Language Interoperability for High-Performance Parallel Scientific Components*

Brent Smolinski, Scott Kohn, Noah Elliott, and Nathan Dykman

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551

Abstract. Component technologies offer a promising approach for managing the increasing complexity and interdisciplinary nature of high-performance scientific applications. Language interoperability provides the flexibility required by component architectures. In this paper, we present an approach to language interoperability for high-performance parallel components. Based on Interface Definition Language (IDL) techniques, we have developed a Scientific IDL (`SIDL`) that focuses on the abstractions and performance requirements of the scientific domain. We have developed a `SIDL` compiler and the associated run-time support for reference counting, reflection, object management, and basic exception handling. The `SIDL` approach has been validated for a scientific linear solver library. Initial timing results indicate that the performance overhead is minimal (less than 1%), whereas the savings in development time for interoperable software libraries can be substantial.

1 Introduction

The scientific computing community is beginning to adopt component technologies and associated programming methodologies [1, 2, 10, 17] to manage the complexity of scientific code and facilitate code sharing and reuse. Components require language interoperability to isolate component implementation details from applications. This ensures that applications and components can be created and evolve separately. With the proliferation of languages used for numerical simulation—such as `C`, `C++`, `Fortran 90`, `Fortran 77`, `Java`, and `Python`—the lack of seamless language interoperability negatively impacts the reusability of scientific codes.

Providing interoperability among the many languages used in scientific computing is a difficult problem for both component and library developers. Without language interoperability, application developers must use only the same language as the components, even though better languages may exist. If language

* Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48. This work has been funded by LDRD grant 99-ERD-078. Available as LLNL techinical report UCRL-JC-134260.

interoperability is desired, component developers and users are often 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. Both approaches couple the components and applications too tightly, restricting component reuse and flexibility.

1.1 Language Interoperability Design Considerations

The design considerations associated with language interoperability for high-performance scientific computing differ from those of the business sector, which is supported by industry efforts such as COM [7, 15] and CORBA [16]. The Common Component Architecture (CCA) [1], Equation Solver Interface [9] and other scientific computing working groups require support for complex numbers, `Fortran`-style dynamic multidimensional arrays, object-oriented semantics with multiple inheritance and method overriding, and very efficient function invocation for components living in the same address space. The CCA consortium is developing component technologies appropriate for high-performance parallel scientific computing. The ESI is developing standards for linear solvers and associated preconditioners based on component approaches to increase the interoperability of numerical software developed by different development teams.

1.2 Related Interoperability Approaches

Several language interoperability packages have been developed that automatically generate 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 the mediating code that allows these routines to be called from scripting languages such as `Python`. Such approaches typically introduce an asymmetric relationship between the scripting language and the compiled language. Calls from the scripting language to the compiled language are straight-forward, but calls from the compiled language to the scripting language are difficult or are not supported.

Foreign invocation libraries have been used to manage interoperability among targeted languages. For instance, the *Java Native Interface* [13] defines a set of library routines that enables `Java` code to interoperate with libraries written in `C` and `C++`.

Such interoperability approaches support language interoperability among only a limited set of languages, and they do not support a single, universal mechanism that works with all languages. In the worst case, interoperability among N languages could require $O(N^2)$ different approaches. Component architectures require a more general approach, which we describe in the following section.

1.3 Interoperability Through an IDL Approach

One interoperability mechanism used successfully by the distributed systems and components community [12, 15, 16, 18] 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. The advantage of an IDL approach over other approaches is that it provides a single, uniform mechanism for interoperability among a variety of languages.

Current IDL implementations are not sufficient for specifying interfaces to high-performance scientific components. First, standard IDLs such as those defined by CORBA and COM are targeted towards business objects and do not include basic scientific computing data types such as complex numbers or dynamic multidimensional arrays. Second, approaches focused on distributed objects do not generally provide support for high-performance, same address space function calls between different languages. Our performance goal is to reduce the overhead of single address space function calls to about that of a `C++` virtual function invocation. Third, many IDLs do not support multiple inheritance or have a limited object model. For example, COM does not support multiple inheritance and supports implementation inheritance only through composition or aggregation, which can be computationally expensive and difficult to implement. CORBA does not support method overriding, which is required for polymorphism.

