Preprint
UCRL-JC-134260

Language Interoperability for High-Performance Parallel Scientific Components

N. Elliott, S. Kohn, B. Smolinski

This paper was prepared for submittal to the
International Symposium on Computing in Object-Oriented Parallel Environments, San Francisco, CA, September 29 – October 2, 1999

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

May 18, 1999

Language Interoperability for High-Performance Parallel Scientific Components

Noah Elliott, Scott Kohn, Brent Smolinski

Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA.

Abstract. With the increasing complexity and interdisciplinary nature of scientific applications, code reuse is becoming increasingly important in scientific computing. One method for facilitating code reuse is the use of components technologies [16, 17, 9], which have been used widely in industry. However, components have only recently worked their way into scientific computing [2, 1, 11, 18]. Language interoperability is an important underlying technology for these component architectures. In this paper, we present an approach to language interoperability for a high-performance parallel, component architecture being developed by the Common Component Architecture (CCA) group.[1] Our approach is based on Interface Definition Language (IDL) techniques[6]. We have developed a Scientific Interface Definition Language (`SIDL`), as well as bindings to `C` and `Fortran`. We have also developed a `SIDL` compiler and run-time library support for reference counting, reflection, object management, and exception handling (`Babel`). Results from using `Babel` to call a standard numerical solver library (written in `C`) from `C` and `Fortran` show that the cost of using `Babel` is minimal, where as the savings in development time and the benefits of object-oriented development support for `C` and `Fortran` far outweigh the costs.

1 Introduction

Component technologies and component programming methodologies are beginning to work their way into the scientific community [2, 1, 11, 18] in the hopes of facilitating code reuse. One group developing a component architecture for high-performance, parallel computing is the Common Component Architecture group. An integral part to this component architecture is a mechanism for language interoperability. All components that operate within a component architecture should adhere to a standard behavior, which includes being able to easily interoperate with software written in other languages. With the proliferation of languages used for numerical simulation in recent years, like `C`, `C++`, `Fortran 90`, `Fortran 77`, `Java`, and `Python`, language interoperability can be a huge barrier to developing components, as well as developing reusable scientific applications and libraries.

[1] The CCA group consists of representatives from DOE laboratories and academia working towards the specification of a component architecture for high-performance scientific computing

Getting the many languages used in scientific computing to interoperate can be a difficult problem for developers. For both component and library developers, the choice of implementation language may severely limit the reuse of their software. Without language interoperability, users of components may be required to adopt the language of the component for future applications development, even though better alternatives may exist. If language interoperability is desired, component developers and users would be forced to write "glue" code that mediates data representations and calling mechanisms between languages. However, this approach is labor-intensive and in many cases does not provide seamless language integration across the various calling languages. `Fortran 90` is a particular challenge for language interoperability, since `Fortran 90` calling conventions vary widely from compiler to compiler.

1.1 Pairwise Approaches

There have been attempts at automatically generating glue code to support calls among a small set of targeted languages. For example, the SWIG package [3] reads `C` and `C++` header files and generates glue code so that these routines can be called from scripting languages such as `Python`. Pyffle [19] is similar to SWIG except that it provides seamless integration of `Python` and `C++`. The problem with these approaches is that they either don't provide two-way interoperability between the scripting language and the target language, or all calls between languages must occur through the scripting environment, which makes them inappropriate for a high-performance component architecture. For instance, if a simulation package written in `C` wanted to call a numerical solver package written in `Fortran 77` the package would have to make the call through the scripting environment. This would be much too inefficient for general use in scientific computing. These methods are not general enough to support a high-performance component architecture.

Foreign invocation libraries, such as *Java Native Interface* [14], have been used to handle interoperability between two targeted languages. For instance, the *Java Native Interface* defines a set of library routines that enables `Java` code to interoperate with applications and libraries written in `C` and `C++`. The problem with this type of approach is that given N languages, $O(N^2)$ different software packages would be needed to get all the languages to interoperate. Again, this is not general enough to support a high-performance component architecture.

