Component Metadata Management and Publication for the Grid

Diego Puppin, Fabrizio Silvestri, Domenico Laforenza
ISTI - CNR
via Moruzzi, 56100 Pisa, Italy
diego.puppin@isti.cnr.it
http://hpc.isti.cnr.it/

Abstract

There is growing attention to component-oriented software design of Grid applications. Within this framework, applications are built by assembling together independently developed software components.

Two main approaches are commonly used to manage, develop and publish software components: one is based on an Interface Description Language (IDL); the other is typical, for instance, of Java and is based on introspection and design conventions.

In this paper, we compare them and we propose a third approach that merges the flexibility and fast learning curve of the latter, with the rigor of the former. Our proposal is meant to help the transition towards more modern tools, which is required to develop versatile Grid applications.

1. Introduction

In an emerging Grid economy [6], where supply and demand of computing resources are driven by a market economy, we envision an open marketplace of components, where an application developer can find and pay for the needed services. To create this market of software components, it is important to develop a way to automate the management, creation and publication of software components from existing code, quickly and efficiently.

In this market, the way software components are managed, defined and modeled has a strong impact on the speed of a transition to a component-oriented application design. This is why, in this paper, we analyze two different approaches to component modeling, in terms of performance, ease-of-use and possibility of automation, and we propose a new approach.

This modeling approach should lead to the definition of a set of meta-information elements, i.e. meta-data, that will result useful in many application areas, such as component discovery and linking.

The rest of this paper is structured as follows. In the next section, we describe two existing component frameworks, which we consider particularly representative in their approach to components. Then, we consider their pros and cons, with some quantitative analysis. Next, we highlight our proposal of a third approach to component modeling and linking. We give our conclusion in the last section, after a brief survey of some related works.

Preliminary results of this work were presented at the poster session of CCGRID 2004 in Chicago [13].

2. Two Examples of Component-oriented Frameworks

Here, we analyze in detail two existing frameworks, which are representative, in our opinion, of two approaches to component modeling. One is based on a rich Scientific Interface Definition Language (SIDL), and its goal is to offer language interoperability for building large component-based scientific applications. The second is based on Java, and leverages its modern features (introspection, object-oriented design...) to provide a simple and portable programming environment.

2.1. The Common Component Architecture (CCA)

CCA [4] is a proposed standard aimed at promoting the integration of heterogenous software components. It defines a set of minimal services to be offered by a component: *Port, getPort, setServices*. CCA is implemented by a variety of frameworks, offering different programming models.

In the CCA vision, a component is a software entity that allows different languages, different hardware/software architectures to co-operate. It supports a structured, modular approach to software development, very useful in the case of third-party libraries and external services. Every component exports one or more input and output ports (*uses* and *provides* ports), identified by their type. Inheritance and specialization allows the programmer to refine definition and semantics of a component. Also, multiple ports can offer a different point of view of the same service.

The CCAffeine Framework. CCAffeine [5] is a CCA-compliant framework oriented to SPMD (or SCMD — Single Component Multiple Data) applications. It can assemble together heterogeneous, internally parallel components. As an example, in Figure 1, three parallel components

IEEE COMPUTER SOCIETY

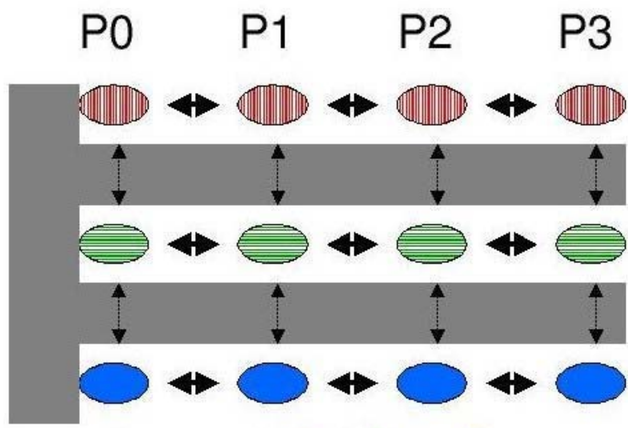

Figure 1. Communications among and within components (from [5]).

(the first with vertical stripes, the second with horizontal stripes, and the third has solid color) build up an application. Intra-processor communication, among different components (vertical arrows) is managed by the framework: it is responsible of data conversion if the components are heterogeneous. On the other side, inter-processor communication, within a parallel component (horizontal arrows), is responsibility of the component developer, who can use the tool of his/her choice.

