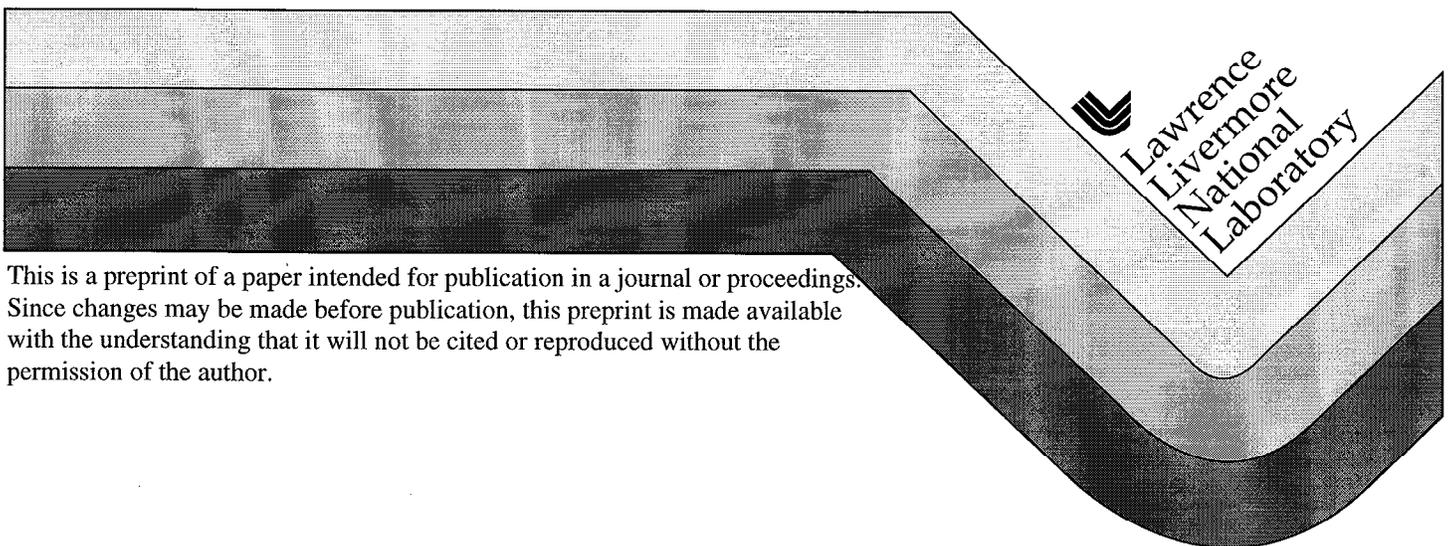


## Language Interoperability Mechanisms For High-Performance Scientific Applications

A. Cleary, S. Kohn, S. G. Smith, B. Smolinski

This paper was prepared for submittal to  
Proceedings of the Society for Industrial & Applied  
Mathematics Workshop on Object- Oriented Methods for  
Interoperable Scientific and Engineering Computing  
Yorktown Heights, NY  
October 21-23, 1998

September 18, 1998



This is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint is made available with the understanding that it will not be cited or reproduced without the permission of the author.

#### DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

# Chapter 1

## Language Interoperability Mechanisms for High-Performance Scientific Applications\*

Andrew Cleary<sup>†</sup>   Scott Kohn<sup>†</sup>   Steven G. Smith<sup>†</sup>   Brent Smolinski<sup>†</sup>

### Abstract

Language interoperability is a difficult problem facing the developers and users of large numerical software packages. Language choices often hamper the reuse and sharing of numerical libraries, especially in a scientific computing environment that uses a breadth of programming languages, including C, C++, Java, various Fortran dialects, and scripting languages such as Python. In this paper, we propose a new approach to language interoperability for high-performance scientific applications based on Interface Definition Language (IDL) techniques. We investigate the modifications necessary to adopt traditional IDL approaches for use by the scientific community, including IDL extensions for numerical computing and issues involved in mapping IDLs to Fortran 77 and Fortran 90.

