

Component Architectures and the Future Structure of Physics Codes

LA-UR 01-4750

Skip Egdorf

Teri Roberts

June 27, 2001

Application areas outside scientific computing in general and physics codes in particular have successfully adopted a computing support structure known as component architectures. This structure has generally fulfilled the promises of reuse and sharing made since the introduction of object-oriented programming roughly twenty years ago. The success of this structure seems to make its adoption over other software problem domains inevitable. This paper describes a plan for allowing migration of existing classic physics codes into this new structure. These codes then gain the benefits provided by component architectures, while not losing the rich scientific attributes that have made these codes so valuable. The changes to the MCNPX code to support a component architecture are described and the ramifications of these changes for the community of code users is discussed.

1.0 Terminology and Basic Concepts

Several terms are used within the software community to describe aspects of the software engineering process. This section provides a brief introduction to some of these concepts while also describing how these terms and concepts will be used within this document.

1. Objects

Objects are the fundamental unit of software design and implementation.

Think at the level of objects when a single software engineer or a team of software engineers are modelling individual entities within a single program.

Object-oriented specification, design, and implementation techniques are widely accepted in the software engineering field as the most proper method of developing software currently available.

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. It is sufficient for purposes of a component architecture 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 thus simpler than those of a full object-oriented design; we require no inheritance, polymorphism, encapsulation, or any other trappings 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.

Think at the level of modules 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 are 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.

Think at the level of components 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 generally means that the interface is formally defined by an Interface Definition Language (IDL) that is processed to automatically enforce modular containment and bounding.

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.

Several component architectures have been proposed and implemented. Two seem to be dominating at this point. The first is Microsoft's Component Object Model (COM) and Distributed Component Object Model (DCOM). The second is the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA).

2.0 Component Architectures

2.1 General Description

Object-oriented programming techniques have been generally accepted by the computing industry as the most robust, correct, and efficient technology for design and implementation of individual programs. Structuring the analysis and design processes and the implementation itself around modules of objects is the current accepted practice in the software engineering community. However, individual programs are not all there is to a computing community. The ability to develop new software is constrained largely by the amount of re-use of existing objects that can be obtained and by how well existing programs can co-exist and cooperate. By themselves, the techniques of object-oriented programming are of little help for this larger problem.

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 the two conditions of well-defined interfaces and run-time control of module discovery combine to allow implementation details to be hidden behind interface definition and request broker facilities.

The primary reason for moving to a component architecture 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.

Some such cooperative communities have formed in the past around subroutine libraries such as LINPAK or DISPLA. Two more recent examples of such communities built around function libraries are the wealth of programs that have been built on top of the X Window system on Unix and the WIN32 libraries used on Microsoft operating systems. It is instructive to note how few such communities exist and the limitations on the size of these communities.

For a community built around function libraries to emerge at all seems to require that the library have a long life span with a stable interface so that communities of expert developers can form a stable and long-lived community around the library. The maximum size of community also seems to be limited by the expertise required to use the library and the breadth of use that can be made of the library by those experts. The main limitations to

the development of communities and economy due to the expertise required of the community using these libraries is twofold.

First, the library approach does not enforce complex interfaces, nor does it offer tools to simplify its use for non-domain experts. It is too easy to break the library interfaces, and to correctly use the complex library interfaces requires a high degree of expertise in the library details. It is difficult for experts in fields outside software engineering to gain the expertise needed in software engineering to build domain specific libraries on such a low-level base. Thus, one finds more complex windowing interfaces (MOTIF) built on top of lower-level windowing interfaces (Xt) but there are few examples of domain specific graphics toolkits built on top of the lower level interfaces. It seems that the cost of entry to using the library excludes all but a few domain experts. This limits the size of community that can form.

Second, the libraries are very limited on how they may depend on other libraries. Complex systems are built of interacting components. Libraries, by the nature of their simple function-call interfaces, are limited to end points in a graph of interactions. The most that can be done is to make simple calls to other libraries. The limit to these facilities seems to be the development of other libraries on top of the existing structures. The two examples given above regarding the expertise needed to use the libraries also demonstrates this second point. For example, the Motif library on top of the X Windows Xt toolkit has no facility for plugging in lower level or same level capabilities such as rendering a scene using ray tracing rather than simple raster graphics. In the same way, the Microsoft Foundation Class libraries on top of WIN32 have little capability to build new structures below the library that extend its capabilities other than simple child-class definition at compile time.

These two limitations lead to a model of program construction that is dominated by monolithic programs with libraries statically linked into the single address space of the program. The only communities that can evolve are those that are experts in the primary problem domain of the library. In the realm of physics codes, there are just too few particle physicists who are also true experts in software engineering or too few software engineers who are also particle physicists. For a community to evolve, a sufficiently robust set of capabilities must be developed to allow a domain expert in some field other than software engineering to safely and correctly use software components appropriate to that domain of expertise.

