A Face-Lift for Aging FORTRAN Scientific Applications

Teri Roberts and Skip Egdorf

Abstract
Los Alamos National Laboratory has a rich legacy of physics-based modeling codes that date back to the Manhattan Project in the 1940's. As some of these codes have made the transition from the early weapons context in which they were first developed to a broader, industrial design and modeling context, they have retained constraints imposed by the limited structure of those early computing environments. These physics modeling codes represent decades of research and development. Advancement of the physics modeling in these codes has outpaced adoption of advances in computing technology. The availability of newer technologies such as parallel computing, distributed web applications, and advanced design paradigms compels evolution of these codes. As these codes evolve, their inherent value must be preserved and placed within the reach of a broader scientific audience.

Constraints of Early Computing Environments
Compared to today's standards, early computing systems were slow, expensive and required large amounts of physical space. Economical use of precious resources dictated the programming practices of the day. Optimizations at a micro level were extremely important and common.

Typical coding styles were very terse. For example, all variables local to a routine might be no more than two characters in length and all COMMON variables might be limited to three to six characters in length. Common program structures were limited to arrays. Massive use of EQUIVALENCE statements and offset indexing were often used to handle dynamic memory limits and variable array sizes. Comments were sparse.

The concept of dynamic linking did not exist until introduced by Multics[1] in the late 1960s and did not become common until its implementation in Windows and UNIX[2] appeared in the late 1980s. Consequently, the common construction mechanism used to build executable programs was static linking of all code references into one monolithic image. Overlay mechanisms allowed these monolithic programs to run on severely limited memory resources.

Integrating related programs was a challenge. Magnetic tapes or disk files were typically used as the basis for integration. Initial inputs were often limited to a card-image model of data. New capabilities were often added without full consideration of how the collection formed a coherent product.

[1] Organick, E. I., *The Multics System: An Examination of its Structure*, MIT Press, Cambridge MA, 1972. ISBN 0-262-15012-3. 1972. Multics as it was in the 60s. Reprint available from MIT Press.
[2] R. A. Gingell, M. Lee, X. T. Dang, and M. Weeks, "*Shared Libraries in SunOS*," in Proceedings of the USENIX Summer Conference, USENIX Association, Phoenix, 1987.

These constraints lead to a structure that is hard to approach except by individuals who are capable of grasping both the software structure inherent in the monolithic code and the physics concepts being modeled by the code. Such individuals are rare.

Encouraging Removal of Constraints

Today’s computers are faster, resources such as disk and memory are cheaper, and far less physical space is required than when these codes were first developed. It is possible to construct massively parallel machines by harnessing a set of relatively inexpensive workstations or personal computers. Human resources with appropriate scientific computing knowledge and skill are far more expensive than computing platforms. Optimizations at a macro level and improved interactions between program modules have become more important than maximizing resource use at a micro level. An architectural solution is needed that supports this different philosophy.

The size of the monolithic structure embodied in many scientific programs has reached a scale where very few scientists have an overall view of the parts as a whole. The development community for these codes is thus very small. To achieve an architecture that is approachable by a larger development community, we must create a cohesive set of smaller, modular components that can be composed to achieve the same effect as the huge monolithic program.

Terminology

In the subsequent discussion of both existing and suggested approaches, we will use three key terms.

1. *Objects*

Objects are the fundamental unit of software design and implementation. Objects are used when software engineers model individual entities within a single program.

A distinction is drawn between object-based design and implementation models and object-oriented design and implementation models. Object-based design and implementation techniques typically require only that the program design or implementation be structured out of entities that encapsulate processing and storage for each entity. Object-oriented design and implementation techniques typically add inheritance, polymorphism, and various other structuring facilities onto the object-based system.

Organizing scientific codes as a set of objects may be done either as a new development or as a re-design of an existing legacy code. For the purposes of component architectures, it is sufficient that the software design have characteristics of encapsulation and isolation of entities so that these objects may be grouped into modules. It is often the case that existing codes break these requirements, as they may have been designed years before current technology existed, and may have been in continual use and modification since.

The process of modular decomposition of these codes is primarily the application of these object-based encapsulation requirements to the existing codes to produce objects sufficient for grouping into modules.

Our requirements for an object in software design and implementation are simpler than that of a full object-oriented design; we require no inheritance, polymorphism, encapsulation, or any other capabilities currently in vogue for objects. From the point of view of component architectures, the type of analysis, design, and programming technologies used for individual programs is simply not a concern.

