CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2006; **18**:151–162
Published online 11 October 2005 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.909

Bridging the language gap in scientific computing: the Chasm approach

C. E. Rasmussen[1,*,†], M. J. Sottile[1], S. S. Shende[2] and A. D. Malony[2]

[1]*Advanced Computing Laboratory, Los Alamos National Laboratory, Los Alamos, NM, U.S.A.*
[2]*Department of Computer and Information Science, University of Oregon, Eugene, OR, U.S.A.*

SUMMARY

Chasm is a toolkit providing seamless language interoperability between Fortran 95 and C++. Language interoperability is important to scientific programmers because scientific applications are predominantly written in Fortran, while software tools are mostly written in C++. Two design features differentiate Chasm from other related tools. First, we avoid the common-denominator type systems and programming models found in most Interface Definition Language (IDL)-based interoperability systems. Chasm uses the intermediate representation generated by a compiler front-end for each supported language as its source of interface information instead of an IDL. Second, bridging code is generated for each pairwise language binding, removing the need for a common intermediate data representation and multiple levels of indirection between the caller and callee. These features make Chasm a simple system that performs well, requires minimal user intervention and, in most instances, bridging code generation can be performed automatically. Chasm is also easily extensible and highly portable. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: Fortran 95; C; C++; language interoperability; XML; compilers

INTRODUCTION

Fortran is an integral part of the computing environment at major scientific institutions. It is the language of choice for developing applications that model complex physical, chemical, and biological systems. The Fortran[‡] language (in particular, Fortran 95) provides a familiar and friendly environment for scientific programmers at the application level.

*Correspondence to: C. E. Rasmussen, Los Alamos National Laboratory, CCS-1, B287, Los Alamos, NM 87545, U.S.A.
†E-mail: crasmussen@lanl.gov
‡In this paper, the term *Fortran* without a version qualifier is assumed to refer to the latest Fortran version supported by most compiler vendors.

Contract/grant sponsor: U.S. Department of Energy; contract/grant numbers: W-7405-ENG-36 and LA-UR-03-3957

Received 19 June 2003
Revised 3 October 2003
Accepted 3 May 2004

The C++ language, while not having the long heritage of Fortran, is increasingly being used in scientific programming. While Fortran is most frequently used at the application level, C++ is most frequently chosen for software development at the library level. This has lead to a language gap between tools written in C++ and applications written in Fortran (that need to access the C++ tools). The Chasm toolkit was designed to bridge this gap.

The language interoperability issue is not new. Many libraries and application frameworks provide both Fortran 77 and C interfaces (MPI [1], for example). However, with modern languages like C++ and Fortran 95, simple Fortran 77 and C interfaces are often not enough or are inconvenient to use. For example, Fortran 90 formally introduced arrays into the Fortran type system. This caused interoperability problems because a Fortran, assumed-shape array dummy argument (e.g. `integer, dimension(:) :: a`) is passed by an array descriptor (meta-data describing the array) rather than just a raw memory pointer. C++ also has interoperability issues with templates, function overloading, and classes, all of which are tricky to express in Fortran. Furthermore, information that is necessary to link with C++ programs is not explicitly specified by the C++ standard, making symbol name mangling compiler dependent. Similarly under-specified language features exist in Fortran, such as array arguments and module procedure names, again making code generation compiler specific.

There have been many tools created to aid with language interoperability. Perhaps the most well known is SWIG [2], a software development tool that connects programs written in C and C++ with a variety of high-level (normally interpreted) programming languages. Other tools provide language interoperability as part of an object or a distributed object mechanism. These systems are based on Interface Definition Languages (IDLs) and include the Common Object Request Broker Architecture (CORBA) [3], the Inter-Language Unification system (ILU) [4] and Babel [5].

Chasm takes a different approach to language interoperability than do systems requiring IDLs. Chasm eliminates the need for IDLs by using compiler-based tools and static source analysis to generate an XML-based representation of procedure interfaces. This simplifies life for Chasm users as they are not required to create IDL files describing procedure interfaces. In addition, since Chasm is based on compiler tools and language-inclusive XML representations, it provides a more complete mapping between Fortran and C++ than do language-neutral IDL approaches.