Despite recent moves to the dynamic shared library structure pioneered by Multics in the 1970s, the new capability allowed by dynamic libraries is little used by current scientific codes. For example, most systems now use a dynamic library for basic system services. The standard C library is a good example. In theory, on any system commonly available to scientific programmers today, this library can be replaced from run to run of a program to get some new capability. Why is this never done? Because the structure of the libraries makes it too hard to discover the interfaces provided by the library, and the simple structure of the library provides little incentive to use anything other than the single system-supplied correct version of a library routine. Again, the level required for reuse of components in a specific problem domain has not been met.

2.2 Component Architecture Characteristics

There are many examples of places where scientific codes could benefit from more modularity, sharing, and reuse. For example, there exist several solid modeling packages that are designed to handle geometries of entities in the real world. Examples are Autocad and Pro-Engineer. Why shouldn't physics models have an easy capability to plug in different solid modeling inputs in the same way we have standardized on a common light-bulb base for plugging in our sources of reading light, or a standard socket for plugging a quad TTL NAND-gate into a circuit board? Why shouldn't a scientific code be able to choose one of several variants of a physics algorithm for some specific interaction at run time based on the problem input rather than having to code one and only one variant into the program? Why shouldn't a code group that prefers Fortran 95 not be able to easily use portions of work of a group that prefers C++ even where serious internal details of implementation and design exist? Why shouldn't an organization that specializes in one aspect of a physics problem be able to provide pluggable modules with its specialty to other organizations with other fields of expertise and get modules in return?

The component architecture attempts to address these issues.

The introduction of an interface definition language that specifies the interactions of a set of modules that make up a component allows libraries to be developed where more complex interfaces are possible because automated tools handle the complexity of dealing with the interfaces.

The introduction of a request broker mechanism allows dynamic linking of libraries into a single image at runtime where efficiency is required (as it often is in complex scientific codes) and yet also allows network client-server structures where these are appropriate. The use of the IDL interface descriptions means that the domain-specific programmer is freed from direct concerns about these interactions.

2.3 Specific Examples

These two characteristics seem to have the needed set of capabilities that have allowed other problem domains to develop robust economies and communities of traded components. Many examples can be found at the OMG's web site (<http://www.corba.org>) in the area described as *Domain CORBAFacilities*. Examples in many business areas can be found by a Web search for Microsoft Visual Basic components in various domains. It is only a matter of time before large scientific codes move in this direction.

Two main variations on component architectures are currently common and available in the software community. These are the Common Object Request Broker Architecture (CORBA) and Microsoft Distributed Component Object Model (DCOM). As DCOM is a proprietary system and scientific codes reside to a large extent on Unix systems, it seems that CORBA is the system-of-choice for this work.

CORBA is a standard way to achieve cooperation between different hardware, software, networks, and programming languages. Specifications are coalesced and issued by the

Object Management Group (OMG). The OMG, founded in April of 1989, is a consortium government, business, and university members committed to the adoption of a standard architecture for distributed object computing. Frequently the computing environment available to programmers is heterogeneous. It contains different hardware, software, networks, and programming languages. This presents barriers to inter-operability of the diverse parts that contribute to whole programming solutions. CORBA represents one inter-operability solution.

CORBA uses Interface Definition Language (IDL), language mappings for most modern programming languages, Object Request Brokers (ORBs), Object References, Internet Inter-ORB Protocol (IIOP), and asynchronous invocation modes. A simplified explanation follows that ties all of these concepts together.

A CORBA component consists of objects that have a unique identity or handle (Object Reference) that is like an address with built-in forwarding as the object travels. Objects exhibit standardized interfaces expressed in terms a contract; the contract advertises the object's services, showing how to invoke it, and informs the communication infrastructure (ORBS and IIOP) about all messages the object can send and receive, and what message formats can be used. The contracts are written in a standardized language (IDL) that allows programs written in different programming languages to interact through requests for services. The ORB coordinates the interaction of connecting the different languages through their interface definitions. The ORB may need to forward a request to another ORB that is remotely located but accessible via an internet or may optimize a call to a server collocated (and possibly dynamically loaded) in the client's address space. The program that made the request may continue computing and then later check back with the ORB for the result of the request (asynchronous invocation mode).

The key parts of this for scientific codes are that module interfaces are fully defined and distributed as IDL and IDL compilers hide all the details of implementation from the scientific user, and the ORB allows ranges of inter-operability from cross-network client-server structures to direct calls when the client and server implementations are in the same image, again with the implementation details hidden from the domain-specific user.

In late October of 2000 OMG announced that its CORBA® Inter-operability platform was adopted by the International Organization for Standards (ISO) and International Electrotechnical Commission (IEC) as an international standard, ISO/IEC 19500-2.

