Preprint
UCRL-JC-140549

Component Technology for High-Performance Scientific Simulation Software

T. Epperly, S. Kohn and G. Kumfert

This article was submitted to
International Federation for Information Processing, Ottawa, Ontario, Canada, October 2-4, 2000

November 9, 2000

U.S. Department of Energy
Lawrence Livermore National Laboratory

DISCUSSION

Speaker: Scott Kohn

Thierry Priol : I do not understand why the data distribution specification is not exposed in the IDL associated with a parallel component.

Scott Kohn : One of the goals of our work is to support the redistribution of very complicated scientific data objects, such as unstructured meshes, hierarchical adaptive mesh refinement structures, various matrix representations, and so on. We are not planning to build data distribution specifications into the IDL because, at least at this time, we do not understand how to represent these diverse data decompositions in a static IDL description. To our knowledge, the only work in this area has focused on array structures. Another issue is that the IDL description is static, whereas data decompositions often change during the course of a simulation. We plan to concentrate on run-time descriptions of data objects. For example, a distributed parallel object may be required to support one of a set of data distribution interfaces through which the object describes its data distribution state. We feel this approach is more appropriate for sophisticated data decompositions that change during the course of a computation.

Michael Thuné : With regard to your conclusion, one could ask: Can we do without component technology? What would be the alternative?

Scott Kohn : I think some form of component technology will be necessary, whether it is scripting or some other style of integration approach. I am simply not sure that our particular design choices are the correct ones. For example, how important is language interoperability? Is it sufficient to support one scripting language and one compiled language? If language interoperability is important, should we use an IDL approach? Should the IDL express parallelization and redistribution for complex data objects? I believe that there is still a lot of exploration and research to be done by this community.

Richard Fateman : Regarding alternatives to component technology: Monolithic systems such as Lisp machines (built at various times by Xerox, Texas Instruments, Symbolics, and LMI) provided access to all aspects of a computing environment: operating system, networking, compilers, memory management, object representation, visualization. There are major advantages to such an approach as shown by the impressive productivity of these systems when used by skilled programmers. Inadequate languages force system builders to deal with inter-language communication and many associated complexities—typically poorly as when error indicators are unchecked at interfaces, memory management is inconsistent, and data must be repeatedly rearranged and reformatted.

Scott Kohn : I agree that choice of language and the programming environment can significantly impact productivity. I question whether the scientific

software community would adopt a single environment or a single language. In some sense, limiting ourselves to only one language would be a bad choice in that it would limit exploration. We use many different languages because each language offers different advantages. Fortran, in spite of all of its limitations, is a very good language for array manipulation. C++ offers object-oriented capabilities at a reasonable cost in terms of performance. Java is a better object-oriented language, but performance is not as good as C++. Python provides scripting capabilities. I don't see any single language as an overall solution. Component technology is a very pragmatic solution to the integration of diverse languages and environments.

John R. Rice : Suppose everyone agreed to use a single language forever more. How would this eliminate the need for a component technology? I think it would still be essential.

Scott Kohn : I agree, although I think the need for component technology would be diminished. For example, performance considerations aside, Java and Python are very good programming languages that share many characteristics of a good component system: physical deployment and packaging standards dynamic loading, good support for abstraction, interface metadata, and common object behaviors. In the scientific domain, I think components also offer advantages for distributed computing and parallel data communication between components. To be pragmatic, however, technology is always changing, and the community would not want to choose a single language forever more. We need an integration approach such as components that will adapt to the changing technology landscape.

COMPONENT TECHNOLOGY FOR HIGH-PERFORMANCE SCIENTIFIC SIMULATION SOFTWARE*

Tom Epperly, Scott Kohn, and Gary Kumfert
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA, USA
tepperly@llnl.gov
skohn@llnl.gov
kumfert@llnl.gov

Abstract We are developing scientific software component technology to manage the complexity of modern, parallel simulation software and increase the interoperability and re-use of scientific software packages. In this paper, we describe a language interoperability tool named **Babel** that enables the creation and distribution of language-independent software libraries using interface definition language (IDL) techniques. We have created a scientific IDL that focuses on the unique interface description needs of scientific codes, such as complex numbers, dense multidimensional arrays, complicated data types, and parallelism. Preliminary results indicate that in addition to language interoperability, this approach provides useful tools for thinking about the design of modern object-oriented scientific software libraries. Finally, we also describe a web-based component repository called **Alexandria** that facilitates the distribution, documentation, and re-use of scientific components and libraries.