### 1 Introduction

In recent years, the scientific computing community has seen a proliferation of languages used for numerical simulation. The traditional Fortran mainstay, Fortran 77, has been joined by Fortran 90. C and C++ have become popular because of their support for dynamic memory allocation, data structures, and—in the case of C++—object oriented abstractions. The popularity of Java has driven standards proposals for Java numerical libraries [8]. Computational scientists have also experimented with the use of high-level scripting languages such as Python to coordinate large numerical simulations [4].

Language interoperability in this multilanguage environment is a difficult problem for developers of new large numerical software packages and also for users of legacy software. For library developers, the choice of one implementation language over another may severely limit the reuse of their numerical software, especially considering the breadth of programming languages used in the scientific computing environment. Users of legacy software may be required to adopt the language of the legacy package for future applications development, even though better alternatives may exist. If language interoperability is desired, numerical software 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. Fortran 90 is a particular challenge for language interoperability, since Fortran 90 calling conventions vary widely from compiler to compiler (see Section 3.3 for details).

---

\*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 both ASCI PSE and DOE2000.

<sup>†</sup>Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA.

One interoperability mechanism used successfully by the distributed systems community [7, 12, 14] and the operating systems community [5, 6, 11] 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 the software package from other programming languages.

In this position paper, we propose to explore the IDL approach to language interoperability and modify it for use by the scientific community. We begin with the object oriented CORBA IDL specification [12] as a starting point and investigate the modifications necessary for high-performance scientific computing.

Although IDLs are a proven technology for other communities, IDL techniques have not been applied to high-performance scientific computing. We anticipate three primary research issues in adopting IDL technology. First, we must determine what features are needed in a scientific IDL to support numerical computing. For example, standard IDLs such as that defined by CORBA do not include basic scientific computing data types such as complex numbers or dynamic multidimensional arrays. Second, we must address performance considerations. Our goal is to make the overhead of calls through the IDL about as expensive as the invocation of a C++ virtual function. Finally, we must determine how IDL features such as objects and their methods are to be mapped onto the various Fortran dialects.

This paper is organized as follows. We begin with a survey of common interoperability mechanisms. Section 3 describes our design goals for language interoperability, features needed in a scientific IDL, language mappings, and run-time issues. Finally, we conclude in Section 4 with an evaluation of our proposed IDL interoperability approach.

## 2 Survey of Interoperability Mechanisms

Language interoperability—the problem of connecting software modules written in different programming languages—is a somewhat imprecise term. Programming languages provide differing degrees of support for data abstraction, object oriented design, dynamic memory allocation, or array-based computation. Such differences limit the level to which language interoperability can be supported. For example, there are limitations to interoperability between C and Fortran 77 since Fortran 77 does not support C’s notion of a pointer.

Of course, one of the reasons that language interoperability is so desirable is that it enables programmers to exploit the various strengths of different programming languages. No language has been shown to be the single best language for scientific computing. Fortran is an excellent language for efficient array computation, but does not provide the data abstraction and object oriented features of a language such as C++ or Java. Scripting languages such as Python provide a powerful environment for experimenting with scientific simulations [4], but do not offer the performance of a compiled language such as C.

### 2.1 Multiple Language Bindings

Probably the most common method of language interoperability for scientific libraries is through the use of hand-generated library bindings. In this approach, library designers select a (typically small) set of supported languages that will be able to call their library. For each of these supported languages, the designers write a language binding specification that describes the library interface—the objects (if any), functions, and data types—in

that particular language. Essentially, library interfaces are redesigned for every supported language. Finally, the library developers implement this language binding specification, typically using glue code that connects the target language to the language used in the implementation of the library. MPI [10] is one scientific library that takes the multiple language binding approach; the MPI specification describes bindings for both C and Fortran 77.