The main reason for discussing objects here is to note that the common terms dealing with object-oriented analysis, design, and programming and all the lifecycle tools and technologies built around these concepts do not affect our discussions of component architectures. This is not to discourage the use such technology as it will certainly aid the construction of correct codes at the single-program level and may simplify the migration of classic monolithic architectures to the modular architectures needed to build components.

2. Modules

Modules are the fundamental unit of software distribution. Modules are used when describing the structure of individual programs or program libraries. Modules are groupings of freely interacting objects that are self contained and bounded. Interactions with objects in other modules are allowed only through well-defined interfaces. To other objects, a module is a black box whose only visible effects are those provided by the well-defined interfaces of the module. Modules are self-contained when they communicate with other modules only through these interfaces. Modules are bounded when objects outside the module only interact with the objects inside the module via the defined interface.

A great deal of work is underway to modernize old scientific codes or produce new scientific codes at this level. Common techniques at this level include converting existing programs or writing new programs that use FORTRAN 90/95 modules or that use C++ class libraries.

3. Components

Components are the fundamental unit of software marketing and commerce. Components are used when describing different organizations cooperating to produce specialized software capabilities that inter-operate across different individual programs.

Components are groupings of modules with two additional constraints. First, the well-defined interface for the modules that compose the component is defined and enforced by computer tools. This means an Interface Definition Language (IDL), which is processed to automatically enforce modular containment and bounding, formally defines the interface. Second, a mechanism is added to allow control and discovery of the modules that make up the component and to control access to the modules within the component.

This may be as simplistic as the system's mechanism for finding and linking dynamic libraries, or as complex as a network interconnected request broker system.

Existing Approaches

Both the object and module approaches described above have been used with varying degrees of success to modernize the scientific codes that reside in the Laboratory.

Language Issues

One approach to the modernization problem is to rewrite the applications in newer languages such as C or C++ and newer FORTRAN versions such as FORTRAN 95. This language rewrite approach introduces integration problems when two or more modeling capabilities implemented in different languages must be coordinated to work together. We need solutions that go beyond a simple language upgrade. The various developers of the codes will never agree upon a single language.

We are aware of several examples of this language issue. They include two variants of one code that are both moving from FORTRAN 77 to FORTRAN 90, one code written entirely in C++, one code written in FORTRAN 90 with a C++ main routine that is moving toward more C++, one FORTRAN 90 code with a C++ simulation module, and one code written in a hand-crafted object-oriented FORTRAN.

Clearly, any attempt at integration of such disparate codes would have to go beyond a language-based solution.

Complexity Issues

A natural tendency, given expanded computer resources, is to add more complex data structures and functionality to the existing codes to support broader design and modeling contexts. Problems arise when the low level of abstraction (global variables and shared common data) present in many of the legacy codes introduces difficulties in identifying interfaces and integration points.

Examples of this complexity expansion include programs whose structure still reflects the set of overlays from memory starved machines and programs that are the merger of two or more early programs that communicated with tapes and disk files.

We need solutions that transcend the monolithic structure present in these codes. A limit has been reached in the size and complexity of these structures.

Suggested Approaches

In addition to the object and module level approaches in use, an architecture that supports the expansion of the community of users and developers is needed. This is and important characteristic of a component architecture.

Object-oriented Techniques Alone Are Not Enough

While object-oriented analysis, design, and programming techniques are widely accepted by the computing community as an efficient and robust method for software construction, the resulting individual programs are not sufficient by themselves to support the needs of a computing community. The ability of the community to extend old software and develop new software is constrained largely by the amount of reuse of existing objects that can be obtained and by how well existing programs can coexist and cooperate. By themselves, the techniques of object-oriented programming are of little help for this larger problem.

Modules Alone Are Not Sufficient

Modules represent a higher-level approach than objects. Typical module implementations include subroutine libraries and class libraries. These libraries are common on most current computing platforms. However, the discovery and dissemination of the module content is limited to user documentation and ad hoc browsing tools. Consequently, there is little or no enforcement of the use of the interfaces contained within the modules. This limits the ability to dynamically discover and reuse the module content in a way that enforces consistent and correct use. Modules of executable code are sufficient for delivery of an interacting set of objects, but by themselves, do not contribute to the dissemination of the interfaces provided by the modules.

Moving to a Component Architecture

Components add two things to module libraries. The definition and enforcement of module interfaces using IDL specifications and IDL compilers forms the basis for correct dissemination and use of module content. Using request broker facilities contributes to the discovery and delivery of the module contents.