3.0 MCNPX version 3

MCNP is a general purpose Monte Carlo N-Particle code that can be used to simulate neutron, photon, electron, or coupled neutron/photon/electron transport through different materials. The Monte Carlo method originated within the Mahattan Project in World War II and the MCNP code has been extensivel developed ever since that time. The MCNPX project was started in 1995 for the Accelerator Production of Tritium program, and extends the capabilities of MCNP to all particles and all energies. MCNPX includes the use of physics models to compute interaction probabilities where tabular cross-section

data are not available. Physics models are currently included from the LAHET and CEM high-energy codes.

MCNPX extends the traditional user base of MCNP in significant ways, and major applications include:

- Design of Neutron Spallation Sources
- Creation and destruction of radionuclides in accelerator environments
- Accelerator shielding design, health physics and dosimetry calculations
- Medical applications such as proton therapy and the design of source for Born Neutron Capture Therapy
- High altitude aircraft dosimetry
- Shielding of spacecraft and evaluation of single event effects
- Material damage assessment in accelerator environments
- Radioactive ion beam target design

The code and associated nuclear data libraries are developed and maintained at Los Alamos by project participants from several groups that include CCS-4, D-10, IM-8, LAN-SCE-12, X-5, and T-16. Export control approved distributions of the code and manuals for MCNP and MCNPX can be obtained from the Radiation Safety Information Computational Center (RISCC) in Oak Ridge, TN.

MCNPX is a single monolithic program consisting of about 100,000 lines of Fortran 77 with a bit of C for a few system interface functions. It is structured primarily around a large global common block for data. This common block is used to simulate an assembly language level base-offset addressing scheme by-passing Fortran's array abstraction. It is, frankly, about as far from a modular modern program as is possible.

Currently a good deal of effort is underway to modernize the basic structures of the code. New physics capabilities are being added regularly. The set of developers with sufficient expertise to deal with both the large monolithic code and the physics involved is quite small. Software re-structuring is also underway. The code is also being modernized internally to use Fortran 95 modules.

These efforts are worthwhile and important, but from the point of view of developing an international community of physicists who can share, extend, and re-use physics capabilities from MCNPX, other additions must be made. In particular, a component architecture must be developed on top of the current extension and modulerization efforts already underway to allow the physics community to make use of different capabilities currently existing only inside the MCNPX code. Once such a component architecture exists, MCNPX becomes a model of how other physics codes can be restructured so that a robust community and economy in physics capabilities can be realized.

3.1 A Component Infrastructure

The route to a component architecture for this code has several intermediate steps.

3.1.1 Basic Infrastructure

The first step is to provide an architecture that will allow components to be stripped out from the current MCNPX monolithic structure. To this end, modifications have been made to the MCNPX build procedure to allow the main MCNPX code to be packaged as a shared library with a single defined entry point which is the equivalent to the original main program entry for the program. This component of the new component architecture is known as the *classic MCNPX* module.

It is important to emphasize that in this step it is immaterial how the classic MCNPX module is designed and implemented internally. The current version of the classic MCNPX module code uses the Fortran-77 single shared common block version of the code that is very similar to the monolithic version of the same program. It can easily and invisibly also use the planned Fortran-95 module-based version of the code or a re-written hybrid Fortran-95 C++ version. From the point of view of the component architecture, those details don't matter. This is not to say that the modernization efforts at those levels are not important to this effort. Removing the global dependencies in the code will be vital to allowing components to be stripped out of the existing structure into their own components in the future.

A driver program has been developed which becomes the new very small main program for the MCNPX code. The purpose of this new main program is to provide a module loading capability and to provide a few communication functions that allow inter-module communication between modules in separate components. This driver program is controlled by a script that currently loads three modules.

1. The first module is a bit of code that enables the FORTRAN run-time system and allows other FORTRAN programs to be loaded and called without a classic MAIN program. As a note, the various FORTRAN vendors make this an operation that ranges from merely painful to hideously difficult.
2. The second module is the classic MCNPX module. This is all the code that makes up the current monolithic program minus the main program entry point. This code is constructed in a slightly different way from the monolithic program so that it appears as a shared library.
3. The third module is a simple bit of code that uses the thread of control of the module-loading protocol and invokes the one defined interface of the classic MCNPX module. The code then runs as if it were the monolithic version.

A user exercising the component version of MCNPX at this level sees no difference between it and the classic monolithic version.

A programmer extending the component version of MCNPX does see differences. The build system requires a different compiling and linking than the static version in order to

produce proper shared libraries. However, the main characteristic of this version from the point of view of a programmer using the system is that the interface definitions used for cross-module communication must be manually written. There is no IDL compiler. This makes use at this stage still very experimental and dangerous.

