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 software, such as complex numbers, dense multidimensional arrays, and parallel distributed objects. Preliminary results indicate that in addition to language interoperability, this approach provides useful tools for the design of modern object-oriented scientific software libraries. 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 understanding 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, and physical deployment. By removing such barriers, component approaches will allow computational scientists to concentrate on building more sophisticated numerical simulations and reduce effort wasted integrating incompatible software.

In this paper, we describe our recent work in two areas of component technology: language interoperability and a component repository. As part of our language interoperability efforts, we are developing 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 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, the **Babel** tool, and experiences using **Babel** in a high-performance scientific software library. Section 4 introduces the **Alexandria** web-based component repository and its implementation architecture. Finally, Section 5 summarizes the contributions of this work and discusses future research directions for the scientific component community.

2. SCIENTIFIC COMPONENT TECHNOLOGY

Component technology [25] is an extension of object-oriented software technology that focuses on the issues of software interoperability and re-use. Component technology provides language independence, compiler independence, and seamless access to distributed object resources. Component technology is more than object-oriented approaches, software modules, scripting [3, 4], or software frameworks [7, 8, 10, 14]; however, component approaches do make use of these other related technologies. 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 [12], JavaBeans [24], and CORBA [19]. COM 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 [18] that combines ideas from COM and `Java` and will likely be the future of Microsoft technology. Sun Microsystems has developed JavaBeans and Enterprise JavaBeans [23] based on the `Java` programming language. CORBA, by the Object Management Group (OMG), is a cross-platform distributed object specification that supports the interaction of complex objects written in different programming languages distributed across a network of computers.

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. The work presented in this paper addresses only a small part of the overall component technology solution. Community collaborative work such as that by the Common Component Architecture (CCA) [1] forum and others is essential. In the following, we review related component technology work in the scientific community.

Both CORBA [19] and COM [12] address language interoperability through the use of an Interface Definition Language (IDL). An IDL describes the interface of a software component using a new descriptive language that is independent of any particular programming language. We follow a similar approach in our language interoperability work, which is presented in Section 3. IDL technology has the advantage that, in some sense, all languages are equal, and any language may call any other language. The primary disadvantage of an IDL approach is that the developer must write a separate interface description of the software library and then must follow certain programming conventions that map the interface description into the programming language. Automatic wrapping approaches such as SWIG [3] or SILOON [17] support language interoperability without requiring a separate IDL description but are typically limited to the case of a scripting language (such as `Python`) calling a compiled language (such as `C` or `C++`). In contrast, IDL approaches allow method invocations in both directions.

Beyond language interoperability, component architectures typically require that all components support some common set of behaviors. Common behaviors are important for the discovery of component capabilities (e.g., *"What interfaces do you export?"*) required by GUI development tools and problem solving environments [6, 13, 20]. For example, the CCA specification requires that all CCA components support the notion of a *port* [1]. Ports describe the interfaces used by and provided by a component. Our IDL technology plays a role as a mechanism for describing component port interfaces.

Component problem solving environments (PSEs) may also require standards for describing the physical deployment of component software. For example, CCAT [6] employs an XML [28] component deployment descriptor that enables the PSE to understand component ports, port interface types, platform dependencies, and associated component metadata. One of the goals of the **Alexandria** component repository described in Section 4 is to provide a common repository for component descriptions for use by tools such as a PSE.

Unlike industry approaches, scientific component technology must support communicating parallel components. In most high-performance applications, components will communicate within the same memory address space, although the components themselves may be distributed across processor memories in a SPMD fashion. Some applications, however, will span multiple parallel computers. For example, a large simulation running on thousands of processors may be connected to a visualization component running on a small visualization engine with a few tens of processors. In this case, the component architecture must support some form of parallel data redistribution. A number of researchers have addressed this issue for certain limited classes of data types. Both PAWS [5] and CUMULVS [16] support parallel redistribution of arrays and other predefined data items such as particles or simple unstructured meshes. PARDIS [15] and Cobra [22] support distributed sequences and arrays in CORBA. We and other members of the CCA working group are researching approaches for extending this work to more general scientific objects, but that work is preliminary and beyond the scope of this paper.

