end(go)
```

7. Rebuild the Python driver and components with the following commands:

```
cd $STUDENT_SRC/components
```

```
rm -f .undrivers.PyDriver-python .library.CxxUnitsLibraryComp-cxx \
  .library.PyUnitsLibraryComp-python library/python/.lib \
  library/cxx/.lib undrivers/python/.lib .repository
make
```

Rerun the example. The components should now produce this output:

```
converter converts meter to inch
reverseConverter converts inch to meter
1 meter =  39.3700787402 inch
1 inch  =  0.0254 meter
##specific go command successful
```

However, this “solution” probably leaves you feeling a little hollow. You may be asking yourself, what's the point of `invertConversion`. I could have built `reverseConverter` by simply reversing the arguments to `lookupConversion`. If you're thinking this, you're absolutely right; this approach isn't very useful.

8. Undo the changes to `units.sidl` from Step 5 and the changes to `PyDriver_Impl.py` from Step 6. Rebuild the port definitions using the instructions above in Step 5 and the component implementations as instructed in Step 7.

9. In this step, we are going to change the C++ component that implements `units.UnitsLibrary`. You will need to edit `$STUDENT_SRC/components/library/cxx/library_CxxUnitsLibraryComp_Impl.cxx`. The current implementation of `invertConversion` is as follows:

```
void library::CxxUnitsLibraryComp_impl::invertConversion (
  /* inout */ ::units::Conversion& convert ) throw ()
{
  // DO-NOT-DELETE splicer.begin(library.CxxUnitsLibraryComp.invertConversion)
  ::library::CxxSimpleConversion sc = ::babel_cast < ::library::CxxSimpleConver
  sc.reverse();
  // DO-NOT-DELETE splicer.end(library.CxxUnitsLibraryComp.invertConversion)
}
```

First, it down casts `convert` to the actual implementation class, and then it calls `reverse` to reverse the direction of the conversion.

We would like to replace that implementation with one that creates a new, distinct `Conversion` object. Something like this will do the trick.

```
void library::CxxUnitsLibraryComp_impl::invertConversion (
  /* inout */ ::units::Conversion& convert ) throw ()
{
  // DO-NOT-DELETE splicer.begin(library.CxxUnitsLibraryComp.invertConversion)
  convert = self.lookupConversion(convert.convertTo(),
                                  convert.convertFrom());
  // DO-NOT-DELETE splicer.end(library.CxxUnitsLibraryComp.invertConversion)
}
```

Rebuild the example by executing **make** in the `$STUDENT_SRC/components` directory. After rebuilding the component, rerun it and verify that the C++ implementation now produces:

```
converter converts meter to inch
reverseConverter converts inch to meter
1 meter =  39.3700787402 inch
1 inch  =  0.0254 meter
##specific go command successful
```

10. Now perform a similar modification to the Python implementation of the component. You will need to edit `$STUDENT_SRC/components/library/python/library/PyUnitsLibraryComp_Impl.py`. The current implementation is similar to the initial C++ implementation.

    ```
    # DO-NOT-DELETE splicer.begin(invertConversion)
    sc = library.PySimpleConversion.PySimpleConversion(convert)
    sc.reverse()
    return sc
    # DO-NOT-DELETE splicer.end(invertConversion)
    ```

    Edit this to match the following. Please note that leading white space is significant in Python, so make sure your statements start at the same column.

    ```
    # DO-NOT-DELETE splicer.begin(invertConversion)
    return self.__IORself.lookupConversion(
      convert.convertTo(), convert.convertFrom())
    # DO-NOT-DELETE splicer.end(invertConversion)
    ```

    Rebuilding the python implementation requires a few commands.

    ```
    cd $STUDENT_SRC/components/library/python
    rm -f .lib .lib-nolibtool
    make
    ```

    Now rerun the python component too to verify that the changed fixed things.

Appendix A. Remote Access for the CCA Environment

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

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

A.1. Commandline Access

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

A.2. Graphical Access using X11

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

Warning

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

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

A.2.1. OpenSSH

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

A.2.2. PuTTY

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

A.3. Tunneling other Connections through SSH

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

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

A.3.1. Tunneling with OpenSSH

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

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

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

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

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

  ### Tip

  Don't worry if you don't understand the details of the java command that invokes the GUI. It is described in more detail in Section 2.3, “Using the GUI Front-End to Ccaffeine”. The key features for this discussion are the `--builderPort 3314 --host localhost` arguments, which tell the GUI to connect to the *local* end of the tunnel.

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

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

which can be used precisely as in the preceeding examples.

A.3.2. Tunneling with PuTTY

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

Tip

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

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

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

$Revision: 1.22 $
$Date: 2005/09/22 22:08:26 $

The primary tools you'll be using are the Ccaffeine CCA framework [http://www.cca-forum.org/ccafe/] and the Babel language interoperability tool [http://www.llnl.gov/CASC/components/babel.html]. This section provides brief instructions on how to download and install a distribution of these tools (named, creatively enough, “cca-tools”) that has been tested for compatibility with the tutorial code. In Chapter 6, *Using TAU to Monitor the Performance of Components* you will be using the TAU performance observation tools [http://www.cs.uoregon.edu/research/paracomp/tau/tautools/] in conjunction with the CCA, and if you plan to do that exercise, it will be necessary to install TAU on your system as well.

Caution

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

B.1. The CCA Tools

B.1.1. System Requirements

Note

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

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

- gcc >= 3.2
- Java Software Development Kit >= 1.4. The **java** and **javac** commands must be in your execution path.
- 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. (A network connection is required both to download the code cca-tools package and during the build process.)

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

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

  Note

  At present, there are no exercises that require MPI.

- Python >= 2.2 built with **--enable-shared** (on platforms that support shared libraries), and Numerical Python (NumPy). If you have multiple versions of Python installed and prefer to have a version in your execution path that does *not* meet the criteria above, you should set the `PYTHON` environment variable to point to a suitable version for the CCA tools prior to configuring them. You can check the python version with **python `-V`**.
- Fortran 90: A variety of Fortran 90 compilers are supported. Because Babel needs to know about the format of the array descriptors used internally by the compiler, the CCA tools will have to be configured with both the path to the compiler and information about which compiler it is. Here is the list of currently supported compilers and the associated labels recognized by the CCA tools configuration script.

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

  You should have the compiler in your execution path, and any relevant `.so` libraries in your `LD_LIBRARY_PATH`. These are required to properly configure the CCA tools package.

B.1.2. Downloading and Building the CCA Tools Package

1. The latest version of the CCA Tools package can be found at http://www.cca-forum.org/tutorials/#sources with a filename of the form `cca-tools-`*`version`*`.tar.gz`.

2. Untar the cca-tools tar ball some place that is convenient to build and follow the instructions in the README to build it.

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

B.2. The Ccaffeine GUI

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

B.2.1. System Requirements

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

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

B.2.2. Downloading and Setting Up the GUI

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

2. The scripts expect to be located in the same directory as the `jar` file. Instructions for using the scripts can be found in Section 2.3, “Using the GUI Front-End to Ccaffeine”.

B.3. Downloading and Installing TAU

Note

Note that TAU is only needed for Chapter 6, *Using TAU to Monitor the Performance of Components*. If you're not planning to do that exercise, or want to delay installing TAU until then, everything else should work fine without it.

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

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

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

   ### Note

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

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

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

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

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

B.4. Setting Up Your Login Environment

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

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

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

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

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

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

csh, tcsh and Related Shells.

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

bash, ksh, sh and Related Shells.

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

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

Tip

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

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

csh, tcsh and Related Shells.

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

bash, ksh, sh and Related Shells.

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

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

Appendix C. Building the Tutorial and Student Code Trees

$Revision: 1.10 $
$Date: 2005/11/11 22:24:59 $

The code for the tutorial itself comes in two forms, with pointers to both at http://www.cca-forum.org/tutorials/#sources. The file `tutorial-src-`*`version`*`.tar.gz` is the complete package, which has the full code for all of the components created in this Guide as well as a number of others. The file `student-src-`*`version`*`.tar.gz` is a stripped-down version of the tutorial code, from which we've removed all of the components created in working through this Guide.

Note

At the time this particular version of the Hands-On Guide was generated, the *`version`* was 0.4.1_rc1. If there's a more recent version available, you should probably use it, but you should also look for a more current version of this Guide to go with it. Both should have the same base version number (i.e. 0.4.1) with different release numbers. Take the highest available release number. Note too that because both the CCA tools and the tutorial code are evolving over time, you should make sure to use the version of the CCA tools distribution that is recommended for the particular tutorial version you're working with.

In order to give you a richer set of components to play with initially, we use the `tutorial-src` tree in Chapter 2, *Assembling and Running a CCA Application*, and the `student-src` tree for the remaining exercises. Throughout, the `tutorial-src` tree can be used as a reference, so see how things *should* look when you complete the exercises.

If you're participating in an organized tutorial, we will have built the `tutorial-src` tree for you in advance in a common location, whereas if you're working through these exercises on your own, you'll need to build it yourself. In both cases, you'll need to build your own copy of the `student-src` tree to work in. The procedure for both is nearly identical, and unless otherwise indicated, we will use `tutorial-src` to indicate *either* `tutorial-src` or `student-src`.

Tip

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

1. Download the file(s) you need from the location above. (If you're participating in an organized tutorial, the `student-src-`*`version`*`.tar.gz` tar file will already be on the system system, in the location indicated in your account information handout.)

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

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

and TAU, and Setting Up Your Environment).

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

   ```
   SUCCESS building arrayDrivers.CDriver
   ```

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

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

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

5. 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. 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.
   ====

   Test simple example involving a subset of the available languages: \
        c cxx f90 f77 python java

   #### Testing component connection and execution.

   ====
   ==== All simple run tests passed, go command executed successfully.
   ====
   ```

   Note that the *second test* (“Testing component connection and execution” is expected to *fail* at this stage when building the `student-src` tree because the component you're going to write in these exercises are missing. Both tests should succeed for the `tutorial-src` tree.

