Chapter 3

Component Architectures for Scientific Computing

3.1 Introduction

Scientific software libraries are often designed to solve a set of related problems. Scientific applications often require the use of more than a single library to solve an entire problem. However, combining libraries in a single application can lead to unwanted complexities.

The software libraries may not be directly compatible. For example, they may not be written in the same, or even compatible, languages, for example Fortran 90 and C++. One library may require a model of programming that is not appropriate for the others. For example, a library intended for use on parallel computers may place requirements on serial libraries that can not be easily met.

Object-oriented libraries can also add complication in that the key concept of object-orientation involves the definition of new data types. These data types have to be handled by any code interfacing with the library. Using two libraries in a single code can require that types be converted for passing between libraries, or that one library be modified to accommodate another’s defined types.

This was not an issue with earlier non-object-oriented languages popular for scientific coding such as Fortran77; there where no user-defined types to cause problems. Fortran 77 libraries written by different individuals could be connected without the issues raised by object-oriented methods.

However, even non-object oriented code libraries can cause inter-penetration of defined types and concepts. An example of this is parallel message passing libraries, such as MPI, which is used for distributed-memory parallel programming. A code using MPI must use the set of types MPI has defined, which are intended to ease the programming of parallel codes. However, these types are pervasive, in that they must be accounted for throughout the code, often in places where no parallelism is used. Also, adapting the code to work with another parallel library can be made difficult by the dependence of the code on the library’s defined types. Some projects have decided to simply define the library’s types as base types for the entire project, such
as the TAO project, which is built on PETSc, an MPI-based library [7] [5]. This binds TAO to the use of MPI.

Our instance of this problem of combining different code libraries is the need to combine software libraries to solve simulation based control and optimization problems. This often requires connecting one or more optimization libraries with one or more physical simulators. The simulation and optimization libraries are often written by different sets of experts. The difficulty is making the libraries interact.

In this experiment, we present a problem of optimal control of an acoustic wave field. The problem requires finding a time-series of control vectors that, by way of the structure of the problem, dampen a given wave field. Solving the problem requires simulating the wave field, and finding improved control series, which involves the coupling of a simulation package to an optimization package.

The optimization package used was the Hilbert Class Library (HCL), an object-oriented C++ package that abstracts the concepts common to optimization problems [2]. HCL provides a hierarchy of C++ classes in which optimization problems are more easily framed. This made writing the optimization component very simple.

Two acoustic-wave simulator where developed for the experiment. The first was a serial application; the second was a distributed-memory parallel application which used MPI. The parallel simulator followed the form described in the first chapter of this paper. Both simulators were written in C++, but could have been written in any other appropriate language, such as Fortran.

We solved the problem of connecting the optimization and simulation libraries by applying a software concept called components. We divide the problem into a HCL-based optimization component, and a simulation component.

Components are a means of structuring software by separating functionality into logical units. Each component is a self-contained entity that is capable of providing some set of services, which are defined in a stated interface. This allows the separation of the implementation of an algorithm, with its related details and complexities, from the application of the algorithm, or essentially a separation of implementation from interface.

Components can be used to separate a monolithic code so that the resulting portions are easier to develop and maintain. This separation also promotes code reuse, as each separate piece, since it has a more well-defined function, can be more easily applied to other problems.
In many ways, this is similar to the focus of object-oriented methods, which divide a code into smaller pieces by encapsulation of data and operations on the data, and dividing the encapsulated data into a hierarchy of related types. Components, however, stress the point of separation over abstraction, in fact, in his book, Szyperski claims that object-oriented and component techniques can be thought of as being orthogonal [24]. The result of using components is code with clearer distinctions between logical sub-units. However, this does not mean that object-oriented and component methods can not be used together, in fact their combination can be a valuable tool.

Components can also be seen as a means of bringing separate codes together to solve a common problem. Components provide a mechanism by which the libraries can be included in an application without undue complication. Components provide an interface, behind which all complications of the library, such as language, and other details of implementation, are hidden. Only types agreed upon, and stated in the interface, may be communicated to and by the component.

We liked the following conceptual model: components are similar to the workings of the United Nations. A large number of representatives, who may speak different languages, expect different protocols of communication, etc., are allowed to interact by the use of language interpreters and a defined structure for the proceedings. Note that ideally the translation is a matter of translating languages, and not meanings.

The component model has several advantages. The components are self-contained; there is nothing inside available to change. This means a user does not have to familiarize themselves with the internal workings of the component before they use it. The user simply has to know what the component does, and how it communicates with other components.

The hiding of the component’s interior allows the component to be written in any language capable of solving the problem, and implementing the communication methods necessary. This means that components written in different languages can easily be connected to perform a task. Keeping the internal workings hidden also allows parallelism to be hidden in a way that allows parallel codes to interact with non-parallel codes.