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`, `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.

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, description, and distribution of language independent software libraries. In the following sections, we describe our interoperability approach, the **Babel** tool architecture, and an example of using **Babel** in a parallel linear algebra software library.

3.1. SCIENTIFIC IDL

Babel addresses the language interoperability problem using Interface Definition Language (IDL) techniques [12, 19]. An IDL describes the calling interface (but not the implementation) of a particular software library. IDL tools use this interface description to generate "glue code" that allows a software library implemented in one supported language to be called from any other supported language. We have designed a Scientific Interface Definition Language (SIDL) that addresses the unique needs of parallel scientific computing. SIDL supports complex numbers and dynamic multi-dimensional arrays as well as parallel communication directives that are required for parallel distributed components. SIDL also provides other common features that are generally useful for software engineering, such as enumerated types, symbol versioning, name space management, and an object-oriented inheritance model similar to `Java`.

As illustrated in Figure 1, SIDL bears a close resemblance to CORBA and `Java`. The `package` keyword introduces a new namespace. A namespace may contain a class, interface, enumerated type, or another package. Classes and interfaces contain methods. The methods in an interface are abstract; that is, they are not implemented by the interface. As in CORBA, `in`, `out`, and `inout` modify method arguments and denote the direction of information transfer. SIDL also supports `Javadoc`-style documentation comments, which may be used to automatically generate browsable documentation (see the **Alexandria** discussion in Section 4).

```
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 {
      Vector clone();
      void scale(in double a);
      double dot(in Vector x);
      void axpy(in double a, in Vector x);
      int getGlobalDimension();
      int getLocalDimension() local;
   }

   /**
    * 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 {
      // methods used to build a structured matrix omitted
   }
}
```

Figure 1 A simplified SIDL interface description for portions of the *hypre* software library described in Section 3.3.

The following sections provide additional details concerning some of the more unique characteristics of the SIDL interface definition language.

3.1.1 Symbol Versioning. In SIDL, every package, enumerated type, class, and interface is assigned a particular version number. Every SIDL description begins with one or more `version` statements. Each `version` statement contains a package name and an arbitrary version string consisting of a sequence of integers separated by periods. All symbols within a package share its version number. For example, the `version` statement on the first line of Figure 1 states that all symbols defined in the `hypre` package will be version 1.0 of that symbol. A `version` statement is required for every new outermost package defined in a SIDL description. A `version` statement may also be used to give an explicit version number for resolving external symbols referenced in a SIDL description. If a version is not specified for a particular external symbol, then the most recent version of that symbol is used.

Symbol versioning is an important consideration for the development of community-wide standards and specifications. Consider a standards committee that releases version 1.0 of a particular specification. Components will be written to and implement that version of the standard. When the committee releases version 2.0 of the specification, some components will immediately implement the new standard, whereas others will take longer. Versioning removes ambiguity about which version of the specification a particular component implements.

3.1.2 Import. Like `Java`, SIDL supports a type of `import` statement. The `import` statement adds the specified package name to the symbol resolution path. For example, a SIDL description that references symbol `Vector` in package `hypre` could either use the fully qualified name `hypre.Vector` or begin with "`import hypre`" and then simply use the name `Vector` (assuming, of course, that another `Vector` did not already exist in that name scope). External symbol references are resolved by searching an associated symbol repository, either a file repository or a web-enabled repository such as **Alexandria**.