We have adopted an IDL approach for handling language interoperability in a scientific computing environment. We have developed a Scientific IDL called `SIDL` [6, 14] as well as a run-time environment that implements bindings to `SIDL` and provides the library support necessary for a scientific component architecture. Currently `SIDL` supports bindings to `C` and `Fortran` 77, although others are under development. Preliminary experiments with a scientific solver library have shown that `SIDL` is expressive enough for scientific computing and that language interoperability is possible with little measurable run-time overheads.

1.4 Paper Organization

This paper is organized as follows. Section 2 introduces `SIDL` features that are necessary for high-performance parallel computing. Section 3 describes the bindings of `SIDL` to `C` and `Fortran` 77, as well as the run-time environment, which includes a `SIDL` compiler and library support. Section 4 details the process of applying the `SIDL` interoperability approach to a scientific software library and provides parallel performance results for both `C` and `Fortran`. Finally, we conclude in Section 5 with an analysis of the lessons learned and the identification of future research issues.

2 Scientific Interface Definition Language

A scientific IDL must be sufficiently expressive to represent the abstractions and data types common in scientific computing, such as dynamic multidimensional arrays and complex numbers. Polymorphism—required by some advanced numerical libraries [9]—requires an IDL with an object model that supports multiple inheritance and method overriding. The IDL should also provide robust and efficient cross-language error handling mechanisms.

Unfortunately, no current IDLs support all these capabilities. Most IDLs have been designed for operating systems [7, 8] or for distributed client-server computing in the business domain [12, 16, 18] and not for scientific computing.

The design of our Scientific IDL borrows many ideas from the CORBA IDL [16] and the `Java` programming language [11]. `SIDL` supports an object model similar to `Java` with separate interfaces and classes, scientific data types such as multidimensional arrays, and an error handling mechanism similar to `Java` and CORBA. `SIDL` provides reflection capabilities that are similar to `Java`.

The following sections describe `SIDL` in more detail. An example of `SIDL` for a scientific preconditioning solver library is given in Figure 3 of Section 4.

2.1 Scientific Data Types

In addition to standard data types such as *int*, *char*, *bool*, *string*, and *double*, `SIDL` supports *dcomplex*, *fcomplex*, and *array*. An *fcomplex* is a complex number of type float, and a *dcomplex* is a complex number of type double. A `SIDL` *array* is a multidimensional array contiguous in memory, similar to the `Fortran`-style arrays commonly used in scientific computing. The *array* type has both a type, such as *int* or *double*, and a dimension, currently between one through four, inclusive. In comparison, CORBA supports only statically-sized multidimensional arrays and single-dimension sequences, and COM supports only pointer-based, ragged multidimensional arrays.

2.2 SIDL Object Model

The `SIDL` object model is similar to that of the `Java` programming language. We chose the `Java` object model for `SIDL` because it provides a simple model for multiple inheritance. `SIDL` supports both interfaces and classes. A `SIDL` class may inherit multiple interfaces but only one class implementation. This approach solves the ambiguity problems associated with multiple implementation inheritance in languages such as `C++`.

SIDL provides a new set of interface method declarations. These declarations provide optimization opportunities and increase the expressiveness of the IDL. Like `Java`, class methods may be declared **abstract**, **final**, or **static**. An **abstract** method is purely declarative and provides no implementation; an implementation must be provided by a child class. A **final** method is one that cannot be overridden by child classes. The **final** construct enables optimizations in the run-time system that eliminate potential dereferences to an overriding method. As in C++ or Java, **static** methods are associated with a class, not a class instance, and therefore may be invoked without an object. The **static** construct simplifies developing SIDL interfaces to legacy libraries that were written without object-oriented semantics.

2.3 Scoping and Exception Handling

Every class and interface belongs to a particular package scope. Packages in SIDL are similar to namespaces in C++ and packages in Java. The package construct is used to create nested SIDL namespaces. Packages help prevent global naming collisions of classes and interfaces that are developed by different code teams.

Component architectures require robust error handling mechanisms that operate across language barriers. We have designed an error reporting mechanism similar to Java. All exceptions in SIDL are objects that inherit from a particular library interface called *Throwable*. Error objects support more complex error reporting than what is possible with simple integer error return codes. Error conditions are indicated through an environment variable that is similar to CORBA.

2.4 Reflection

Reflection is the mechanism through which a description of object methods and method arguments can be determined at run-time. Reflection is an critical capability for component architectures, as it allows applications to discover, query, and execute methods at run-time. This allows applications to create and use components based on run-time information, and to view interface information for dynamically loaded components that is often unavailable at compile-time.