Keywords: component technology, language interoperability, software repository, parallel high-performance scientific software

*Work performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under Contract W-7405-Eng-48. Work funded by LLNL LDRD grant 00-SI-002 and the ACTS program of the DOE Office of Science.

1. MOTIVATION

Numerical simulations play a vital role as a basic research tool for understating fundamental physical processes. As simulations become increasingly sophisticated and complex, no single person—or even single institution—can develop scientific software in isolation. Development teams rarely possess sufficient resources and scientific expertise in all required domains to successfully create a complex application from scratch. Instead, physicists, chemists, mathematicians, and computer scientists concentrate on developing software in their domain of expertise. Computational scientists create simulations by combining these individual software pieces.

In collaboration with the Common Component Architecture forum [1], we are developing software component technology for high-performance parallel scientific computing. The goal of this effort is to improve the software development processes of scientific codes by using proven techniques and technology from industry. Component technology addresses technological barriers to software re-use and integration, such as incompatibilities in programming languages, interface descriptions, or physical deployment. By removing such barriers, component approaches will allow computational scientists to concentrate on building more sophisticated numerical simulations and make scientific progress instead of wasting effort integrating incompatible software.

In this paper, we describe our recent research and development efforts in two areas of component technology: language interoperability and a component repository. As part of our language interoperability efforts, we have developed a tool called **Babel** to enable the creation and distribution of language independent software libraries. To use **Babel**, library developers describe their software interfaces in a special Scientific Interface Definition Language (SIDL). **Babel** uses this SIDL interface description to automatically generate "glue code" that enables the software library to be called from any supported language. We have also designed and implemented a prototype web-based repository called **Alexandria** to encourage the distribution and reuse of scientific computing software components and libraries. **Alexandria** provides a convenient web-based delivery system and thus lowers the barrier to adopting component technology.

This paper is organized as follows. Section 2 surveys component technology approaches for scientific computing and discusses related work. Section 3 discusses our language interoperability approach, modifications necessary for the scientific domain, and the **Babel** tool. Section 4 introduces the **Alexandria** web-based component repository and its implementation architecture. Section 5 concludes with a discussion of how these component tools have been used in the context of a high-performance scientific software library.

2. SCIENTIFIC COMPONENT TECHNOLOGY

Component technology is an extension of object-oriented software technology that focuses on the issues of software interoperability and re-use. There is no universally accepted definition of the term *component* by the software community [15]. In this paper, we loosely define a component as a software entity that adheres to certain standard interoperability behaviors that facilitate re-use (see below). These standard interoperability behaviors are defined by a component architecture. To use a hardware analogy, a component architecture is like a hardware back-plane that defines standard signal pin-outs and standard bus negotiation protocols, and components are interoperable cards that plug into that back-plane.

Component technology is different from object-oriented approaches, software modules, scripting [2, 3], or software frameworks [4, 5, 6, 9]. These techniques do not typically address interoperability concerns. However, component approaches do make use of these related technologies. For example, a software framework may be created within a component architecture to address a particular application domain. Scripting languages may be used as an integration language to connect existing software components.

Industry has created component technology to address issues of interoperability due to different programming languages, the complexity of applications developed using third-party software, and the incremental evolution of large legacy software. There are three common component technology standards in the business community: COM [8], JavaBeans [14], and CORBA [11]. COM (Component Object Model) is Microsoft's component standard that forms the basis for interoperability among all Windows-based applications. Microsoft recently introduced a new component initiative called .NET [10] that combines ideas from COM and `Java` and will likely be the future of Microsoft technology. Sun Microsystems has developed JavaBeans and Enterprise JavaBeans [13] based on their `Java` programming language. CORBA (Common Object Request Broker Architecture) is a cross-platform distributed object specification that supports the interaction of complex objects written in different languages distributed across a network of computers running different operating systems.

Component technologies such as CORBA, COM, and JavaBeans have been very successful in industry; unfortunately, they are designed for the business environment and do not address many of the issues associated with large-scale parallel scientific computing. For example, industry approaches do not address data distribution support for massively parallel SPMD components.