1.2 IDL Approach

One interoperability mechanism used successfully by the distributed systems and components community [16, 13, 17, 20] is based on the concept of an Interface Definition Language or IDL. The IDL is a new "language" that describes the calling interfaces to software packages written in standard programming languages such as `C`, `Fortran`, or `Java`. Given an IDL description of the interface, IDL compilers automatically generate the glue code necessary to call that software component from other programming languages.

This approach shows promise, however, current IDL implementations are not sufficient for specifying interfaces to single-program multiple-data (SPMD) type of components. First, standard IDLs such as those defined by CORBA and COM do not include basic scientific computing data types such as complex numbers or block style dynamic multidimensional arrays. Second, all of these approaches do not provide support for high-performance same address space function calls for all the programming languages needed in scientific computing. Our goal was to make the overhead of calls through the `SIDL` about as expensive as the invocation of a `C++` virtual function. Third, some of these approaches don't have support for true multiple inheritance (e.g. COM does not support multiple inheritance and implementation inheritance is done with composition or aggregation, which can be computationally expensive or difficult to implement), and those that do have support use a limited object model (e.g. CORBA does not support method overriding and their implementation of multiple inheritance is prone to method name collisions). It is important that an IDL supports true multiple inheritance to allow specification of standards for numerical library interfaces, like the Equation Solver Interface (ESI) specification [10].

We have used an IDL approach for handling language interoperability in a scientific computing environment. We have developed a Scientific IDL (`SIDL`) as well as a run-time environment (`Babel`) that implements bindings to `SIDL` and provides support necessary for a component architecture, like reflection . Currently `SIDL` has bindings to `C` and `Fortran 77`. `Babel` implements those bindings on Solaris and AIX, with plans to port them to most major platforms. Preliminary efforts have shown that `SIDL` is expressive enough for scientific computing and that the binding implementations are fast.

This paper is organized as follows. Section 2 describes the features of `SIDL` that are necessary to support high-performance parallel computing. Section 3 describes the bindings of `SIDL` to `C` and `Fortran 77`, as well as `Babel` run-time environment, which includes a `SIDL` compiler and library support. Section 4 gives the results from wrapping a standard solver library with `Babel` and calling it from both `C` and `Fortran`. Finally, we conclude in Section 5 with an analysis of the lessons learned while wrapping *hypre* and identification of future research and additions to `Babel`.

2 Scientific Interface Definition Language

For an IDL approach to work in the scientific domain, the IDL must be sufficiently expressive to represent the abstractions and data types common in scientific computing, such as dynamic multidimensional arrays and complex numbers. Additionally, the IDL must have an object model that supports true multiple inheritance. This is necessary for satisfying the CCA component architecture specification as well as interface standardization efforts like those being implemented by the ESI. The IDL should also provide error handling mechanisms which are robust and efficient. Unfortunately, no standard IDL currently exists that supports all of these, since most IDLs have been designed for operating systems [7, 8]

or for distributed client-server computing in the business domain [13, 17, 20]. However, SIDL does borrow heavily from the CORBA IDL [17] and the Java programming language [12]. Some of the features SIDL provides are an object model similar to Java, language constructs necessary for scientific computing like complex numbers and dynamic multi-dimensional arrays, and an error handling mechanism that is a cross between Java and CORBA's exception handling mechanisms. Also, implicit constructs in SIDL allow SIDL implementation environments, like Babel, to provide reflection capabilities, which is a necessary feature for component architectures.

2.1 SIDL Object Model

Currently, interfaces and classes are the only two user defined types in SIDL. SIDL adopts the same object model as the Java programming language. The Java object model is advantageous because it is well defined, where other models, like those used in C++ and CORBA, are not as well defined. For instance, in C++, a class can inherit from multiple non-abstract classes. This poses a problem if any two or more of the parent classes have method(s) with the same signature. Java avoids this problem by only allowing single implementation inheritance and multiple interface inheritance.