The framework is responsible for instantiating a copy of the application graph on every machine in the computing environment. CCAffeine internally uses MPI and creates the requested components on all machine in the MPI world. The *run* command will start, in parallel, execution on each machine. Over all, CCAffeine offers a language-independent wrapping to parallel components (mostly, MPI-based). Thus, it gives support to the usage of heterogenous development environments.

Babel. Babel [1] is a SIDL language oriented to high-performance scientific programming. It is aimed at supporting the co-operation of software written with different languages. It offers a set of features similar to CORBA [2], but enriched by scientific-oriented structures, among which support for dynamic multi-dimensional arrays, and complex numbers.

As other IDLs, Babel works as an interface compiler. Given the interface description, Babel creates server- and client-side support: it exports a set of stubs to be used to call the implemented object from a program written in any other supported language. The process of creating a component from the Babel interface requires a complex series of steps: (1) definition of the component interface using Babel; (2) implementation of the component in the Babel-generated Impl file (in any supported language); (3) definition of the CCA ports in the Impl file; (4) creation of a wrapper and a CCA definition file; (5) creation of a suitable makefile.

In our experience, this process caused problems of software engineering and was error-prone. In particular, there are some naming conventions that needs to be respected in the development of the wrapper and CCA files (fourth step), which can be broken and are not recognized properly by the development tools.

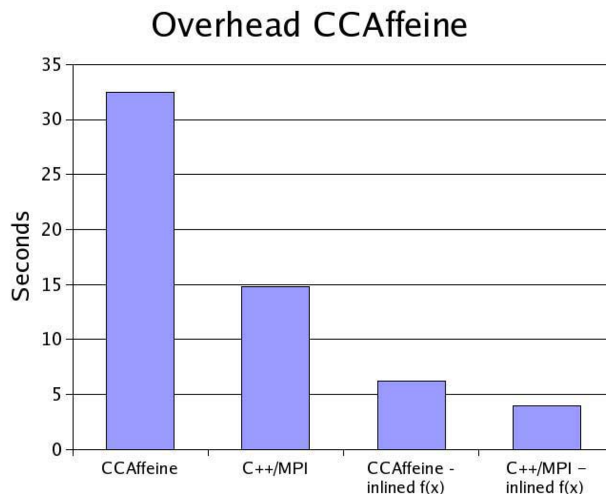

Figure 2. Comparison of CCAffeine and native code. The first column is CCAffeine implementation; the second is C++/MPI; the third is CCAffeine with inlined function; the fourth is C++/MPI with inlined function.

Performance. In our tests, we registered huge performance overhead, even when homogeneous components were communicating. In Figure 2, we show a comparison of a C++ implementation vs. a component implementation of a simple integration program. A parallel MPI integrator distributes function evaluations among a set of workers. Partial results are then *reduced* to compute the final sum. The data flow for the CCAffeine implementation is: a driver component invokes the services of a generic parallel integrator, which invokes the method `evaluate()` of a function component. The parallel integrator is responsible for orchestrating the final reduction. In the C++ version, components are replaced by functions, and service invocations are replaced by function calls.

A second version was tested, where the function was inlined in the integrator body. When the function to be integrated is an autonomous component, a big overhead for the function call can be observed. When the function is inlined, the framework start-up time dominates. Tests were performed on a cluster of eight dual-Pentium machines, running Linux RedHat, Babel 0.8.4, and CCAffeine 0.4.1.

In Section 3.2, we show more comparison between this framework and native code. Nonetheless, we want to highlight here an important feature of Babel arrays. As well known, an important factor affecting performance when manipulating matrices is the ordering of column and rows. If the order is correct, elements can be accessed sequentially, other-

Matrix size	RCR	CRR	CRC	Transp.
16	0.06	0.02	0.02	0.00
32	0.08	0.03	0.03	0.00
64	0.12	0.1	0.09	0.10
128	0.37	0.31	0.29	0.31
256	1.62	2.99	2.65	2.30
512	12.11	54.95	54.13	62.32

Table 1. Matrix multiply, with different orderings. Time in seconds.

wise a more complex access is needed. Babel allows the programmer to define the order to be used by the data passed to a component: a component can choose to access row-major or column-major matrices. In row-major mode, rows can be scanned sequentially, increasing the column number. If the data is ordered differently from what is expected, the system is responsible for transposing it, a costly operation for large matrices. In Table 1, one can compare the cost of a matrix multiply for some choices of orders for the matrices A, B and C (with $C = A \times B$). R stands for row-major, and C for column-major. The last column indicates what the cost is when a matrix is transposed because it does not match the parameter definition.