Furthermore, Chasm focuses only on language interoperability, as opposed to Babel, which provides an object model, and CORBA, which provides a distributed-object model. The concentration only on language interoperability reduces the complexity of the tools for both Chasm users and Chasm developers. Also, because Chasm does not require translation to and from a common, polylingual type system, it allows optimizations for better run-time performance.

The paper proceeds as follows. First, a summary of related work and tools is provided, followed by a description of the Chasm architecture. Then we describe the XML schema and the tools used to generate it from C, C++, and Fortran source code (and Java compiled bytecode). Next, a brief outline of the XML stylesheet transformations (XSLT), used to generate interlanguage bridging code, is given. Finally, a summary of the paper can be found in the concluding section.

RELATED WORK

Many modern programming languages have some mechanism for interoperation with procedures written in other languages. These range from sophisticated foreign function interfaces such as those found in OCaml [6], Python [7], Ruby [8], and the Java Native Interface (JNI) mechanism [9], to traditional and very simple C-style interfaces. Frequently, this type of language interoperability is provided within the language implementation and is bilingual (e.g. between Java and C). Older, and more widely used languages in the scientific community, languages such as C, C++, and Fortran, provide little or no proscribed functionality dedicated to interoperability with foreign languages. The exception to this is the next Fortran version, F2003 [10], which includes a specification for interoperability with C (see also [11]). In most instances, it is the user's responsibility to ensure that symbols follow the appropriate conventions and that any required type conversions take place.

Perhaps the most promising multi-lingual environment is the Common Language Infrastructure (CLI), adopted by the European Computer Manufacturers Association (ECMA) standards body [12]. The C# language exploits most of the features of the CLI, as do other languages in Microsoft's .NET environment. The key to interoperability among CLI languages is a unified type system. This allows language interoperability to be managed entirely by the compiler, linker, and loader, all without programmer intervention. (Another system that takes advantage of compiler technology is Concert [13].) Unfortunately, CLI is a relatively new technology and has not spread into the high-performance computing environment, although it and the related .NET technology are rapidly being adopted in the larger commercial computing community.

The Concert and CLI systems fit the definition of seamlessness (transparency of interoperability) given by Barrett [14] in that few or no demands are placed on the programmer to achieve interoperability. At the other end of the spectrum are systems that use an IDL, essentially requiring users to learn a new language in order to specify procedure interfaces. These systems (e.g. CORBA [3] and ILU [4]) are frequently elaborate middleware solutions geared toward distributed computing. Babel [5] is an interoperability solution that is specifically geared to high-performance computing (HPC) and also uses an IDL (SIDL, for Scientific IDL).

While position papers, such as [15] and [16] highlight the limitations of IDLs, the claim that IDLs are a least-common-denominator solution is not always accurate. For example, because the Babel project provides users with the capability of developing SIDL object classes in Fortran 77, in a sense, Babel can be seen as providing language extensions, not just langauge interoperability. Babel provides types (SIDL objects) that are not present in the original language, so clearly it is not a least-common-denominator solution. Chasm does not provide such a capability. Chasm allows users to call methods on C++ objects from Fortran, but it does not provide the capability of *developing* object classes in Fortran[§].

IDL-based solutions restrict users to programming language types that the IDL is capable of representing. Solutions such as Concert [13], a multilanguage distributed programming system, assume that the interface specification is the responsibility of the programming language, not of a separate IDL. This allows a wider range of types to be passed between communicating language pairs, with the only restriction that corresponding types be isomorphic between the two languages (see [17]).

Concert treats the IDL as an intermediate language generated by the compiler. This is much the same approach taken by Chasm and by its predecessor SILOON [18], an interoperability system geared to scripting languages (as is SWIG [2]). In this case, the IDL need not even be human readable.

[§]Objects come to Fortran in F2003 as extensions to user-defined types.

The Chasm approach uses an XML format based on the intermediate representation generated by C, C++, and Fortran compiler front-ends. Unlike Concert, however, Chasm does not require special compilers for producing executable code.

Perhaps the 'holy grail' for language interoperability has been laid down by Barrett [14] in his definition of a *polylingual system* (and implemented in the PolySPIN systems [19]):

> 'Informally, a polylingual system is a collection of software components, written in diverse languages, that communicate with one another transparently. More specifically, in a polylingual system it is not possible to tell, by examining the source code of a polylingual software component, that it is accessing or being accessed by components of other languages. All method calls, for example, appear to be within a component's language, even if some are interlanguage calls. In addition, no IDL or other intermediate, foreign type model is visible to developers, who may create types in their native languages and need not translate them into a foreign type system.'.