All methods are equivalent in semantics to C++ virtual functions. Methods can be overridden by child classes, which means the methods in all the parent classes and interfaces, which have the same signature as the method in the child class, will be defined by that method in the child class. Methods can also be declared **abstract**, **final**, or **static**. An **abstract** method is purely declarative and has no implementation provided for it. When a method is declared **abstract**, the class also becomes abstract. All methods of interfaces are abstract. A **final** method is one which can not be overridden by child classes. We include the **final** construct to allow implementations of the SIDL bindings to perform optimizations by eliminating a lookup in a class's virtual function table. A **static** method is also **final**, with additional semantics. **Static** methods are invoked through a class, not an instance of a class. They are the closest thing to "global" methods in SIDL. We include the **static** construct to ease wrapping of non-object oriented language libraries and components.

Every class belongs to a nested package scope. Packages in SIDL are similar to namespaces in C++ and packages in Java. The package construct is used to create nested SIDL name space scopes. It is the only SIDL construct that creates a new name scope. Packages help prevent global naming collisions of classes and interfaces.

2.2 Scientific Data Types

Most IDL's, like those used in COM and CORBA, do not support all the types needed in scientific computing. For instance, both COM and CORBA's IDLs do not support complex numbers nor block style, dynamic multidimensional arrays. CORBA only supports static multidimensional arrays and sequences,

where COM only support ragged dynamic multidimensional arrays. In addition to the standard types like *int*, *char*, *bool*, *string*, and *double*, we have included *dcomplex*, *fcomplex*, and *array*. *dcomplex* is a complex number of type double. *fcomplex* is a complex number of type float. The *array* type has both a *type* specification and a *dimensions* specification. The *type* specification tells what type of elements the array contains and the *dimensions* specification tells how many dimensions are in the array. A SIDL *array* is the same as a Fortran block style array.

2.3 Exception Handling

Component architectures need robust mechanisms for error handling that can work across languages. For instance, COM requires all synchronous methods to return an error code and all asynchronous methods to return void. The mechanism for COM is not robust and requires a lot of run-time support to gain meaningful results, as with an exception mechanism found in Java. CORBA uses an exception mechanism where an environment variable is passed as the last argument in an argument list in a method and exceptions are set in that environment variable. We use a mechanism very similar to CORBA except that exceptions are not defined as structures, as they are in CORBA, but rather as objects, as they are in Java. All exceptions in SIDL are objects that inherit from *Throwable*. Also, we are exploring using a static environment variable, which would allow exceptions to be thrown without explicitly passing an environment variable as a method argument. Of course implementations of this model will have to be thread safe since they will be used in parallel applications.

2.4 Reflection

SIDL has constructs that allow support tools to implement reflection capabilities, which is necessary for components (e.g. CCA components). Recall that SIDL's object model is very similar to Java. SIDL also borrows it's introspection capabilities from Java. For instance, like Java, all SIDL objects implicitly inherit from *Object*. *Object* has a method *getClass* which returns a *Class* object. This *Class* object contains information about a particular object's methods, fields, and constructors, which can be queried and invoked at run-time. Every object has a *Class* object associated with it that contains information on it's methods, fields, and constructors. Given this, SIDL implementation tools, like Babel, can provide reflection capabilities by implementing SIDL's introspection specification.

3 Bindings and Implementation

This sections discusses the bindings of C and Fortran 77 to SIDL, as well as the implementation of those in the Babel run-time environment. This discussion is of only the more challenging aspects of developing the bindings and implementation. See [15] for a complete specification of SIDL and it's bindings to C and Fortran 77.