3.1.3 Inheritance Model. The SIDL inheritance model is similar to that of `Java`. SIDL supports both interfaces and classes. The methods in an interface are abstract and thus not implemented by that interface. The methods in a class may be either abstract or implemented by that class. SIDL supports multiple inheritance of interfaces but single implementation inheritance of classes. An interface may extend other interfaces. A class may implement many interfaces but extend only one other class. This inheritance model simplifies the **Babel** implementation and removes the diamond implementation inheritance ambiguity associated with C++. Like COM [12], all classes and interfaces implicitly inherit from a common base interface that provides reference counting and simple query interface capabilities.

Based on suggestions from our users, we have augmented the Java inheritance syntax with an implements-all keyword, which declares that the associated class implements all of the methods in the specified interface. This keyword is equivalent to using the implements keyword and repeating the definition of all interface methods in the class body. The implements-all shorthand is cleaner and more closely reflects the way many of our users think about designing scientific libraries. They typically define abstract interfaces that describe the desired functionality and then combine those interfaces together into classes and components that implement that functionality.

3.1.4 Arrays. SIDL supports the style of dynamically-sized, dense, multi-dimensional arrays that are common in scientific applications. Existing IDLs such as CORBA [19] support only dynamically-sized, one-dimensional arrays (a CORBA sequence) and statically-sized, multi-dimensional arrays. Dense arrays consist of one physical segment of memory that can be accessed efficiently by an optimizing compiler. Such arrays are common in the scientific community due to its Fortran heritage and because dense arrays offer better access performance than "array of array" implementations.

3.1.5 Parallelization Support. We have just begun to develop support for parallel data redistribution in the **Babel** tools. Therefore, the following discussion should be considered preliminary, although it does indicate our basic approach. SIDL currently supports parallel communication directives that describe method behavior in a parallel execution environment. For example, the local method modifier in class Vector of Figure 1 indicates that the getLocalDimension method is valid only when invoked on an object in the same memory address space. For this method, the number of local vector elements owned by a particular processor has no meaning for a Vector object distributed across a different set of processors.

Unlike PARDIS [15] and Cobra [22], we do not intend to add data distribution directives to the SIDL language. We do not believe that static IDL data distribution directives will be sufficient to describe the dynamic complexity and wide range of parallel objects used in scientific computing. Instead, we plan to use run-time data descriptions of data objects. Distributed parallel objects will be required to support one of a set of data distribution interfaces through which the object describes its internal data distribution state. The **Babel** runtime will use that information to manage data redistribution during method invocations. We feel this approach is more appropriate for sophisticated data decompositions that change during the course of a simulation.

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 the **Alexandria** component 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 the **Alexandria** web interface, reads SIDL interface specifications and generates an intermediate XML [28] representation. XML is a useful intermediate language since it is amenable to manipulation by tools such as a repository or a problem solving environment. XML interface descriptions are stored either in a local file repository or on the web using **Alexandria**. The vision is that a scientist downloading a particular software library from the **Alexandria** component 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 the specified software library. This glue code mediates differences among calling languages and supports efficient inter-language calls within the same memory address space and, eventually, across memory spaces for distributed objects. The code generators create four different types of files: stubs, skeletons, **Babel** internal representation, and implementation prototypes. The **Babel** internal object representation created by the code generators is similar to that used by COM [12], CORBA's Portable Object Adaptor [19], and scientific libraries such as PETSc [2]. The internal object 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, parent classes and interfaces, and **Babel** data structures. Stub and skeleton code translates between the calling conventions of a particular language and the internal **Babel** representation. The code generators also create implementation files that contain function prototypes to be filled in by the library developers. To simplify the task of library writers, we have added automatic `Makefile` generation as well as a "code splicing" capability that preserves old edits during the regeneration of implementation files after modifications to the SIDL source.

3.3. TECHNOLOGY DEMONSTRATION IN HYPRE

In collaboration with members of the *hypre* development team, we have integrated some of the **Babel** language interoperability technology into *hypre* [9]. The *hypre* library is a suite of parallel scalable linear solvers and preconditioners implemented in `C` with MPI. There were four primary goals of this collaboration. First, the **Babel** team wished 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* source. 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 collaboration 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.