We believe that a successful component technology for scientific simulation must address four issues: language interoperability, common component behavior, physical deployment standards, and support for distributed parallel

communication. In the following, we review related work in the scientific community building towards parallel component technology.

3. LANGUAGE INTEROPERABILITY TECHNOLOGY

Computational scientists developing large simulation codes often face difficulties due to language incompatibilities among various software libraries. Scientific software libraries are written in a variety of programming languages, including Fortran 77 and 90, C, C++, or a scripting language such as Python. Language differences often force software developers to generate mediating "glue" code by hand. In the worst case, computational scientists may need to re-write a particular library from scratch or not use it at all. For maximum portability across different languages, library developers sometimes implement their software in C; however, this approach either ignores advanced software techniques such as object-oriented development or forces library developers to generate and maintain low-level object-oriented support code by hand.

We have developed a tool called Babel that addresses language interoperability and re-use for high-performance parallel scientific software. Its purpose is to enable the creation and distribution of language independent software libraries. In the following sections, we describe our interoperability approach, modifications necessary for the scientific domain, and the **Babel** tool architecture.

3.1. SCIENTIFIC IDL

Babel addresses the language interoperability problem using Interface Definition Language (IDL) techniques. An IDL is a special language used to describe the calling interface (but not the implementation) of a particular software library. IDL tools use this interface description to generate "glue code" that allows the software library to be called from any supported language. IDL approaches are common in industry component architectures such as CORBA or COM. However, these IDLs are primarily targeted for business applications. We have designed a Scientific Interface Definition Language (SIDL) that addresses the particular needs of parallel scientific computing. SIDL supports complex numbers and dynamic multi-dimensional arrays as well as parallelization attributes and communication directives that are required for general parallel distributed data structures. SIDL also provides other features that are generally useful but not necessarily related to scientific computing, such as an object-oriented inheritance model similar to Java, name space management, and interface versioning.

3.2. BABEL TOOL ARCHITECTURE

The Babel tool suite consists of a number of separate pieces: a SIDL parser, a code generator, a small run-time support library, and a software repository. Currently, Babel supports Fortran 77, C, and C++; we plan to develop support for Java, Python, Fortran 90, and MATLAB in the following year.

The Babel parser, which is available either at the command-line or through a web interface, reads SIDL interface specifications and generates an intermediate XML representation. XML is a useful intermediate language since it is amenable to manipulation by tools such as a repository or a GUI development environment. XML interface descriptions are stored locally or in a shared web-based software repository called Alexandria. The vision is that a scientist downloading a particular software library from the repository will receive not only that library but also the required language bindings generated automatically by the Babel tools.

The Babel code generator reads SIDL XML descriptions and automatically generates glue code for a particular software library. This glue code mediates differences among calling languages and supports efficient inter-language calls within the same memory address space. The internal object representation used by the code generator is similar to that used by COM or CORBA's Portable Object Adaptor or by scientific libraries such as PETSc. The intermediate representation is essentially a table of function pointers, one for each method in an object's interface, along with other information such as internal object state data and Babel data structures. The code generators generate stub and skeleton code that translate between the calling conventions of a particular language and the intermediate representation.

4. THE ALEXANDRIA REPOSITORY

The **Alexandria** repository was designed and built to facilitate the adoption of component technology for high-performance scientific simulation software. Our goal was to provide a network service where component developers can publish their software and interface definitions and where application developers can find and download components and the language bindings needed to provide needed features. The system was intended to have a user interface to support human and machine clients.

We chose to implement a web application (i.e., a web server with dynamic content managed by a program) to achieve these goals. A web application can provide a sophisticated and friendly user interface designed for human clients and a simple, feature-rich interface for machine clients. By using web technologies, we make the repository's services available to the largest possible network audience. Machine clients can use standard network libraries to access the repository. Other network approaches would require installation of special

```
version hypre 1.0;

/**
 * A SIDL type description for the <em>hypre</em> library.
 */
package hypre {

   /**
    * <code>Vector</code> represents a mathematical vector.
    */
   interface Vector {
      void clear();
      void copy(in Vector x);
      Vector clone();
      void scale(in double a);
      double dot(in Vector x);
      void axpy(in double a, in Vector x);
   }

   /**
    * An <code>Operator</code> maps one vector into another vector.
    */
   interface Operator {
      void apply(in Vector x, out Vector y);
   }

   /**
    * This interface represents the class of linear mappings.
    */
   interface LinearOperator extends Operator {
   }

   /**
    * <code>StructVector</code> is a vector for structured grids.
    */
   class StructVector implements-all Vector {
      array<int> getGhostCellWidth();
   }

   /**
    * The structured matrix class implements all operator functions.
    */
   class StructMatrix implements-all Operator {
      // functions used to build a structured matrix omitted
   }
}
```

Figure 1 Portions of the hypre interface specification written in SIDL.