Validation and testing of code is simplified since each component can be tested in isolation. Each component can be treated as a black-box, that is we are not concerned with the internal workings, we are only concerned that the correct outputs
are returned for a given set of inputs. Since the internal workings can not be changed by the user, we do not have to worry that their modifications will invalidate the component. Also, since the internal workings of the component can not be casually modified, there is no fear of someone making poorly conceived modifications.

Components simplify the writing of applications by groups of people, since each component can be developed separately.

A project related to the use of components in a more sophisticated way is the NetSolve project [8]. NetSolve uses the component concept to allow users to construct codes that will eventually be executed on several machines. The NetSolve architecture handles the issues of finding available computing resources, and portioning the tasks to the available computers. We are not as concerned with the ability to execute code on multiple widely separated machines, but simply to divide applications in to portions that can be more easily written.

Another related project is the Common Component Architecture group, which is developing a standard for scientific computing with components [6].

We needed a framework in which the components could communicate. For this we chose the Common Object Request Broker Architecture (CORBA), a “vendor-independent architecture and infrastructure that computer applications use to work together over networks” [20]. CORBA is well supported, with implementations for most every computer architecture and operating system [14].

CORBA supports a client-server model. That is, one application, the server, provides services to another application, the client. For our experiment, we found it most natural to code the optimization component as the client, and the wave simulator as the server.

The optimization client and parallel wave simulators worked together to solve the boundary control problem. The key point, however, was that neither sub-unit knew of the internal details of the other. The HCL optimization client was not a parallel application, yet could be used to control the parallel wave simulator. Likewise, the simulation server knew nothing of HCL, yet could be controlled by the client.

3.2 Component Background

What is a software component? While several definitions of components exist [6] [24], we have found the following requirements to be useful
Components are building-blocks of code

Components are connection oriented

Components are binary composable

Components communicate in a known manner

Components are self-contained building-blocks of code from which software applications are constructed. A component is a deployable entity [24]; a unit of software that can be used without modification. The internal workings of the component are hidden from the user, which allows the developer to hide such issues as parallelism, and allows components to be written in different languages.

Components are often provided in a compiled form which deters modification. This promotes reuse since, if nothing can be modified, and the component fits the task, it is simply reused.

A component provides a known set of services to the application. The services are listed in a defined interface, which is specified in a language-neutral format. The defined interface is often in the form of a list of named operations that can be performed by the component [24]. Any data types, other than the most basic concrete types, that are communicated by the component are defined in the interface.

A component’s interface forms a signature for that component, with the intention that any component that implements the interface can be used in place of any other component implementing the interface. An example of this is the serial and parallel simulators developed for this project. They support the same simulation interface, and so can be used interchangeably. Changing simulation servers does not require changing, or modifying, the optimization client.

Applications are constructed by connecting components which provide the needed functionality. The components perform the desired task by communicating messages amongst themselves. These communications are in the form of concrete streams of data, which results in a message that can be easily transported between applications. The messages can also be easily transported over a network connection, which allows applications to be constructed that execute on more than one computer.

The fact that the messages must be representable as concrete streams of data, and that components may be written in more than one language, implies that the types communicated between the components must be completely specified in the
interface. This limits the use of objects as messages, unless the components support the same set of object definitions. In general, it is simpler to avoid communicating non-concrete types.

The actual communication is commonly handled by a “middle-ware” layer: the component framework. The framework provides communication protocols for the components, as well as a means of defining the interfaces. The framework hides the actual mechanism by which the messages are relayed, so the application looks the same whether the components are connected by inter-process communication, network sockets, or some other means.

Components are binary composable. That is, components are connected without the need for recompiling; no binary modification is made to the components when connected. This allows components to be connected to the application while the application is running, which allows an application to dynamically structure itself. The ability to link at run-time would allow, in more complicated applications, the ability to locate and use the best resources available at run-time [8]. In general, run-time composure results in a more flexible application.

Are there examples of components in widespread use today? Surprisingly, a Fortran 77 library is an example of a component. The library is “connected” by linking it into the application. The linking does not require that the library be re-compiled, and could be done at run-time. The interface is a list of procedures that can be used by the application. The interface is simple, since Fortran 77 does not support user-defined types; only basic types, such as integers and floating point numbers, are communicated through the interface. Fortran 77 libraries can be called from other programming languages, such as C/C++ and Java.

Modern operating systems are also an example of component systems [24]. Each application, as well as lower sub-system services such as device drivers, can be thought of as a component. The operating system provides the framework in which the components communicate, which is often a system with even wider applicability, such as Microsoft’s component object model (COM), or CORBA as used by the Gnome desktop project for Linux [12].