Language Interoperability Mechanisms for High-Performance Scientific Applications*

Andrew Cleary† Scott Kohn† Steven G. Smith† Brent Smolinski†

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 `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 multi-language 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 implementation language 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. Published in the *Proceedings of the SIAM Workshop on Object-Oriented Methods for Interoperable Scientific and Engineering Computing*, Yorktown Heights, NY, October 21-23, 1998. LLNL report UCRL-JC-131823.

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

One interoperability mechanism used successfully by the distributed systems community [7, 13, 15] and the operating systems community [5, 6, 12] 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 package from other programming languages.

In this position paper, we 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 [13] 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 it 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 in Section 2 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. Although a careful design of the language bindings can address these issues, the difficulty grows with the size of the library and the number of supported languages.

2.2 Pairwise Language Interoperability

Another technique for language interoperability uses foreign invocation libraries or automatically generated glue code to support calls between two targeted languages or 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` [14] 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 would 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 techniques described in the previous section except that it unifies all targeted 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 package. For each supported language, an IDL designer defines a language mapping that maps constructs in the IDL onto that target language. For example, an `interface` specification in an object oriented IDL might be mapped onto a `class` in `C++` or an `interface` in `Java`. Glue code is automatically generated by an IDL compiler that takes as input the IDL description of the software component and a language mapping for the target language. The IDL approach reduces the $O(N^2)$ potential language mappings of the previous section to only $O(N)$, since every language is 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 [15] are standard mechanisms for specifying remote procedure call interfaces and have been used widely for both UNIX and Windows NT client/server programming. Microsoft borrowed heavily from the OSF/DCE IDL for its Component Object Model (COM) IDL specification [5, 11]. CORBA [13] is an industry-wide specification for a distributed object system; it describes object interfaces in an IDL that hides language and operating system dependencies. The ILU (Inter-Language Unification) project [7] is applying similar techniques to explore language interoperability in a distributed object environment.

The primary drawback to the IDL approach is that interface specifications are limited to the facilities and types expressed in the IDL, which may be a subset of the capabilities in the targeted languages. For example, a `C` pointer cannot be expressed in an IDL interface if that IDL that does not support the pointer type. However, this does not necessarily imply 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` [13].

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 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 by the language mappings and the stub glue code. For example, the IDL function `dot()` given above would be mapped to method `Solver::Vector::dot()` following CORBA `C++` mapping conventions or 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 numerical 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. Indeed, the interoperability needs of numerical libraries for massively parallel computation are the driver for much of this work. 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. Shared memory multiprocessors are becoming increasingly important for scientific computing, and we plan to support a simple model of thread parallelism. The run-time system will be thread-safe, and library writers will be responsible for managing the parallelism within their objects. This model appears to be most natural for scientific libraries and avoids many of the overheads in other threaded object systems [5].

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.

We have decided to begin with the object oriented CORBA IDL specification [13] and then modify it as necessary for high-performance scientific computing. The CORBA IDL was chosen for several reasons. It is fairly elegant with a syntax similar to `Java` or `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. We will be able to leverage these language mappings and the other work in the CORBA community.

In the following sections, we describe some of the issues in modifying 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 and get/set methods. These constructs may be included later, if warranted. We have replaced CORBA's fixed-length arrays and sequences with dynamic multidimensional arrays, as described below.

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, dynamic 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 true multidimensional arrays. As illustrated in the IDL sample code in the beginning of this section, arrays are specified as `array<TYPE,N>`, where `TYPE` is the type of the array (e.g., `double`) and `N` is the array rank.

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. We plan to evaluate three potential solutions. The first would automatically convert the array to the representation assumed by the implementation language. Thus, arrays passed to a Fortran library from a C application would 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 are 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 within 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 current CORBA specification (v2.2) [13] supports neither 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 allow method redefinition (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 multiple superclass methods share the same signature; references to such methods are 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 its 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 is 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

The IDL language mapping determines how IDL features are mapped onto the target language. The CORBA specification defines IDL language mappings for `C`, `C++` and `Java` [13]. 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 in the expanded `Fortran 90` type system. Strings that are `char *` in `C` and `C++` will become `character *(*)` in `Fortran` with different termination conditions (`NULL` for `C` and `C++` but an explicit length for `Fortran`). Array 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 1 illustrates the differences in naming and parameter passing conventions for two `f90` compilers. There are two important points. First, the compilers generate different linker 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).

Once these calling conventions are established for each `f90` compiler, the IDL compiler will automatically generate the glue code necessary to tie `Fortran 90` 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 mechanics.

3.4 Run-Time Support Library

Because the various software modules linked by our IDL approach will share the same address space, we will require only a minimal run-time support library. In comparison, distributed object systems require a substantial run-time system (e.g., the CORBA Object

```
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. *These* `C` *functions are high-level representations of the assembly code generated by two* `f90` *compilers for the* `Solver` *example in the text. The code on the left was generated by the Sun* `f90` *compiler (v1.2) and the code on the right by the SGI* `f90` *compiler (v6.2).*

Request Broker [13]) to manage distributed object references and parameter marshaling. Our run-time libraries will provide basic support for object reference counting, safe type casting up and down the type hierarchy, method introspection, and basic error reporting. We will implement multiple inheritance using basic function table pointer dispatch mechanisms similar to that used in `C++` or `PETSc` [1, 2].

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 laguages 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, object introspection, 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`, `Fortran 90`, and `MatLab`.

Acknowledgments

Much of this work was motived by discussions at Common Component Architecure (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] JavaSoft, *Java Native Interface Specification*, May 1997.

[10] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard (v1.1)*, June 1995.

[11] Microsoft Corporation, *Component Object Model Specification (Version 0.9)*, October 1995. Available at `http://www.microsoft.com/oledev/olecom/title.html`.

[12] 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.

[13] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, February 1998. Available at `http://www.omg.org/corba`.

[14] Paul Dubois, personal communication. See `http://xfiles.llnl.gov/CXX_Objects/cxx.htm`.

[15] J. Shirley, W. Hu, and D. Magid, *Guide to Writing DCE Applications*, O'Reilly & Associates, Inc., Sebastopol, CA, 1994.