ARCHITECTURE

Chasm adopts the principals of a polylingual system while producing code for ordered pairs of languages operating in a HPC environment. Details of the Chasm architecture are provided in this section. For simplicity, the bilingual capability of Chasm is described in terms of making Fortran procedure calls from the C++ language. However, this is for ease of description only, as, in principle, the Chasm approach can be used for general, bilingual interoperability. While Chasm-generated bridging code is bilingual, calls to procedures written in a number of different languages may be made from one program, simply by generating bridging code for each language set.

Requirements

There are several key requirements made of the Chasm architecture.

1. The architecture should provide language interoperability between ordered pairs[¶] of languages (*caller* and *callee*).
2. Interoperability mechanisms should be as efficient as possible, especially in a HPC environment.
3. Interfaces to foreign procedures should appear natural within the calling language.
4. Distributed objects do not need to be supported.
5. The architecture should place a minimal burden on the user.

These requirements both restrict and free the design of Chasm. Requirement 1 frees Chasm from a polylingual requirement. This may lead to better performance because it is not necessary to transform/copy caller types to a general intermediate type system and then to transform again to the callee type system. Requirement 3 hinders efficiency because it normally requires control flow to pass through at least one bridging procedure (a stub). Requirement 4 allows for run-time

[¶]Ordered pairs make the important distinction that bindings are dependent on which language is acting as caller versus callee. The reverse generally requires significantly different bindings to be created.

efficiencies (no network and no need to support heterogeneous architectures). It also means that Fortran is not transformed into an object-oriented language based on a distributed-object model. Fortran programmers may call an object-oriented language, but doing so does not force them to design and write Fortran as object-oriented code. Finally, requirement 5 implies that users should not need to describe interfaces in a foreign language (an IDL). Interfaces in Chasm are determined (and published in XML) by the compiler, which has full access to interface information. Users are not forced to modify or even look at these files, unless they wish to change or update interface meta-data, thus providing more information to the tools that subsequently use the XML.

Design overview

Chasm users must complete a series of steps to make calls between foreign languages.

1. *Static analysis.* During the first phase, users process their source code using Fortran or C++ compiler front-ends supplied by the Program Database Toolkit (PDT) project [20]. The PDT uses commercial compiler front-ends supplied by Mutek for Fortran [21] and Edison Design Group (EDG) for C++ [22]. The front-ends parse a program and emit its abstract syntax tree in the compilers intermediate language (IL) representation. An IL analyzer then parses this language specific representation and produces a uniform program database (PDB) format text file common to Fortran and C/C++. The DUCTAPE library provides an API for accessing entities represented in the PDB file. The PDB file is translated to an equivalent XML form in the next phase. The final output of this phase is a human readable XML file describing the interfaces discovered by the compiler. Information regarding the XML schema is provided in the next section.
2. *Bridging-code generation.* These tools take as input the XML file generated in phase 1 and output stub and skeleton bridging code (described below).
3. *Compile and link.* This final phase is accomplished in a familiar fashion specified by the user. Generally this is with a makefile. In fact, all three stages can be automated by combining them within a makefile. Note that this meets the final requirement above. The only burden placed on users is to create the makefile. Note that in rare instances, caller and callee types may not be isomorphic and Chasm users may have to modify the callee interface. For example, Fortran does not have a type equivalent to a C++ iterator, so a C++ function taking an iterator as a parameter is not naturally callable from Fortran.

Stub and skeleton interfaces

Language interoperability is provided by stub and skeleton interfaces. This code is generated by language transformation programs, which input the XML intermediate form published by the compiler and output the stub and skeleton interfaces. These interfaces are described below.

A diagram showing control flow for a C++ caller and Fortran callee is shown in Figure 1. The caller makes a call to a C++ stub interface. The stub adapter hides C++ users from the various symbol-naming conventions used by Fortran compiler vendors. For instance, Fortran symbols are sometimes upper case, sometimes lower case (or mixed as in C) and there may be one or two trailing underscores appended. In addition, the stub adapter hides the name mangling of Fortran module symbols. In summary, the stub provides a natural C++ interface to the user and hides compiler dependencies such as symbol names.