Once software is structured as components, the modules that make up the components can be treated as commodity items that are purchased or traded and reused. This is because of the two conditions of well-defined interfaces and run-time control of module discovery that combine to allow implementation details to be hidden behind interface definition and request broker facilities.

Our motivation for moving to component architectures is to allow for the development and evolution of communities that can share and co-develop the software components in a way that allows an economy to emerge. These communities can be based around various economic models ranging from free trade to strict commercialization.

We are not suggesting that communities cannot form around module libraries. Examples of such communities can be found,[3] but with limitations.

In order to solve the language, complexity, and evolution of community of issues, a component framework must be adopted that allows multi-language integration and facilitates a modular decomposition and integration of large and complex codes. An industry standard framework that addresses these two issues exists in the Common Object

[3] Skip Egdorf, Teri Roberts, *Component Architectures and the Future Structure of Physics Codes*, LA-UR 01-4750, Los Alamos National Laboratory, 2001.

Request Broker Architecture (CORBA). Both an overview and detailed information can be found at the CORBA web site at http://www.corba.org.

The CORBA framework has a standard Interface Definition Language (IDL) for expressing public application interfaces that permit decomposition of large codes. This mechanism also mitigates the language integration problem.

CORBA typically operates in a networked environment. This does not accommodate the performance demands of physics modeling codes. Sufficient performance can be achieved by using run-time loading of dynamic modules as an implementation of CORBA objects while retaining the benefits of the CORBA framework. This framework offers the desired multi-language interoperability, facilitates evolution of the code, and retains the existing scientific value of the codes.

Additional Component Architecture Benefits

With a modular and dynamic framework in place, new user interface technology is possible. These include adoption of newer input data specification languages such as Extensible Markup Language (XML) and newer output representations like Java components that make these codes more approachable by a broader user audience in a web-based setting.

The program structures that process inputs are generally ad-hoc, dispersed throughout the code, and tightly coupled to the low-level data abstractions. It is difficult to understand and extend input processing in these codes. An architectural approach that localizes and contains input processing allows us to extend and expand the types of data processed.

XML is particularly appealing because of the self-documenting structure that can be achieved when using this kind of language to describe the various materials used in physics modeling. XML processing modules for the various types of data used in the modeling programs could then be used dynamically by the core modeling code when and where necessary.

Historically, inputs were often limited to 80 column card images, with ad-hoc comment conventions, special characters for field delimiters, continuations, and input terminators, and multiple special case overloading interpreted by matching ad-hoc code.

As an example, a material specification might appear as:

```
m1   13027.40c 1
m2   26000.40c 1
```

Without referring to a manual, how does one know or remember that the 13027.40c is the concatenation of atomic number, atomic mass, a separator, a library identifier, and a data class specification?

XML describes a class of data objects and partially describes the behavior of computer programs which process them.

A structured self-documenting material specification in XML might be:

```
<material>
  <num>1</num>
  <atomnum>130</atomnum><atommass>27</atommass>.<libraryid>40</libraryid>
    <dataclass>c</dataclass>
  <atomfraction>1</atomfraction>
</material>
```

Or if defined as an element with attributes:

```
<material num=”1” atomnum=”130” atommass=”27” libraryid=”40” dataclass=”c”
atomfraction=”1”>
```

While this looks formidable at first glance, keep in mind that these kinds of entries can now be the result of choosing items from graphical displays and do not have to be hand-written. Existing input files can be translated to XML syntax with by scripts written in pattern matching languages.

There are utilities to render graphical representations of the data used in the physics modeling codes. Often these are hand crafted two or three dimensional plotting packages that are tightly coupled to the low-level data abstractions present in the codes. A component approach would introduce a more independent and reusable visual presentation capability with its own attributes, physical layout, and containment.

With input and output handling mechanisms such as the ones just described, the physics modeling codes can now be deployed to a broad audience on a network and browser based environment as well as a more traditional graphical user interface.

Assumptions and Conditions

Our suggested CORBA-based component approach is based on several assumptions. We believe that optimizations at the Object Request Broker (ORB) level that recognize shared address space interactions make these types of solutions reasonable. When both the client and server appear in the same ORB address space, the need for marshalling-transmit, then transmit-demarshalling of the parameters exchanged between the (client) stub and (server) skeleton is eliminated. A simplified look-up mechanism similar to what occurs for dynamic linking can be used. This optimized approach can be found, for example, in the ORBit object request broker.

ORBit is a CORBA 2.2-compliant Object Request Broker (ORB) that is developed and released as open source software under the GNU General Public License and GNU Lesser General Public License (GPL/LGPL) and is supported by Red Hat and Ximian as part of the GNOME project. ORBit is engineered for the desktop workstation environment, with a focus on performance, low resource usage, and security.