3.1 Bindings to C and Fortran 77

Mapping SIDL onto C and Fortran 77 posed some interesting challenges. For instance, mapping SIDL objects into C and Fortran 77 objects was not altogether obvious since neither language has object oriented features. Also, mapping complex numbers to C as well as mapping the SIDL array syntax to the two languages, posed some challenges as well. Besides these, the mappings of SIDL to C and Fortran 77 were fairly straight forward.

For C, SIDL objects are mapped to opaque structure pointers. In Fortran an object is mapped to an integer. Of course a run-time environment that implements these bindings will have to provide library support that can translate an integer representation of an object to the actual object, in order to get access to that object's data and methods (this is done by the Babel run-time environment). A method is invoked by passing the reference to the object, whether it be an integer in Fortran or an opaque structure pointer in C, as the first argument in the argument list.

Complex numbers in C are mapped to a structure with two elements. The first element is the real part of the number and the second part is the imaginary part. Arrays are mapped to structures in C that contain three elements. The first element is a single dimensional array that contains the lower bounds for each of the dimensions. The second element is the same as the first, except it contains the upper bounds. The third element is a pointer to the data. In Fortran, arrays are simply mapped to the corresponding array representation in Fortran. Bounds for the dimensions need to be specified explicitly in Fortran, since Fortran does not have structures.

3.2 Implementing the Babel Run-Time Environment

Most of the effort in developing the Babel compiler and run-time was in implementing the object model, namely: the virtual function tables, the object lookup table, reference counting, dynamic type casting, the exception handling mechanism, and reflection capabilities. The Babel run-time is implemented in C and the compiler is written in Java, however, the "glue" code that is generated from the compiler is in C. All of the object support is contained in the "glue" code and the run-time library. For instance, every object has a skeleton associated with it. The skeleton contains the implementation of the object, including the virtual function table (which is implemented like a static C++ virtual function table), constructors, destructors, support for dynamic type casting, etc... The run-time library contains support for reference counting, the object lookup mechanisms (which is necessary for supporting objects in Fortran), and the exception handling mechanism. The reflection capability is supported through both the skeleton and the run-time library.

One of the goals while developing Babel was to make function calls made through Babel fast. We were able to limit C to C function calls to one extra function call and one lookup. Calls between C and Fortran 77 required two extra function calls and one lookup, and Fortran 77 to Fortran 77 calls require

three extra function calls and one lookup. The extra function calls between languages are needed to translate between the different signatures. `Babel` does take advantage of the **static** and **final** constructs in `SIDL` by eliminating a function table lookup to functions of those types.

4 Results from Wrapping *hypre*

As a test case, we used Babel to create new interfaces for the *hypre* semicoursening mulitigrid solver (SMG) [4], a linear solver that is part of the *hypre* preconditioner library [5]. *hypre* is a library of parallel solvers for large, sparse linear systems being developed at Lawrence Livermore National Laboratory's Center for Applied Scientific Computing. The library currently consists of over 30,000 lines of C code, and it has 94 encapsulated user-interface functions. To test Babel we created a new interface (*hypre* is written in C, with a C interface provided by the authors) for both `C` and `Fortran 77` using `Babel`, and ran similar test drivers using the two `Babel` generated interfaces and the original `C` interface already provided by the library. We then compared the results from all three.