The advantage to generating language bindings by hand is that the binding can be tailored to the style and conventions of the particular language. For example, the MPI specification dictates that MPI routines in C return error codes as function return values whereas Fortran routines return error codes through an integer parameter in the argument list, which follows the standard programming conventions for these two languages.

The primary disadvantage of this approach is that it is very labor-intensive. Both the language binding and the glue code must be generated by hand for each supported language. Although the generation of glue code is typically straight-forward, many lines of glue code will be needed to wrap every object, function, and data type that is to be accessible by the calling language.

Furthermore, the use of multiple language bindings does not necessarily ensure simultaneous cross-language use of the library. For example, the MPI language bindings for both Fortran and C contain routines that create MPI communicator objects. However, there is no well-defined mechanism for sharing MPI communicator objects between Fortran and C. Therefore, a C application that allocates an MPI communicator cannot pass that communicator to a Fortran numerical library routine. A careful design of the language bindings can address these issues, but the difficulty grows with the size of the library and the number of supported languages.

## 2.2 Bilateral Language Interoperability

Another technique for language interoperability uses libraries or automatically generated 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 may be called by scripting languages such as Python. Pyffle [13] is similar in approach to SWIG and supports an almost seamless integration between Python and C++. The *Java Native Interface* [9] defines a set of library routines that enables Java code to interoperate with applications and libraries written in C and C++. Python supports a calling interface for C. These approaches solve part of the interoperability problem by developing custom solutions that link particular languages; however, they do not address the larger issues involved with interoperability for all of the scientific computing languages. Indeed,  $N$  languages could potentially require  $O(N^2)$  different software packages for full interoperability.

## 2.3 Interface Definition Languages (IDLs)

The IDL approach to interoperability is somewhat similar to the approach described in the previous section except that it unifies all languages through a common mechanism. The basic idea behind an IDL is to create a new language—the Interface Definition Language—to provide a description of the interface for a software routine. An IDL language mapping is defined for each supported language that maps constructs in the IDL into the target language. For example, an interface in an object oriented IDL might be mapped onto a class in C++ or an interface in Java. Glue code is generated automatically by an IDL compiler that takes as input the IDL description of a software component and a language

mapping for the target language. The IDL approach reduces the  $O(N^2)$  potential language mappings to only  $O(N)$ , since every language gets mapped to the IDL, from which every other language is accessible.

IDL's have been in use for a long time in the distributed computing field. Sun RPC IDL and OSF/DCE IDL have been standard mechanisms for specifying remote procedure call interfaces and have been widely used for both UNIX and Windows NT client/server programming [RPC REF, 11]. Microsoft borrowed heavily from the OSF/DCE IDL for its COM IDL specification. Microsoft's Component Object Model (COM) is the most widely used component framework. CORBA ... The ILU (Inter-Language Unification) project [7] is apply

The drawback to the IDL approach is that language interoperability is limited to the facilities and types described in the IDL, which may be a subset of the capabilities in the native languages. For example, a C pointer cannot be described in an IDL interface for an IDL that doesn't support the pointer type. However, this does not necessarily mean that IDLs represent the lowest common denominator for all the languages of interest. Many IDL capabilities, such as object oriented constructs, can be supported through a combination of clever language mappings and run-time library routines, even for simple languages such as C or Fortran 77.

### 3 IDLs for Scientific Applications

Of the three interoperability approaches described in the previous Section, we believe that IDL techniques offer the most potential for the automated, seamless interoperability of scientific libraries. In this section, we describe the modifications necessary to adapt existing IDL methods for the scientific computing environment. We begin with a description of how our approach would be viewed from the perspective of both a library developer and a library user.

The developer of a numerical software library would perform the following steps.

1. Specify an interface to the library in the IDL. The IDL specification provides a high-level, language-independent description of the library interface. For example, the following is an IDL specification for a Vector object in a hypothetical Solver library.

```
package Solver {
  interface Vector {
    // data access to the vector
    void setData(in double data);
    void setData(in array<double,1> data);
    void getData(out array<double,1> data);

    // standard vector functions
    double dot(in Vector y);
    void scale(in double a);
    void axpy(in double a, in Vector y);
    ...
  };
};
```

2. Compile the IDL specification using the IDL compiler to generate skeleton glue code in the implementation language of the library.