The developers of *hypre* identified a number of advantages to using **Babel** for their 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 SIDL was relatively 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 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 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.

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 associated language bindings. The system was intended to have a user interface to support human and machine clients.

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 machine 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 or other development tools.

We chose to implement a web application (i.e., a web server with dynamic content managed by a program) to achieve these goals and features. 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; any contemporary web browser can access **Alexandria**. Machine clients can use standard network libraries to access the repository. Other network approaches would require installation of special 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 web 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 [11] and JavaServer Pages [21], and a JDBC [26] connection to an SQL backend. The web server delegates HTTP messages for certain URLs to `Java` servlets, and the servlet provides the content or an 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 **Alexandria** application consists of five subsystems: an access control system, an inexact string matching package, a hierarchy management system, a content package, and an interface to **Babel**. The access control system manages user accounts and provides several different levels of access to the system: administrator, trusted user, normal user and world. The inexact string matching package is a Java implementation of the algorithm from `agrep` [30].

The hierarchy management system provides cataloging, uploading and downloading features. Unlike a normal file system, the hierarchy can hold files with the same name in a common directory as long as they have different version numbers. The expectation is that over time a project will issue multiple versions of individual files.

The content scanning package checks material provided by users to see if it is "safe content." A responsible web server that receives content from users and

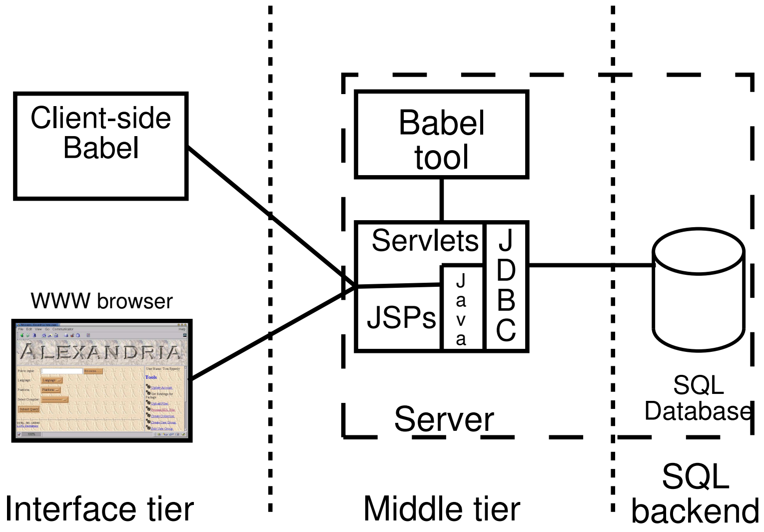

Figure 2 Alexandria architecture

then presents that content back to other users must verify that the user provided material does not contain hostile scripts. Rather than trying to characterize and detect hostile content, **Alexandria** tests user provided content against an XML DTD that contains a safe subset of XHTML 1.0 [27]. A validating XML parser is used to determine if user provided content is safe. If the material does not validate, all the mark-up directives are transformed so they will be interpreted as plain text rather than as mark-up directives.

The interface to **Babel** subsystem provides language bindings for a SIDL file to users. The user's SIDL file is uploaded to the web server, the web server runs **Babel** on the file, the results are packaged in a `TAR` file, and then the user is given the chance to download the file. This saves users from having to install **Babel** and a Java virtual machine on their local machine.

Alexandria maintains a repository of XML type information. Users with sufficient access can translate the SIDL 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 SIDL files. In addition, the web server provides high quality interface documentation to web browser by applying XSLT [29], a evolving standard for translating XML into HTML or other markup languages.

5. CONCLUSIONS

In this paper, we have described two pieces of a component technology architecture for scientific computing. **Babel** is a language interoperability tool that uses the SIDL interface description language to describe component interfaces and to generate code that mediates differences between programming languages. **Alexandria** is a web-enabled component repository that provides a browsable software library, automated access to SIDL type information, and web access to the **Babel** code generators.