According to the developers, component overhead is "negligible [...] for component implementation and abstract interfaces when using appropriate levels of abstraction." Nonetheless, they also measure that parameter passing that can be three times slower than native code [8].

2.2. Java Beans

Java Beans API [7] is a mature software standard developed with the goals of offering a portable, simple, high-quality API for component software development. Each Java Bean is a reusable software component that can be manipulated visually in a builder tool. While visual manipulation is an important part of software development, beans can be more complex and can be used to model larger software entities: data-bases, observational instrumentations etc.

The most advanced and useful features of beans are introspection and the event-driven programming model.

Introspection is a very powerful tool for component publication and connection. Java classes are able to analyze their own structure, and publish their interface. Java Beans' implementation of introspection offers a more advanced set of information methods that are able to retrieve the name of the relevant methods, the set of fired events, the interesting properties.

If desided, an explicit `BeanInfo` service can be provided by the Bean, e.g. if the internal properties of the Bean are to be hidden or abstracted (overriding introspection).

Java Beans are mostly event-driven. This means that they can react asynchronously to particular circumstances that happen during application execution, for example, clock ticks or user input. Moreover, Beans can be multi-threaded, and can operate on multiple requests at a time.

Design Time and Run Time. Java Beans recognize two important moments in the application development. *Design time* is the moment when the bean is manipulated, customized and composed into an application. Customization tools offered by the bean (called `Customizer`) are invoked at design time by the design framework. When the application is completed, and it is packaged to run, the design-time code is stripped out, and only the run-time code is kept in the application code. `Customizers` are available for complex beans that require long, detailed customization, or when the internal implementation needs to be abstracted or hidden.

This strategy can be very good in general for complex component: versatile configuration tools can be packaged along with the main code, and then stripped out to reach performance at run-time.

Design Conventions. The Java standard is highly based on naming and coding conventions, usually called *design patterns* [7], which simplify and coordinate the interaction of software components: these standards are highly suggested (but not mandatory) and are exploited at the moment of interaction.

One common design pattern is used to determine property names and types: when two methods, `PropertyType getSomeProperty()` and `void setSomeProperty(PropertyType s)` are found, a property `SomeProperty` of type `PropertyType` is defined, and presented to the user at design time.

This set of *design pattern* standards allow automatic tools to present suitable customizing tools for the object's properties. For instance, `BeanBox` and `BeanBuilder` will present a palette for a color attribute, a list of choice for an enumeration type and so on.

2.3. Other Component-based Environments

The *CORBA Component Model* [2] is an extension of CORBA with the goal of defining component interfaces, and objects with persistent state. CORBA is a powerful tool to perform method invocation on objects running on remote servers, based on an IDL.

Web Services are another standard to define and represent software components. The OGSI (Open Grid Service Infrastructure) forum has enriched the standard to include some more Grid-oriented services [3]. The design of a Web service is quite straightforward, and highly automated starting from Java classes. Tools are also available for other languages (mostly, C++).

3. Experimental Comparison

3.1. Qualitative Comparison

We want to highlight the main differences of the presented environments, in term of some key features.

Component publication. With Java Beans, there is no need to define interfaces and to pass them around. The JAR file storing the code for the component can be queried: the object is able to introspect and to publish its own methods and data. This is not true for CCA, where interfaces must be explicitly dealt with.

Architecture and language portability. Java byte code is designed to run on any platform that offers a Java virtual machine. It is important to remind here that, in the context of Grid computing, executable code may have to be migrated from one machine to another, and Java byte code seems to be a very flexible way of doing this.

CCAffeine components run on any CCA-compliant framework, the installation of which, at the moment, is far more complex than that of a Java virtual machine. On the other side, the components can be invoked from a variety of programming languages.

Easy visual manipulation. Java Beans and CCA components are both designed to be easily manipulated with graphical tools. This can be an interesting feature if the component is to be used in a Problem Solving Environment (PSE) or some computer-aided application designer.