3. Write the functions that implement the interface. In doing so, the library developer must ensure that the function signatures match those expected by the skeleton glue code. For example, the CORBA language mappings for C specify that the `dot()` member function given above would be implemented by C function `Solver_Vector_dot()`. For implementors of a new library, these naming conventions are not particularly difficult to follow. However, library developers that wish to wrap existing libraries in the IDL for interoperability may need to write a small amount of glue code to convert between the expected IDL function names and the names used by the library. Note that this glue code need be written only once to map the IDL to the library, as opposed to writing glue code for every language as in the approach described in Section 2.1.
4. Deliver the library code along with the skeleton glue code generated by the IDL compiler.

To create an application that uses the library described above, a library user would:

1. Compile the IDL specification provided by the library developer for the application target language. The IDL compiler will generate stub glue code that will connect the applications code to the library.
2. Write the applications code. The library user will reference the library interface as specified language mappings and the stub glue code. For example, using CORBA conventions, the IDL function `dot()` given above would be mapped to method `Solver::Vector::dot()` for C++ and to function `Vector_dot()` in module `Solver` for Fortran 90.
3. Compile and link the applications code with the stub and skeleton code generated by the IDL compiler, the library code, and the small run-time library needed by the IDL system.

### 3.1 Design Considerations

Our IDL approach must not introduce significant overheads at run-time; otherwise, it will not be used in a high-performance computing environment. Traditionally, IDLs have been used for distributed applications spanning multiple address spaces. Because there can be no data sharing across multiple address spaces, distributed run-time systems must marshal, communicate, and un-marshal data arguments during method invocation. Such overheads would be prohibitively expensive for the large scientific data sets found in high-performance computing. Therefore, we require that all software modules linked by our IDL must share the same address space. Within a single, shared address space, data can be passed between modules via reference without expensive data copies.

Note that this design constraint does not preclude the use of our approach for high-performance parallel computation using MPI. The traditional SPMD approach to parallelism already assumes a single address space for each MPI process, and our design fits naturally into this programming model. Indeed, the interoperability needs of numerical libraries for massively parallel computation is the driver for much of this work.

With the increasing use of shared memory multiprocessors in scientific computing, we plan on supporting threads. The runtime system will be designed to be thread safe. Library writers will be responsible for thread safety of components. An open issue is which thread model to use; both OpenMP and Pthreads will likely be in widespread use. At the current

time we do not know how to implement the runtime system to support the simultaneous use of both thread models so we will initially target Pthreads. We do not address the issues regarding the interaction of MPI and threads.

Finally, our design must support the standard numerical programming languages, including C, C++, Fortran 77, Fortran 90, Java, and Python. Additionally, we plan to investigate support for mathematical prototyping tools such as MatLab.

## 3.2 Scientific IDL

For this approach to work, we must choose an IDL that is expressive enough to represent the abstractions and data types common in scientific computing. Unfortunately, no such IDL currently exists, since most IDLs have been designed for distributed client-server computing in the business domain.

Thus, we have decided to begin with the object oriented CORBA IDL specification [12] as a starting point and then modify it as necessary for high-performance scientific computing. The CORBA IDL was chosen for several reasons. It is fairly simple and elegant with a syntax similar to Java or simplified C++, object oriented, and supports an error-reporting exception mechanism. It provides a module construct that helps manage the namespaces for different libraries (e.g., to ensure that the `Vector` object from library A does not clash with the `Vector` object from library B). It is an industry standard and supported by a large user community. With the exception of Fortran and MatLab, language mapping specifications have been written for all of our targeted scientific languages. In our work, we can leverage these language mappings and the other work in the CORBA community.

In the following sections, we describe some of the issues in adopting the CORBA IDL for scientific computing.