Obviously, much work remains in developing production-quality component technology for the scientific computing community. Members of the Common Component Architecture working group have made some initial progress in this direction and have drafted a proposal that covers common behavior standards for components [1]. A number of interesting open research questions remain in extending current parallel data redistribution approaches [5, 15, 16, 22] to arbitrary data components.

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. Curfman-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] S. Balay, W. D. Gropp, L. Curfman-McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997. See `http://www.mcs.anl.gov/petsc`.

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

[4] D. M. Beazley and P. S. Lomdahl. Building flexible large-scale scientific computing applications with scripting languages. In *The 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[5] P. Beckman, P. Fasel, W. Humphrey, and S. Mniszewski. Efficient coupling of parallel applications using PAWS. In *Proceedings of*

the High Performance Distributed Computing Conference, 1998. See http://www.acl.lanl.gov/paws.

[6] R. Bramley, K. Chiu, C. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yechuri. A component based services architecture for building distributed applications. In *Proceedings of the High Performance Distributed Computing Conference*, 2000. See http://www.extreme.indiana.edu/ccat.

[7] D. Brown, W. Henshaw, and D. 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.

[8] K. G. Budge and J. 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.

[9] E. Chow, A. J. Cleary, and R. D. Falgout. Design of the *hypre* preconditioner library. In *Proceedings of the First Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, 1998.

[10] J. Cummings, J. Crotinger, S. Haney, W. Humphrey, S. Karmesin, J. Reynders, S. Smith, and T. 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.

[11] J.D. Davidson and D. Coward. *Java Servlet Specification, v2.2*. See http://java.sun.com/products/servlet/.

[12] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, Redmond, WA, 1998.

[13] D. Gannon, R. Bramley, T. Stuckey, J. Villacis, J. Balasubramanian, E. Akman, F. Breg, S. Diwan, and M. Govindaraju. Component architectures for distributed scientific problem solving. In *IEEE Computational Science and Engineering*, 1998.

[14] R. Hornung and S. 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.

[15] K. Keahey and D. Gannon. PARDIS: A parallel approach to CORBA. In *Proceedings of the Sixth IEEE Symposium on High Performance Distributed Computation*, 1997.

[16] J. Kohl and P. Papadopoulos. Efficient and flexible fault tolerance and migration of scientific simulations using CUMULVS. In *Second SIGMETRICS Symposium on Parallel and Distributed Tools*, 1998. See `http://www.epm.ornl.gov/cs/cumulvs.html`.

[17] Los Alamos National Laboratory. *SILOON: Scripting Interface Languages for Object-Oriented Numerics*. Available at `http://www.acl.lanl.gov/siloon`.

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

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

[20] S. G. Parker, D. M. Beazley, and C. R. Johnson. *The SCIRun Computational Steering Software System*. E. Arge, A.M. Bruaset, and H.P. Langtangen (Eds.), Modern Software Tools in Scientific Computing, Birkhauser Press, 1997.

[21] E. Pelegrí-Llopart and L. Cable. *JavaServer Pages Specification: Version 1.1*. See `http://java.sun.com/products/jsp/`.

[22] T. Priol, C. René, and G. Alléon. Programming SCI clusters using parallel CORBA objects. In *SCI-based Cluster Computing*. Springer Verlag, 1999.

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

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

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

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

[27] World Wide Web Consortium. *The Extensible HypreText Markup Language*. See `http://www.w3c.org/TR/xhtml`.

[28] World Wide Web Consortium. *Extensible Markup Language (XML)*. See `http://www.w3c.org/XML`.

[29] World Wide Web Consortium. *XSL Transformations (XSLT) Version 1.0*, 1999. Available at `http://www.w3.org/TR/xslt/`.

[30] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.