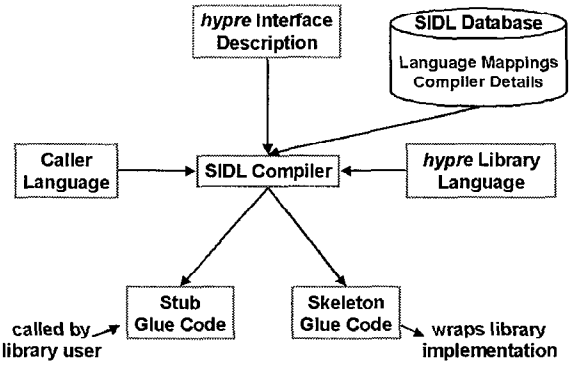

Fig. 1. Wrapping *hypre*

Wrapping *hypre* with `Babel` took three steps. The first step was to write a description of the existing interface in SIDL, which was done by two people, one who was familiar with SIDL and another who was familiar with the *hypre* library. The second step was to run the Babel compiler with the interface description as input to create all of the "glue-code" for each class (see **Fig.** 1). Since the signatures for the library functions were different from those in the virtual function tables, we also had to manually write the calls to the *hypre* functions into the library skeleton generated by the Babel compiler. This was a somewhat mundane task, but it required only one line of code per function, and it needed to be done only once, as the same skeleton code was used for both the `C` and `Fortran` (as well as for all other language bindings). Manually editing of

the skeleton code would not be necessary if the library used naming conventions and calling sequences that complied with the Babel specification (e.g., prepend every function call with an impl_). Once the function calls were manually added, the new C interface was complete, and then the Fortran interface was created almost instantly by running the compiler once more with different options to create the Fortran stub code. The final step was to compile and link the drivers with the skeletons, stubs, and the *hypre* library.

We rewrote an existing SMG test driver to test the efficacy of the new interfaces. The driver uses SMG to solve Laplace's equation on a 3-D rectangular domain with a 7-point stencil. First, all calls in the driver to the *hypre* library were replaced with the new C interfaces that Babel created. Then we wrote a new Fortran driver that sets up exactly the same problem using the same algorithm and calls the same *hypre* functions via the new Fortran interface. **Fig.** 2 shows a portion of the *hypre* interface written in `SIDL` and 3 shows portions of both the `C` and `Fortran` drivers that call the *hypre* library through `Babel`.

```
package hypre {
   class stencil {
      stencil NewStencil(in int dim, in int size);
      int SetStencilElement(in int index, inout array<int> offset);
   };
   class grid {
      grid NewGrid(in mpi_com com, in int dimension);
      int SetGridExtents(inout array<int> ilower, inout array<int> iupper);
   };
   class vector {
      vector NewVector(in mpi_com com, in grid g, in stencil s);
      int SetVectorBoxValues(inout array<int> ilower,
         inout array<int> iupper, inout array<double> values);
      ...
   };
   class matrix { /* matrix member functions omitted in this figure */ };
   class smg_solver {
      int Setup(inout matrix A, inout vector b, inout vector x);
      int Solve(inout matrix A, inout vector b, inout vector x);
      ...
   };
};
```

Fig. 2. SIDL for *hypre*.

Both new drivers ran with no change in numerical results. We compared the efficiency of the new `C` and `Fortran` drivers to the original C driver. The drivers that used the Babel wrappers solved large problems both sequentially and in parallel on 216 processors, with no noticeable effect (less than 1%) on the speed of execution. The overhead added by Babel is negligible when compared to the overhead of the numerical kernel of the library.

In all, this took less than an afternoon to wrap and run *hypre* on both Solaris and AIX using both a `C` and `Fortran` 77 driver. To put this in perspective, there was an effort by the *hypre* team to wrap *hypre* by hand, making it callable from

C Test Code

```
hypre_vector b, x;
hypre_matrix A;
hypre_smg_solver solver;
hypre_stencil s;

b = hypre_vector_NewVector(com, grid, s);
...
x = hypre_vector_NewVector(com, grid, s);
...
A = hypre_matrix_NewMatrix(com, grid, s);
...

solver = hypre_smg_solver_new();
hypre_smg_solver_SetMaxItr(solver, 10);
hypre_smg_solver_Solve(solver, &A, &b, &x);
hypre_smg_solver_Finalize(solver);
```

Fortran 77 Test Code

```
integer b, x
integer A
integer solver
integer s

b = hypre_vector_NewVector(com, grid, s)
...
x = hypre_vector_NewVector(com, grid, s)
...
A = hypre_matrix_NewMatrix(com, grid, s)
...

solver = hypre_smg_solver_new()
hypre_smg_solver_SetMaxItr(solver, 10)
hypre_smg_solver_Solve(solver, A, b, x)
hypre_smg_solver_Finalize(solver)
```

Fig. 3. Sample test code.