Even at this early stage, several interesting things can be done with the code.

It is quite feasible to split off parts of the code as modules where minimal interactions with the global common and other portions of the code exist. There are several pieces of physics in the code as it currently exists that are fairly pure functions or that have little state in the shared common storage. These can be split out into a separate shared library with accessor functions written to exercise the new modular capability and interface code written to communicate with the remainder of the (now slightly smaller) classic MCNPX module.

As such components are split off, they can be immediately used by programs other than MCNPX. For example, unit testing of modules becomes much easier.

This modular decomposition becomes important when one observes that some of these physics capabilities are under some dispute as to just which version best models desired physics. For experimentation with different physics in the old code, the source code would need to be modified and the whole system rebuilt. With this early modular system, different organizations can produce different versions of a specific physics interaction and use a single binary distribution of the rest of MCNPX to exercise them. This allows not only a potential commerce in such modules, but the promise of truly reproducible experiments with these models.

Another direction that can now be pursued is extension of the few existing modules. For example, instead of the third module loaded by the classic MCNPX component system described above using the module initialization protocol to simply call the classic MCNPX module's one defined interface ("run mcnp"), a new module can be written that uses a new thread of control and an event loop to add a simple windowed control panel to MCNPX. Unlike previous versions of the code that require modification of source code for this, such a new windowing interface could be developed by a third party with only the binary libraries for the other parts of the code. Several such interfaces become possible. One such might use a web CGI interface to run MCNPX from a web browser, while others may use different graphics toolkits (Motif vs. GTK for example).

Even in this very early state, the potential for a large community of users developing a community and a commerce in components is visible. For the futurist, the potential for heated competition for acceptance of module interfaces as *standard* and for political posturing by different organizations for control of those standards is also visible, though not yet quite feasible at this stage of development.

3.1.2 Interface Definition Tools

With the basic infrastructure for components in place, some initial modular decomposition of the code can begin. Two limitations to this effort dominate. The monolithic structure of the existing code will make some decomposition difficult. This is being addressed by other efforts. Once decomposition is feasible, the need to manually code the module communication code makes the decomposition difficult. The manual construction of these interfaces also means that other communication mechanisms cannot be exploited (e.g. a network client-server call rather than direct dynamic linking into a single address space).

The next step in building a component architecture capable of supporting a community is to build an interface definition language compiler that automates the process of constructing the inter-module communication procedures.

A subset CORBA IDL is an obvious choice for the language as it provides the facilities needed for the interface construction, has good compiler development tools already available, and sets the stage for later movement toward a full CORBA-compliant architecture.

When this point is reached, release of versions of MCNPX to a larger audience as a part of a component architecture is feasible. This is also the point at which organizations that control other codes that have followed this route toward components can actually begin to use political means to control future standardization of interfaces for components. This should be viewed as a good sign that a community and an economy of physics modelling components is about to emerge.

3.1.3 Full CORBA

Once MCNPX components are being developed, extension of the component architecture to a full CORBA structure is possible. The need to move to a full CORBA structure will be based on the need to use a more standard transport for inter-module communication than the ad-hoc mechanisms developed in the previously described version. At such time as module libraries move from simple in-memory linking and move to client-server structures, full CORBA compliance at the IIOP level will be appropriate. The fast inner computational loops will probably always need the efficiency of in-memory linking of components. However, many aspects of the code that are done once at start-up or shut-down time such as reading geometries, material properties, and other problem descriptions or producing reports and graphics after a run will be natural to move to central servers connected by CORBA IIOP.

Conversion to this structure will involve some additional complex tasks, such as development of CORBA-standard Fortran bindings. Once this is done, a range of services should evolve using client-server models. The work should involve implementation details only. Components developed under the previous architecture and described with IDL files should simply work with the new facilities. Users and component programmers should see little change other than a wider range of component options available to them.

While difficult technical issues remain at this level, the focus of a lot of activity will then move primarily to a political level. There is no requirement that a component derived from MCNPX be used just in MCNPX. It would be expected that other large physics codes would also move in this direction. There would be no technical impediment to using physics in a component derived from some other code could not be used in MCNPX. It is expected that there will exist serious and probably heated discussions and political maneuvering regarding just what components should be defined and just what interfaces should be standardized for those components.

A community should form.

3.1.4 CORBA Standardization

The expected turmoil of a political process whereby standard interfaces for component libraries usable by a wide variety of physics codes is agreed upon will take some time to run its course. Once a community of physicists settles on standard set of components that inter-operate across different codes, real formal standardization of those components by a recognized standards body can take place.

Currently the mechanism for this standardization is by definition of *CORBAFacility* standards through the OMG. This phase must naturally be some way in the future, as such standardization usually happens only when there is consensus in the effected community.