A language binding defines how to use the IDL operations in a programming language. The current content of the CORBA web site indicates that there is no IDL / Language

Mapping Specification for FORTRAN. We assume that a CORBA language binding for FORTRAN can be developed based upon the Language Mapping Specification for C. This is the trickiest part of implementing our component-based solution. Until a formal language mapping is available for FORTRAN, we must manually write and handle the code that an IDL compiler would generate.

Preservation of Existing Value

While the modernization of these scientific codes is underway, we must demonstrate that we have not impacted or removed any existing physics modeling capability. This assurance comes from the use of an automated regression testing mechanism that is run on a nightly basis. Our current regression test mechanism encompasses a dozen different networked computer hosts (referred to as our “test farm”) and exercises 10 different combinations of hardware, operating systems, and compilers (our test farm). It also exercises the static and dynamic construction and execution techniques.

A specially constructed set of roughly 40 to 50 different test problems are run against the constructed executable code. These tests range from code coverage tests that exercise a large percentage of the code through physics model validation tests that exercise specific code features. A set of expected results is compared to the set of computed results. When the answers are a close enough match, we declare that the tests “track” the existing expected answers. Round-off errors on various platforms using different math library versions sometimes prevent exact matches.

The testing proceeds as follows:
Step 1. Each night at a designated time, an operating system command activates the regression test driver program. Using a special testing account, the regression test driver program first assures that the most recent version of the code from the Concurrent Versions System (CVS) repository is placed into a common shared file partition on one of the test farm machines. This common file partition is exported on the network so it can be accessed by all of the test farm host computers. This assures that the same source code base is used on each of the various hardware platforms.

Step 2. The regression test driver program then performs a secure shell login to each of the test farm machines, and runs a shell script to “configure” the source code for the particular ‘hardware/operating system/Fortran compiler/C compiler’ combination being tested. After a successful configure, the shell script “makes” the executable program and performs a “make tests” command to activate the tests. These special “make” commands are generated output of the configure step.

Step 3. All the generated output from the shell script execution is captured on each machine that is tested. The following morning, the output on each platform is gathered and examined.

Future development of this regression test capability includes an automated checking of all of the generated output on each platform and generation of a web page that developers can check.

Analysis

We are in the early stages of our modernization effort. We have demonstrated that CORBA-like in-memory optimizations work and are usable. Existing ORBs such as ORBit already implement and use this optimization. For our work at the Laboratory, as a first step we have produced an independent implementation of a framework that dynamically loads modules and uses an IDL syntax and compiler to generate non-CORBA stub and skeleton code for efficient in-process communication. This structure allows us to experiment with decomposition strategies for large scientific codes while retaining compatibility with CORBA.

The correct decomposition of portions of the code that represent reasonable approximations of architectural physics components must be determined, but we have the necessary structure to begin experimentation. Interfaces for appropriate components must be identified and developed. While the component structure is being discovered, the existing capability in the code must not be compromised and is being assured with our automated regression test mechanism.

Performance and/or Complexity Data

We are running the code both in its monolithic form and within the new dynamically linked framework. Our timing statistics for execution of the regression tests are shown in the accompanying Timing Table and range from little difference on the newer machines to a larger difference on older machines. However, execution speed alone is not the only metric to be considered. We believe the increased access to code capability and ease of extension will guarantee its continued use and further development. We believe "pluggable" physics modules can be attained using component architectures.

Result of the Work

The result of this work is not intended for commercial production. It is being undertaken to protect an investment.

Timing Table

Platform & compilers	Construction Method	Make Tests Seconds	% Difference vs Static
Linux i686 pgf77 gcc	static	1459.64	
Linux i686 pgf77 gcc	shared	1585.30	
Difference		125.66	9 % slower
Linux sparc g77 gcc	static	33092.21	
Linux sparc g77 gcc	shared	36455.07	
Difference		3362.86	10 % slower
SunOS sun4m f77 cc	static	66864.88	
SunOS sun4m f77 cc	shared	86557.35	
Difference		19692.40	29 % slower
OSF1 (Tru64) alpha f77 cc	static	1267.20	
OSF1 (Tru64) alpha f77 cc	shared	1295.60	
Difference		28.40	2 % slower
HP-UX 9000/735 fort77 cc	static	6964.80	
HP-UX 9000/735 fort77 cc	shared	7888.80	
Difference		924.00	13 % slower