The SIDL run-time library will support a reflection mechanism that is based on the design of the Java library classes in `java.lang` and `java.lang.reflect`. The SIDL compiler automatically generates reflection information for every interface and class based on its IDL description. The run-time library will support queries on classes and interfaces that allow methods to be discovered and invoked at run-time.

3 Bindings and Implementation

SIDL defines component interfaces in a language-independent manner. For each programming language, we must define language mappings that map constructs in SIDL onto that target language. In this section, we describe the mappings of SIDL to C and Fortran 77, as well as the required library support for the run-time environment. We discuss only the more challenging aspects of the mappings and implementation; a complete specification can be found elsewhere [14].

3.1 Mappings to C and Fortran 77

Because SIDL is based on CORBA IDL, we were able to use the CORBA specification [16] as a guide in mapping many of the SIDL constructs into C. Fortran 77 mappings closely followed the C mappings, whith exceptions as described below. The mappings for complex numbers and multidimensional arrays to C and Fortran 77, which are not part of the CORBA IDL, where relatively straightforward.

Mapping SIDL classes and interfaces in C and Fortran 77 presented some interesting challenges, since neither language supports object-oriented features. However, the IDL approach allows object-oriented concepts to be mapped onto non-object-oriented languages. For C, SIDL classes and interfaces are mapped to opaque structure pointers that encapsulate private data members, method invocation tables, and other implementation details. For Fortran 77, classes and interfaces are mapped to integers that are used as handles. The run-time environment manages object information and automatically translates between the Fortran integer representation and the actual object reference. Methods on SIDL objects are invoked using a standard C or Fortran 77 function call with the object reference as the first parameter. Figure 3 of Section 4 illustrates these conventions for a scientific linear solver library.

3.2 Implementing the SIDL Run-Time Environment

Much of the effort in developing the SIDL compiler and run-time system was in implementing the object model, namely: virtual function tables, object lookup table for mapping to and from Fortran integer handles, reference counting, dynamic type casting, exception handling mechanism, and reflection capabilities. The run-time library support is implemented in C and the compiler is written in Java. The "glue" code generated from the compiler is in C.

All object support is distributed between the glue code and the run-time library. The glue code contains the implementation of the object mapping, including the virtual function lookup table (similar to a C++ virtual function table), constructors, destructors, and support for dynamic type casting. The run-time library contains support for reference counting, object lookup mechanisms necessary for Fortran objects, and exception handling mechanisms. The reflection capability is supported through both the glue code and the run-time library.

One of the goals of the SIDL run-time environment is to provide extremely fast function calls between components living in the same memory space. For C to C calls, our current implementation requires one table look-up (to support virtual functions) and one additional function call. Calls between C and another language add the overhead of an additional function call, and calls between two non-C languages requires yet another call. These additional function calls are needed to isolate language-specific linker names. Where possible, the SIDL compiler takes advantage of the **static** and **final** qualifiers in SIDL by eliminating a function table lookup to functions for those types.

4 Applying SIDL to a Scientific Library

As a test case, we used the SIDL tools to create new interfaces for a semicoursening mulitigrid (SMG) solver [4], a preconditioner that is part of the *hypre* linear solver 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 our approach, we created a SIDL interface and and created both C and Fortran 77library wrappers with SIDL. We ran similar test drivers for the two SIDL generated wrappers and the original C interface already provided by the library, and compared the results from all three runs.

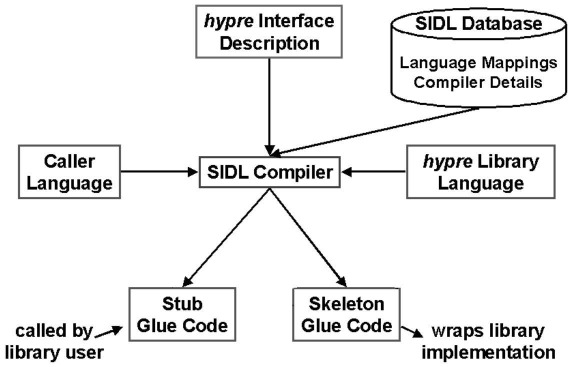

Fig. 1. Generating "glue" code for the *hypre* library using the SIDL tools.