Fortran on a Solaris platform, that took over one week for one person to do. Even after they finished wrapping it, they had to redo the effort when they ported it to another platform.

5 Lessons Learned and Future Work

Our experience using Babel to create new interfaces for *hypre* shows that Babel is an effective tool to support language interoperability for high-performance, parallel scientific computing. While it is not difficult to use for an existing library such as *hypre*, Babel can be easier to use if a library, or component, is designed and written from the beginning with Babel naming conventions in mind. Calls to the library, or component, will also be faster, if these conventions are followed, since there will be one less function call. Calls through Babel can be streamlined even further by declaring methods **final** or **static**, where possible. This will eliminate a virtual function table lookup. Also, developers who use non-object oriented languages can take advantage of the object support that Babel provides.

In the future we will develop bindings for C++, Java, Fortran 90, and Python and implement those bindings in Babel. We will also explore various component composition and introspection models for scientific computing, in conjunction with the CCA, and develop the appropriate library implementations in Babel to support them.

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*, 1999.

2. S. Balay, B. Gropp, L. C. McInnes, and B. Smith, *A microkernel design for component-based numerical software systems*, in Proceedings of the First Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, 1998.
3. 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, Minneapolis, MN, March 1997.
4. P. Brown, R. Falgout, and J. Jones, *Semicoarsening multigrid on distributed memory machines*, in SIAM Journal on Scientific Computing special issue on the Fifth Copper Mountain Conference on Iterative Methods, 1999.
5. E. Chow, A. Cleary, and R. 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.
6. A. Cleary, S. Kohn, S. Smith, and B. Smolinski, *Language interoperability mechanisms for high-performance applications*, in Proccedings of the First Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, 1998.
7. G. Eddon and H. Eddon, *Inside Distributed COM*, Microsoft Press, Redmond, WA, 1998.
8. E. Eide, J. Lepreau, and J. L. Simister, *Flexible and optimized IDL compilation for distributed applications*, in Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers, 1998.
9. R. Englander, *Developing Java Beans*, O'Reilly, 1997.
10. Equations Solver Interface Forum. See `http://z.ca.sandia.gov/esi/`.
11. 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*, 1998.
12. J. Gosling and K. Arnold, *The Java Programming Language*, Addison-Wesley Publishing Company, Inc., Menlo Park, CA, 1996.
13. B. Janssen, M. Spreitzer, D. Larner, and C. Jacobi, *ILU Reference Manual*, Xerox Corporation, November 1997. Available at `ftp://ftp.parc.xerox.com/pub/ilu/ilu.html`.
14. JAVASOFT, *Java Native Interface Specification*, May 1997.
15. S. Kohn and B. Smolinski, *Component interoperability architecture: A proposal to the common component architecture forum. in preperation*, 1999.
16. MICROSOFT CORPORATION, *Component Object Model Specification (Version 0.9)*, October 1995. Available at `http://www.microsoft.com/oledev/olecom/title.html`.
17. OBJECT MANAGEMENT GROUP, *The Common Object Request Broker: Architecture and Specification*, February 1998. Available at `http://www.omg.org/corba`.
18. S. Parker, D. Beazley, and C. 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.
19. Paul Dubois, personal communication. See `http://xfiles.llnl.gov/CXX_Objects/cxx.htm`.
20. J. Shirley, W. Hu, and D. Magid, *Guide to Writing DCE Applications*, O'Reilly & Associates, Inc., Sebastopol, CA, 1994.

This article was processed using the LaTeX macro package with LLNCS style

Work performed under DOE at LLNL under contract No. W-7405-Eng-48.