**3.2.1 Unnecessary CORBA IDL Constructs** The CORBA IDL contains a number of constructs that are either inappropriate or unnecessary for scientific computing. For example, the `oneway` method attribute only makes sense in a distributed environment. To simplify the development of our prototype, we have also eliminated support for `struct` and `union`. Both of these constructs can be represented easily using objects, as is done in Java. These constructs may be included later, if warranted.

**3.2.2 New Types for Scientific Computing** The CORBA IDL specification lacks both complex numbers and dynamic multidimensional arrays, and both are essential to numerical and scientific computing. Complex numbers are fairly trivial to add to the IDL. The only issue is the mapping of the complex numbers into languages without a built-in complex type, but complex number libraries either exist or are straight-forward to implement for all languages of interest.

We have also added dynamic multidimensional arrays to the CORBA IDL. CORBA currently only supports fixed-length arrays and sequences. A sequence is similar to an array but is limited to one dimension. In CORBA IDL, varying-length multidimensional arrays are generally built from sequences of sequences. However, this representation is similar to an C array of pointers to arrays and is not as natural for most scientific computing as multidimensional arrays. As illustrated in the IDL sample code in the beginning of this Section, we have specified arrays as `array<TYPE,N>`, where `TYPE` is the type of the array (e.g., `double`) and `N` is the array dimension.

Another issue is the representation of arrays in the various targeted programming languages. For efficiency, IDL arrays should map onto native array constructs. However,

the native representation of arrays in Fortran (column major) is different from C and C++ (row major) and also Java and Python, which have their own representations. There are three potential solutions we plan to evaluate. The first is to automatically convert the array to the representation assumed by the implementation language. Thus, arrays passed to a Fortran library from a C application would need to be transposed in memory. This is the simplest solution, but also the most expensive, since arrays would need to be copied on every call between languages with different native representations. Second, layout attributes such as `column` or `row` could be added to the IDL to specify the format of the array expected by the implementation. This would provide more flexibility for the library developer and would force data copies only when needed. Finally, the IDL run-time system could provide simple routines that would convert array representations at run-time at the request of the library implementation.

**3.2.3 New IDL Constructs** We have added two new method modifiers to the CORBA IDL: `static` and `final`. Static methods may be invoked without an explicit object reference and are supported by both Java and C++; they can be thought of as a standard function call in a non-object oriented language. Static methods are not supported by CORBA since distributed computing environments require object references to specify the execution context. Static methods will be essential in the generation of IDL descriptions for legacy subroutine libraries that were written without an object model.

The `final` qualifier is taken from Java and indicates that the specified method may not be redefined in subclasses. By default, we adopt the Java convention that all non-static methods may be redefined in subclasses unless they are declared `final`. This is the opposite of the C++ convention, which assumes that methods must be explicitly declared `virtual` to be redefined by subclasses. There is a slight overhead cost associated with dynamic function dispatch for virtual (i.e., non-final) methods. The `final` keyword will enable the stub code and run-time system to optimize away these overhead costs.

**3.2.4 Inheritance Issues** Unfortunately, the CORBA specification does not currently support method redefinition in subclasses nor a useful model of multiple inheritance. We consider both of these necessary capabilities for the object oriented design of general and extensible scientific libraries. It is straight-forward to support method redefinition in the run-time system (see Section 3.4); however, multiple inheritance is more problematic.

There are two potential models for multiple inheritance, which we shall call the C++ model and the Java model. The C++ approach allows a subclass to inherit both interface and implementation from multiple superclasses. Unfortunately, multiple inheritance of implementations causes difficulties when superclass methods share the same signature; references to such methods are generally ambiguous, since the compiler does not know which method implementation to invoke. C++ solves this problem by requiring unambiguous references in the implementation that is enforced by the compiler. Such an approach does not work with an IDL, since the IDL cannot force the compilers used for the library implementations to check the semantics for multiple inheritance.