Appendix D. The Tutorial Build System

$Revision: 1.8 $
$Date: 2005/08/26 03:15:26 $

This appendix contains a description of the makefiles and other scripts used to build the tutorial. The main premise behind the build system is automation of as many steps in the port and component build processes as possible.

Note

While the tutorial build system is generally reusable for simple component projects, it is not intended for "black-box" use with arbitrary component applications. Please email tutorial-wg@cca-forum.org with any questions regarding the reuse of the tutorial build system in existing and new component applications.

The tutorial build system relies on a strictly-defined directory structures, in which the various files associated with the port and component definitions and implementations reside. SIDL definitions for all ports reside in `ports/sidl` directory; similarly the SIDL definitions of all components should be placed in the `components/sidl` directory.

The following files are used during the build and can be modified to customize the build process. The paths are relative to the top-level tutorial source code directory. Files which are expected to be modified by the user are shown with *italisized* names. Modifying the rest of the files could be more error-prone and is not recommended.

- *`components/MakeIncl.components`* is the makefile segment that contains a list of the fully-qualified SIDL class names for all components, with a suffix encoding the language of the implementation. If SIDL definitions for new components are added to or removed from the `components/sidl directory` or a different implementation language for a certain component is chosen, the *`components/MakeIncl.components`* file must be updated accordingly.
- *`MakeIncl.user.in`* is the file from which configure generates `MakeIncl.user`. This file contains some high-level user options, such as an optional installation prefix and the list of languages for which to generate client libraries. While both of these files can be modified by hand, it is better practice to use configure with the appropriate options to generate the desired settings. For example, it is better to run configure with the **`--with-languages`** option than to modify the value of the `USER_LANGUAGES` variable directly.
- *`ports/MakeIncl_template.user`* and *`components/MakeIncl_template.user`* are simple makefile segments that allow user-specified values to be prepended to the `INCLUDES` and `LIBS` variables, used for compilation and linking, respectively. Modifying the top-level template files must be done *before* the Babel-generated code has been built for the first time. In that case, the build system will put copies of these files in each port and component source subdirectories, and the local file will be used in the individual port/component library builds. If the settings of those variables are not uniform across all ports or components, individual *`MakeIncl.user`* files can be modified in their respective port/component source directories *after* the first build attempt (e.g., invoking **`make`** at the top level). It is usually a good idea to use revision control for any user-modified portions of the build system in addition to the files containing component implementations.

Files that should not normally be modified:

- `Makefile` contains targets for invoking the build in all the tutorial subdirectories.
- `configure.in` contains macros for discovering or setting the location of the installed CCA tools, namely, the `ccafe-config` script.

- The `utils/` directory contains a number of utility shell scripts used by the build systems. Modifying these is extremely dangerous and may easily break the build.

Appendix E. License (Creative Commons Attribution 2.5)

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. **Definitions**

   a. **"Collective Work"** means a work, such as a periodical issue, anthology or encyclopedia, in which the Work in its entirety in unmodified form, along with a number of other contributions, constituting separate and independent works in themselves, are assembled into a collective whole. A work that constitutes a Collective Work will not be considered a Derivative Work (as defined below) for the purposes of this License.

   b. **"Derivative Work"** means a work based upon the Work or upon the Work and other pre-existing works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which the Work may be recast, transformed, or adapted, except that a work that constitutes a Collective Work will not be considered a Derivative Work for the purpose of this License. For the avoidance of doubt, where the Work is a musical composition or sound recording, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered a Derivative Work for the purpose of this License.

   c. **"Licensor"** means the individual or entity that offers the Work under the terms of this License.

   d. **"Original Author"** means the individual or entity who created the Work.

   e. **"Work"** means the copyrightable work of authorship offered under the terms of this License.

   f. **"You"** means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

2. **Fair Use Rights.** Nothing in this license is intended to reduce, limit, or restrict any rights arising from fair use, first sale or other limitations on the exclusive rights of the copyright owner under copyright law or other applicable laws.

3. **License Grant.** Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

   a. to reproduce the Work, to incorporate the Work into one or more Collective Works, and to reproduce the Work as incorporated in the Collective Works;

   b. to create and reproduce Derivative Works;

   c. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission the Work including as incorporated in Collective Works;

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

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

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

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

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

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

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

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

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

5. **Representations, Warranties and Disclaimer**

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

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

7. **Termination**

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

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

8. **Miscellaneous**

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

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

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

Requested Attribution. CCA Forum Tutorial Working Group, *A Hands-On Guide to the Common Component Architecture, version 0.4.1_rc1*, 2006, http://www.cca-forum.org/tutorials/.

Or in BibTeX format:

```
@Manual{hog-cca:0.4.1_rc1,
  title =   {A Hands-On Guide to the Common Component Architecture},
  author =  {The Common Component Architecture Forum Tutorial
            Working Group},
  edition = {0.4.1_rc1},
  year =    2006,
  note =    {http://www.cca-forum.org/tutorials/}
}
```