Standardization. Java is one of the most wide-spread software standards. There is large availability of software, and a large number of discussion symposia. CCA is still a proposal of standard, which is competing with several other ones for large acceptance.

Type control. CCA offers a very strong type control, based on port typing and object inheritance/specialization. In Java, type matching can be sometimes overridden if methods with no arguments are used: with Java Beans, control on the type of communicating entities is limited.

Performance of scientific kernels. Java code running on a virtual machine can have lower performance than optimized C or Fortran code. CCAffeine components also incurs in a high overhead due to type conversion.

Also, most of scientific/numeric code is written in languages other than Java (Fortran, C): it can be wrapped to run within CCA with some effort, while porting to Java can be much more costly. See below.

3.2. Quantitative Comparison

The scientific community is usually hostile to the use of Java for high-performance computing. The main critique goes to the inefficiency of its most common implementation, based on byte code and a virtual machine for its execution.

Nonetheless, great advancements have been made in recent years: frequent forums are organized among research groups to share results about high-performance Java applications, including the Java Grande forum. Among other works, Moreira et al. [12] have shown that very high performance (50% to 80% of Fortran performance) can be reached by Java, using suitable classes and compiler optimizations, on numeric kernels. Kielmann et al. [9] were able to execute threads of a Java application, in parallel on different machines, in a very efficient way.

Java is, without any doubt, a tool of choice for the design of portable interfaces, web applications, data-base wrappers and so on. Nonetheless, the low throughput of data-intensive kernels limited the usage of Java in the scientific community, which seems to be more attracted by the use of SIDLs to reach interoperability.

Clearly, we do not have to overlook the high overhead imposed by the latter solution: in term of programming, and in term of performance. As shown above, the overhead to the developer is very high when s/he uses a IDL; also, parameter passing can be three times slower than native parameter passing, using Babel as an example.

To give a more quantitative evaluation of the performance of Java vs. a SIDL-based component, we considered three implementations of a simple matrix multiply. One is a very simple C implementation, using pointers to store the matrices. The second defines and then uses a new Java class, which offers methods for multiplication and printing of matrices. The third uses Babel to define the interface for a CCAffeine component that performs the product of two given matrices, and another that prints a given matrix. The implementation is then written in C++.

The Java implementation is a single file of 68 lines. In C, 81 lines are needed. Babel generates 202 files of stubs, skeletons and implementations. 8 of them were modified to incorporate the C++ implementation (adapted from the C implementation to use Babel arrays) and 4 were written from scratch.

The Java implementation is very compact, and can be run on any machine that runs a Java Virtual Machine or a Java compiler. Also, the new class could be easily wrapped as a Java Bean and then used within a Java Bean framework (such as BeanBuilder).

The components developed with Babel and CCA are expected to interoperate with any other software developed with the Babel interface, even if written in another language, and can run on any CCA-compliant framework (XCAT, CCAffeine...).

The cost to get interoperability across different programming languages through Babel is very high for the programmer, and in term of performance.

In Figure 3 we compare the CCA implementation, with Java (bytecode, bytecode with Just-In-Time compiler, compiled with `gcj` and compiled with optimizations `-O3`), with C (compiled with and without optimizations).

The overhead introduced by Babel wrapping is very high. In particular, in order to be portable, the implementation of arrays is not native and incurs high run-time performance degradation. The Java code, compiled with optimization, reaches a performance comparable to the CCAffeine implementation. As expected, both fall far behind the faster C implementation.

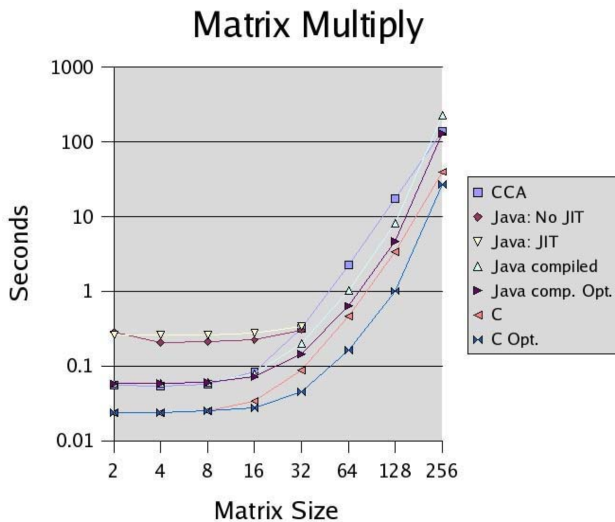

Figure 3. Comparison of matrix multiply, with CCAffeine, C, and Java.