Thus, we have chosen to implement Java's model for multiple inheritance. In this model, a subclass may inherit multiple interfaces but only one implementation. This appears to be a much more elegant model for multiple inheritance and it does not share the limitations of C++'s model. Following the Java model, we have also added an `abstract` qualifier that indicates that a method does not have an implementation and must be defined by a subclass.

### 3.3 IDL Language Mappings

An IDL language mapping determines how IDL features are mapped onto the target language. Language mappings for the CORBA IDL have been defined by the CORBA specification for C, C++ and Java[12]. The ILU project [7] has defined a mapping between the CORBA IDL and Python. Obviously, these mappings must be extended for the features that we have added to the base CORBA IDL, but the mappings for these additional extensions are fairly straight-forward.

Unfortunately, IDL language mappings to Fortran dialects do not exist. For the most part, the mapping between IDL and Fortran will be similar to the mapping between IDL and C with the exception of the representation for objects, strings, and arrays. Objects in Fortran 77 are generally represented using integer identifiers in the same fashion that MPI [10] uses integers to represent MPI communicators; objects in Fortran 90 can either be represented using the same approach or the expanded Fortran 90 type system. Strings that are `char *` arrays in C++ will become `character *(*)` arrays in Fortran with different termination conditions (NULL for C but an explicit length for Fortran). Arrays mappings for Fortran 77 are straight-forward, although Fortran 90 mappings are problematic, as described below.

The primary problem in mapping to Fortran is that calling sequences, name mangling, and Fortran 90 array descriptors vary greatly from compiler to compiler. Thus, in order to generate glue code for Fortran, the IDL compiler system must be aware of the low-level details of Fortran compiler conventions. For example, consider the following potential Fortran 90 mapping of the sample IDL code given in the beginning of this Section.

```

module Solvers
...
contains
  subroutine Vector_setData(this, data)
    type (Vector) this
    real *8, dimension (:) :: data
    ...
  end subroutine Vector_setData
end module Solvers

```

Calling this Fortran 90 code from another language requires that we understand how the Fortran 90 compiler represents the function name and how data is passed into the function. Figure ?? illustrates the differences in naming and parameter passing conventions for two f90 compilers. There are two important points. First, the compilers generate different symbols for the function `Vector_setData()`. Second, the compilers use different array descriptor structures to represent the array argument, including different definitions of bounds and stride (e.g., byte-based for the SGI but word-based for the Sun).

Fortunately, we need to determine these Fortran 90 calling conventions only once and then catalogue them within the IDL compiler system. Once the conventions are established, the IDL compiler will automatically generate the glue code necessary to tie the Fortran 90 code with other languages. Note that it would be exceedingly tedious to generate this glue code by hand considering the significant differences in Fortran 90 calling mechanisms.

### 3.4 Run-Time Support Library

Difficult to support such things by hand. Talk about PETSc. Get a reference to a description of what PETSc does or COM tables or C++ virtual function tables. Support for

```

struct vector { ... };
struct f90_array {
    double *data;
    ...
    short flags;
    short rank;
    ...
    ...
    int lower0;
    int upper0;
    int stride0;
};
void vector_setdata.in.solvers_(
    struct vector *this
    struct f90_array *data)
{
    ...
}

struct vector { ... };
struct f90_array {
    ...
    double *data;
    ...
    short flags;
    short rank;
    ...
    ...
    int lower0;
    int upper0;
    int stride0;
};
void solvers$vector_setdata_(
    struct vector *this
    struct f90_array *data)
{
    ...
}

```

FIG. 1. This figure gives a pseudo-C representation that is equivalent to the assembly code generated by the f90 compilers on the Sun (v1.2, left) and the SGI (v6.2, right). Note that this is not valid C code since neither "." nor "\$" are valid characters within C identifiers.

multiple inheritance.

As mentioned previously the runtime system must be thread aware due to the possibility of multiple threads manipulating components. There should be no significant implementation difficulties, as the design is a similar to a producer/consumer model. Since library writers are responsible for thread safety of components, additional overhead in the runtime system is only needed for object creation, deletion, and reference counting. This additional overhead should be minimal if the relative frequency of object creation is low compared with the amount of computation. This should be the case for the application domains we are considering.