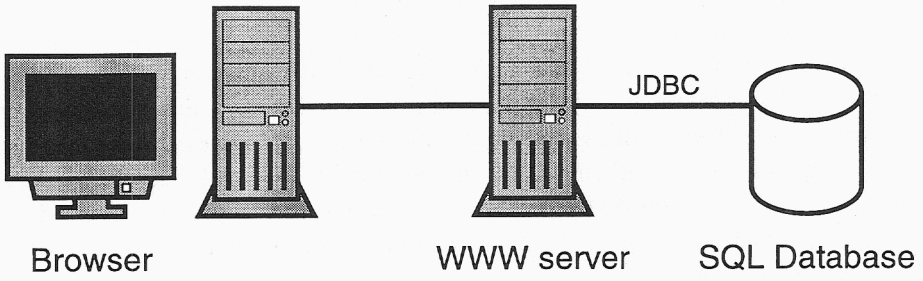

Figure 2 Alexandria architecture

purpose clients or more elaborate machine clients thereby decreasing the potential audience for the service. The HTTP protocol provides all the transaction types necessary for the repository: uploading files and other information from a user interface form and downloading content. The transactional nature of the WWW makes the user interface less interactive than a native application, but the benefits of the web interface seem to outweigh this deficiency.

As shown in Figure 2, **Alexandria** uses a three-tiered architecture: a web browser based user interface, a web server with `Java` servlets[7] and JavaServer Pages[12], and a JDBC[16] connection to an SQL backend. The web server delegates HTTP messages for certain URLs to `Java` servlets, and the servlet provides the content or error response. A servlet is a `Java` class that implements a standard interface or overrides methods inherited from a standard base class. The servlet can use all the features of the `Java` platform in generating its response. JavaServer Pages is a convenient, dynamic way to generate a servlet which usually combines HTML with embedded `Java` code to provide the dynamic content.

The web server provides an access control infrastructure to provide different levels of access to the repository. There is also a servlet interface to the **Babel** IDL processor, so clients can get language bindings for a particular package without having to download and install **Babel**.

The **Alexandria** repository is a web application to provide human and automated clients the information they need to find and use software components and libraries for scientific computing. It is an enabling technology that makes it easier to distribute and use software components. For the human client, **Alexandria** provides a hierarchically organized collection of software packages uploaded by component developers, a fuzzy search capability, an interface definition browser, and a web user interface to the **Babel** language interoperability tool. For automated clients, **Alexandria** provides a repository of XML interface definitions and will hold a repository of shared libraries which implement particular interfaces to enable dynamic graphical application builders.

People running **Babel** on their desktop can connect to **Alexandria** and access it's repository of XML type information. Users with sufficient access can translate the IDL file into an equivalent XML representation and upload the XML representation to the repository. Once it is in the repository, anyone running **Babel** can use the XML information from **Alexandria** rather than having to explicitly download all the needed IDL files. In addition, the web server provides high quality interface documentation to web browser by applying XSLT, a evolving standard for translating XML into HTML or other markup languages.

Alexandria is designed to have multiple versions of an interface each identified by a unique version number. When clients request an interface, they can either provide both the name and version number or just the name in which case they get the version with the high version number.

5. CONCLUSIONS

In collaboration with members of the HYPRE development team, we have integrated some of the Babel language interoperability technology into the HYPRE library. HYPRE is a suite of parallel scalable linear solvers and preconditioners implemented in C and MPI. There were four primary goals of this collaboration. First, the Babel team wanted to demonstrate the technology and get feedback from library developers. Second, the HYPRE project needed automatically generated Fortran bindings that would track changes in the library. Previously, a number of different Fortran bindings were developed by various users but fell into obsolescence with new changes to the HYPRE library. Third, the HYPRE team wanted to explore new design options using object-oriented and component-based software techniques, but the team had no desire to generate and support the necessary object-oriented infrastructure by hand. Finally, HYPRE developers wanted to integrate software developed by other groups who had written code in C++ and Fortran.