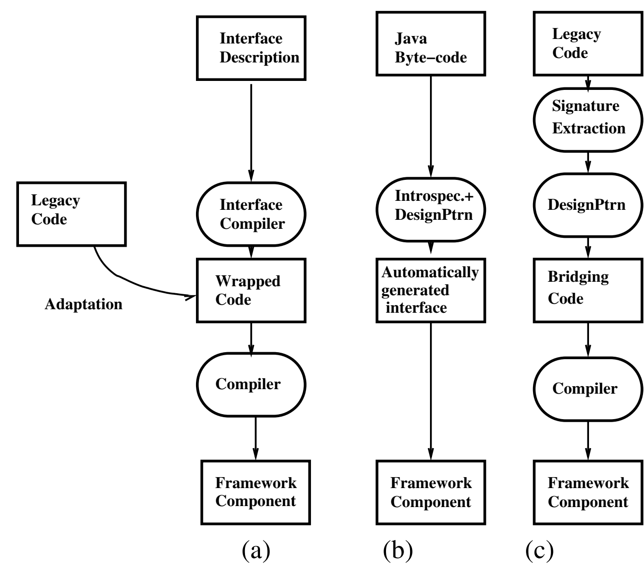

Figure 4. Different approaches to component development: (a) is based on an IDL compiler; (b) is the one followed by Java Beans; (c) is the one we propose, based on automatic manipulation of legacy code.

4. A Choice for a Component Model

As shown, two main ways are available today for component publication and management: one is based on a description language, along with compilers to generate stubs and skeletons; the second is based on object introspection and design conventions, and allows a very fast deployment of Java classes (see Figure 4).

The use of an interface description language (see Figure 4(a)), such as Babel [1] and CCM [2], can be very costly, in term of learning curve, code rewriting, and possibility of errors: in our opinion, present day interface description languages are taking an innatural approach, because they ask the developer to start writing an interface, and then to adapt the existing code to the automatically generated skeleton/stub files. This can be particularly time-consuming with legacy code.

As shown in Figure 4(b), JavaBeans proposes a different approach, based on introspection. Compiled JavaBeans classes are able to respond to specific queries about the methods they provide, presenting to the framework a full description of the available methods, and their interface (name, arguments and their type, returned value). Moreover, the Java Beans API sets a standard for more advanced tools, such as an explicit *Customizer* — presented to the designer when s/he wants to customize the Bean — and a redefined *BeanInfo* method, that overrides the automatically collected information.

We strong believe design conventions will emerge naturally with the advent of a Grid economy [6], in a way similar to what happened with the Web. Any Web-designer is free to design his/her pages in the preferred way, and any Web-browser can open them. Nonetheless, if s/he wants the web site to be discovered easily by a search engine, s/he will adhere to some HTML/XML standards, such as `META` tags, textual `ALT` tags for images and so on. Moreover, s/he will use some self-explaining description for the information provided: a car-company, for example, should add, somewhere in the web site, in a clear textual form, the words “car” and “company”.

We propose a new approach, such as the one in Figure 4(c): legacy code is analyze in order to extract the signature of functions and variables; then, design conventions are recognized in it, and are used to develop the bridging code, needed to link the software unit with the framework and the other componenents at hand. The next section explores this opportunity.

4.1. Extracting Information from Compiled Code

A direction we are investigating is the possibility of extracting information about methods signature from the compiled code, in a way similar to Java Beans. There are very effective tools for code developed with some modern languages such a C#, Lisp, Java, Visual Basic. For Java, for instance, Mocha can extract full information from the code, because Java byte-code is very rich of information.

Binary code offers a more complex challenge. To reach higher performance, debugging information and many data related to the symbol table are stripped out of the executable file. On the other side, dynamically linked libraries are usually called by name: often function names and types is available in the binary. Also, the headers sometimes offered with libraries can be parsed to reconstruct some of the information needed to use the code.

IEEE COMPUTER SOCIETY

As a matter of fact, some tools are available also for languages other than Java. `GDB` can retrieve some information, especially if debugging data are kept. Other basic utilities for binary code (known as *binutils*) are available on Unix systems.