#### 4 Analysis and Future Work

We have proposed a new approach to language interoperability for high-performance scientific applications based on Interface Definition Language (IDLs) techniques. IDL technology would enable computational scientists to use the programming language most appropriate for the task at hand, or to mix legacy software libraries, without concern about implementation languages and interoperability. Furthermore, IDL approaches may solve the very difficult problem of interoperability with Fortran 90 codes.

In this paper, we have emphasized the advantages of IDLs for language interoperability. We see other advantages, as well. Object oriented IDLs provide a common language for specifying object oriented interfaces to numerical libraries. The IDL run-time system also provides support for advanced object oriented features—such as run-time type identification, cross-language error reporting mechanisms, and multiple inheritance—even for those languages that do not directly support object oriented features, such as C or Fortran. Object oriented features have been built into C libraries by hand [1, 2], but an IDL compiler automates this tedious process.

We see two potential weaknesses in the IDL approach. First, the overheads of the IDL run-time system and glue code may be too high for high-performance scientific computing. We believe that the overheads can be limited to about the cost of a C++ virtual function call; most function bodies will contain sufficient work to amortize this small overhead. Second, scientific programmers—traditionally a very conservative group—may not be willing to accept the naming conventions dictated by the IDL compiler or may not be willing to rely on yet another software library. We believe that the benefits of language interoperability and support for object oriented abstractions in C and Fortran will more than outweigh these disadvantages.

To date, we have completed a parser that reads the IDL grammar described in this paper, and we are currently implementing the IDL type checker. Implementation of the run-time system will be straight-forward, since it provides only basic facilities for error handling, run-time type identification, and object reference counting. Next, we will implement the glue code generation routines for the various target languages. We will begin with C, C++, and Fortran 77 to validate our approach and study inter-language performance overheads. Finally, we will implement the glue code generators for Java, Python, and Fortran 90.

### Acknowledgements

Much of this work was motivated by discussions at Common Component Architecture (CCA) workshops. The CCA group consists of representatives from DOE laboratories and academia working towards the specification of a component framework for high-performance scientific computing.

### References

- [1] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, *Efficient management of parallelism in object oriented numerical software libraries*, in Modern Software Tools in Scientific Computing, E. Arge, A. M. Bruaset, and H. P. Langtangen, eds., Birkhauser Press, 1997, pp. 163–202.
- [2] ———, *PETSc home page*, 1998. See <http://www.mcs.anl.gov/petsc>.
- [3] D. Beazley, *SWIG: An easy to use tool for integrating scripting languages with C and C++*, in The 4th Annual Tcl/Tk Workshop, 1996. 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, Minneapolis, MN, March 1997.
- [5] G. Eddon and H. Eddon, *Inside Distributed COM*, Microsoft Press, Redmond, WA, 1998.
- [6] 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.
- [7] 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>.
- [8] Java Grande Forum. See <http://www.javagrande.org>.
- [9] JAVA SOFT, *Java Native Interface Specification*, May 1997.
- [10] MESSAGE PASSING INTERFACE FORUM, *MPI: A Message-Passing Interface Standard (v1.1)*, June 1995.
- [11] J. G. Mitchell, J. J. Gibbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. L. Powell, and S. R. Radia, *An overview of the Spring system*, in Proceedings of Compcon Spring 1994, February 1994.
- [12] OBJECT MANAGEMENT GROUP, *The Common Object Request Broker: Architecture and Specification*, February 1998. Available at <http://www.omg.org/corba>.

- [13] Paul Dubois, personal communication. See <http://xfiles.llnl.gov/CXX.Objects/cxx.htm>.
- [14] J. Shirley, W. Hu, and D. Magid, *Guide to Writing DCE Applications*, O'Reilly & Associates, Inc., Sebastopol, CA, 1994.