The project began by identifying key parts of HYPRE and developing an object-oriented design in SIDL for the primary HYPRE objects. For the most part, existing HYPRE implementations were wrapped using glue code generated by the Babel tools. In spite of this additional intermediate glue code, parallel runs with both Fortran and C drivers indicate that Babel overheads are too small to measure accurately.

HYPRE developers identified a number of advantages to using Babel techniques for a scientific software library in addition to the obvious advantage of language interoperability. Developers found that SIDL was a convenient specification description language for the design of scientific libraries because it eliminated unnecessary implementation details and forced them to focus on the object-oriented design of the library. They felt that the language was rel-

atively easy to master, although some were new to object-oriented design and object-oriented languages. Furthermore, HYPRE developers noticed that they could eliminate redundant code by taking advantage of polymorphism. For example, the previous HYPRE library contained a four different PCG (Preconditioned Conjugate Gradient) routines, each written for a particular type of preconditioner data structure. Through the use of polymorphism enabled by Babel, they were able to reduce the number of PCG routines to one. Finally, the HYPRE developers were able to exploit object-oriented design in C, which has no object-oriented support, using the automatically generated Babel code.

Acknowledgments

We would like to thank Andrew Cleary, Jeff Painter, and Cal Ribbens for integrating the **Babel** language interoperability technology into the *hypre* library and for their many useful suggestions. We would also like to thank members of the Common Component Architecture forum for numerous in-depth conversations about component technology for scientific computing.

References

[1] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high performance scientific computing. In *Proceedings the Eighth International Symposium on High Performance Distributed Computing*, 1999. See `http://z.ca.sandia.gov/~cca-forum`.

[2] David Beazley. *SWIG Users Manual.* See `http://www.swig.org`.

[3] David M. Beazley and Peter S. Lomdahl. Building flexible large-scale scientific computing applications with scripting languages. In *The 8th SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, MN, March 1997.

[4] David Brown, William Henshaw, and Daniel Quinlan. Overture: An object-oriented framework for solving partial differential equations on overlapping grids. In *Proceedings of the First Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, 1998. See `http://www.llnl.gov/CASC/Overture`.

[5] Kent G. Budge and James S. Peery. Experiences developing ALEGRA: A C++ coupled physics framework. In *Proceedings of the First Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, 1998.

[6] Julian Cummings, James Crotinger, Scott Haney, William Humphrey, Steve Karmesin, John Reynders, Stephen Smith, and Timothy Williams. Rapid application development and enhanced code interoperability using the POOMA framework. In *Proceedings of the First Workshop on*

Object Oriented Methods for Inter-operable Scientific and Engineering Computing, 1998. See `http://www.acl.lanl.gov/pooma`.

[7] J.D. Davidson and D. Coward. *Java Servlet Specification, v2.2.* Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303, USA, December 1999. Available at `http://java.sun.com/products/servlet/`.

[8] Guy Eddon and Henry Eddon. *Inside Distributed COM*. Microsoft Press, Redmond, WA, 1998.

[9] Richard Hornung and Scott Kohn. The use of object-oriented design patterns in the SAMRAI structured AMR framework. In *Proceedings of the First Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, 1998. See `http://www.llnl.gov/CASC/SAMRAI`.

[10] Microsoft Corporation. *Microsoft .NET Platform.* Available at `http://www.microsoft.com/net`.

[11] Object Management Group. *The Common Object Request Broker: Architecture and Specification.* Available at `http://www.omg.org/corba`.

[12] E. Pelegrí-Llopart and L. Cable. *JavaServer Pages Specification: Version 1.1.* Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303, USA, December 1999. Available at `http://java.sun.com/products/jsp/`.

[13] Sun Microsystems. *Enterprise JavaBeans Server-Side Component Architecture.* See `http://java.sun.com/products/ejb`.

[14] Sun Microsystems. *JavaBeans Component Architecture Documentation.* See `http://java.sun.com/products/javabeans/docs`.

[15] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[16] S. White and M. Hapner. *JDBC 2.1 API.* Sun Microsystems, Inc., October 1999. Available at `http://java.sun.com/products/jdbc/`.