The process we envision, is so:

- legacy code is developed with the tools of choice of the development team;
- compiled code and, if available, source code and headers, are analyzed to extract signatures of interest: methods' names and types are discovered in the code; design conventions are applied to understand the basic semantics and the relationships among functions; the application developer is offered the opportunity to choose which methods to publish, and which to keep hidden (if not clear from the code).
- this information can be used to generate, automatically, or with some computer-aided process: wrappers in other languages, so to allow inter-operability, featuring the minimal data conversion needed; WSDL descriptions, so to deploy the functions as web services; links to a graphical interface for composing functions.

5. Related Work

Several attempts have been done to perform automatic wrapping of existing code for particular frameworks, with noteworthy results, some of which are listed here.

Taylor et al. [14] developed a way to wrap C function calls in Java components, to be used within the Triana framework. Before them, Mintchev and Getov [11] wrapped C libraries so to be used within a Java program: they were able to use a C implementation of MPI from Java. Also, Li et al. [10] wrapped high-performance MPI legacy code (written in C and Fortran) into Java/CORBA components, and were able to maintain very good performance after conversion.

We agree that Java is a very versatile tool for software distribution. Nonetheless, we think that the bridging code should be generated automatically for the task at hand, skipping all the unnecessary conversion and optimizing for the components to be connected.

6. Conclusions

An effective, quick approach to component management and publication is fundamental to make component-based programming an easy task on the Grid.

Component-oriented approaches, based on IDLs, introduce high overhead in the process of wrapping existing code to be re-used. The CCAffeine framework, taken as an example, require the manipulation of several files to connect two components for a simple matrix multiplication.

This is why we push for a more natural approach, based on automatic extraction of interfaces, and then computer-aided generation of suitable bridging layers. Java Beans offer an interesting example, as they base connection and discovery on interface introspection and design conventions.

In our opinion, there is a need for tools able to analyze legacy code, extract signature information and then generate efficient bridging code. The strong results available in the field of wrapping C into Java components can be a valid starting point.

References

[1] Babel scientific interface description language. http://www.llnl.gov/ CASC/ components/ babel.html.

[2] Corba and corba component model. http://ditec.um.es/ ˜ dsevilla/ ccm/.

[3] The open grid service infrastructure forum. http://www.gridforum.org/ogsi-wg/.

[4] R. Armstron, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinsk. Toward a common component architecture for high-performance scientific computing. In *Conference on High Performance Distributed Computing*, 1999.

[5] D. E. Bernholdt, W. R. Elwasif, J. S. Kohl, and T. G. W. Epperly. A component architecture for high-performance computing. In *Workshop on Performance Optimization for High-Level Languages and Libraries*, June 2002.

[6] R. Buyya, D. Abramson, J. Giddy, and H. Stockinger. Economic models for resource management and scheduling in grid computing. *The Journal of Concurrency and Computation: Practice and Experience (CCPE) - Special Issue on Grid Computing Environments*, 14(13–15):617–630, Nov-Dec 2002.

[7] G. Hamilton. *Java Beans, API Specification, Version 1.01-A*. 1997.

[8] D. Katz, C. Rasmussen, J. Kohl, R. Armstrong, and L. McInnes. Cca tutorial at the cca forum winter meeting. http://www.cca-forum.org/tutorials/2003-01-15/index.html, January 2003.

[9] T. Kielmann, P. Hatcher, L. Boug, and H. Bal. Enabling Java for high-performance computing: Exploiting Distributed Shared Memory and Remote Method Invocation. *Communications of the ACM*, 44(10):110–117, Oct. 2001. Special issue on Java for High Performance Computing.

[10] M. Li, O. F. Rana, and D. W. Walker. Wrapping mpi-based legacy codes as java/corba components. *Future Generation Computer Systems*, 18(2):213–223, October 2001.

[11] S. Mintchev and V. Getov. Automatic binding of native scientific libraries to Java. In *Proceedings of ISCOPE*, pages 129–136, Marina del Rey, California, December, 1997. Springer LNCS 1343.

[12] J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence. Java programming for high performance numerical computing. *IBM Systems Journal*, 39(1):21–, 2000.

[13] D. Puppin, F. Silvestri, and D. Laforenza. An evaluation of component-based software design approaches. In *CCGRID 2004, Poster session*, 2004.

[14] I. Taylor, R. Davies, and H. Marzi. Automatic wrapping of legacy code and the mediation of its data. In *Proceedings of the UK eScience All Hands Meeting*, September 2002.