Wrapping *hypre* using SIDL proceeded in three steps. First, the existing *hypre* interface was written in SIDL 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 SIDL compiler with the interface description as input to automatically generate the glue code for each class (see Figure 1). Since the names created by SIDL compiler are slightly different from those expected by the rest of the original *hypre* library, the library had to be slightly modified to match the new names of the SMG interfaces. This step is not required if SIDL conventions are used and only has to be done once. Once the function calls were manually added for the C language bindings, the Fortran interface was created automatically by running the compiler once more with options for Fortran. 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 performance 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 existing C driver to the *hypre* library were replaced with the new C interfaces created by SIDL. Then we wrote a new Fortran driver for the same problem that calls the same *hypre* functions

```
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> lower, inout array<int> upper);
   };
   class vector {
      vector NewVector(in mpi_com com, in grid g, in stencil s);
      int SetVectorBoxValues(inout array<int> lower,
         inout array<int> upper, 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. Portions of the *hypre* interface specification written in SIDL.

via the new Fortran interface. Figure 2 shows a portion of the *hypre* interface written in SIDL, and Figure 3 shows portions of both the C and Fortran drivers that call the *hypre* library using the automatically generated interfaces.

Both test drivers produced the same numerical results. We compared the efficiency of the new C and Fortran drivers to the original C driver. The drivers that used SIDL 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 SIDL is negligible when compared to the overhead of the numerical kernels in the library.

This entire process required less than an afternoon to generate the SIDL interface, edit the skeleton code, and generate C and Fortran stub code. To put this in perspective, there was an effort by the *hypre* team to manually generate a Fortran interface for *hypre* that required over one person-week of effort. This work was targeted at the Solaris platform. Porting this hand-generated Fortran interface to another platform required a substantial re-write of the interface due to differences in Fortran name representation. Such platform dependencies are managed automatically by the SIDL tools.

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 calling *hypre* interfaces for `C` and `Fortran` 77 generated automatically using the `SIDL` tools.

5 Lessons Learned and Future Work

We have presented `SIDL`, a scientific interface definition language, and a run-time that meets the requirements requirements for scientific computing. `SIDL` borrows heavily from the CORBA IDL and `Java` programming language, while adding features necessary for scientific computing. `SIDL` seems to capture the abstractions necessary for scientific computing, as well as new features that a run-time can use to perform optimizations, which are not present in current IDL standards.

The `SIDL` run-time also provides fast same address space calls, which is important for effective scientific computation. A comparison using the *hypre* library showed that `SIDL` added only one to two percent overhead compared to the native interfaces. This is neglible when compared to the great savings in developer costs and flexibility. The `SIDL` run-time allowed the creation of a `Fortran` 77 interface in the *hypre* library in a fifth of the time required to create a similar interface by hand.

In the future we will develop bindings for `C++`, `Java`, `Fortran` 90, and `Python` and implement those bindings. `Fortran` 90 is challenging since `Fortran` 90 calling conventions vary widely from compiler to compiler. We will also continue our collaboration efforts with the CCA and ESI working groups. Other ESI specifications will require more expressability from `SIDL` than the *hypre* interface requires. Features may also need to be added to `SIDL` to support the specification of high-performance scientific components (e.g. CCA compliant components).

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 Proceedings 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. Equations Solver Interface Forum. See `http://z.ca.sandia.gov/esi/`.
10. 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).
11. J. Gosling and K. Arnold, *The Java Programming Language*, Addison-Wesley Publishing Company, Inc., Menlo Park, CA, 1996.
12. B. Janssen, M. Spreitzer, D. Larner, and C. Jacobi, *ILU Reference Manual*, Xerox Corporation, November 1997. See `ftp://ftp.parc.xerox.com/pub/ilu/ilu.html`.
13. JAVASOFT, *Java Native Interface Specification*, May 1997.
14. S. Kohn and B. Smolinski, *Component interoperability architecture: A proposal to the common component architecture forum. in preperation*, 1999.
15. MICROSOFT CORPORATION, *Component Object Model Specification (Version 0.9)*, October 1995. See `http://www.microsoft.com/oledev/olecom/title.html`.
16. OBJECT MANAGEMENT GROUP, *The Common Object Request Broker: Architecture and Specification*, February 1998. Available at `http://www.omg.org/corba`.
17. 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.
18. J. Shirley, W. Hu, and D. Magid, *Guide to Writing DCE Applications*, O'Reilly & Associates, Inc., Sebastopol, CA, 1994.