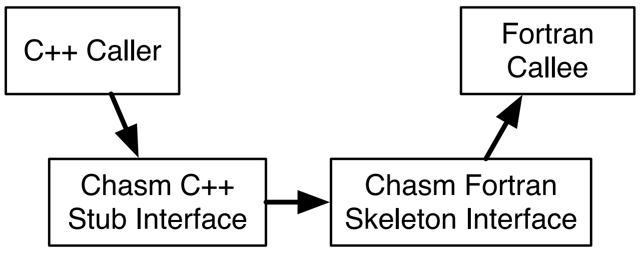

Figure 1. Control flow between a C++ caller and Fortran callee.

The stub passes control by calling a Fortran skeleton interface. The responsibility of both the stub and the skeleton is to marshal C++ parameters to the final interlanguage call, made from the Fortran skeleton to the Fortran callee. The stub performs preliminary marshalling of parameters from C++ to Fortran, while the skeleton completes the marshalling to the exact type required by the Fortran callee.

Recall that no IDL is required by Chasm. The programmer of the callee writes a normal Fortran procedure (or module procedure) and it is the compilers responsibility to discover (and publish) the interface. The callee interface may have Fortran derived types, Fortran assumed-shape arrays, or pointers to either of the two. Since pointers and array-valued parameters are passed in a compiler-dependent fashion (see [23]), it is the responsibility of the Fortran skeleton interface to complete the marshalling to the final form specified by the callee interface and take care of any compiler dependencies arising in this process.

As mentioned above, the stub and skeleton bridging code is generated by XML transformations. Depending on the types in the callee interface, sometimes the skeleton is not necessary and control flow passes directly from the stub interface to the callee, thus increasing efficiency. Sometimes, the stub interfaces may even be inlined, further increasing efficiency.

INTERFACE REPRESENTATION IN XML

Ideally, a language interoperability tool would let a user with an existing library create a description of the library calling interface in some representation language, and use this description repeatedly in generating wrapper code. Users could then include this interface description with a compiled version of their code, so that third-parties could discover the calling interface, with well-defined language bindings, without requiring source code. Systems such as CORBA and Babel do so by using an IDL, while languages such as Java embed the interface information in the compiled code. In fact, this embedded Java interface information is used by Chasm tools to generate XML descriptions for Java class libraries from compiled bytecode, thus eliminating the need for separate processing of source code, as needed for C, C++, and Fortran files. XML was chosen as the IDL because XML is a standardized format with a wide and rapidly growing base of users and tools.

The Chasm XML schema is designed to represent interface characteristics consistently for Fortran, C, C++, and Java. Other languages such as Ruby and Python can be represented, although their lack of type information prior to run-time makes the automatic discovery of their interfaces difficult, particularly when creating bindings to call from explicitly or strongly-typed languages. This simply means that Chasm can easily build code to allow Python to call Fortran, but the Python type system makes it more difficult when calling in the reverse order. For Python, users would need to hand generate the required XML interface description file.

The XML schema takes a hierarchical approach to capturing interface information and type visibility, similar to namespaces in C++, packages in Java, and general scoping rules in most languages. An UML-style representation of the entities in the XML format are shown in Figure 2.

The outermost container in the XML schema is the library, allowing a single XML file to contain descriptions for multiple scopes. A *scope* in this context is roughly equivalent to a Java package, C++ namespace, or Fortran module. Each scope can contain methods, structures (such as classes, C structs, or Fortran user defined types), and other scopes. Structures and methods use the *type* element to describe in detail the type information related to method arguments and structure fields. The *kind*, *ikind*, and *fkind* attributes are enumerations that include all built-in types for C, C++, and Fortran (the reader should refer to the current Chasm XML schema or PDT DUCTAPE API for further details [24,25]). To decouple the concept of a pointer and an array from the actual type of the elements comprising the memory being pointed at, type elements are related to *indirection* elements (for pointers) and *array* elements. Additional elements such as the *proceduremap* (for aliasing procedure names in Fortran) are provided to contain information relevant at code generation time for bookkeeping and emitting specialized wrapper code (*code* elements).

Source-based XML generation

To rapidly bootstrap the necessary XML files for Chasm code generators, a tool is provided that uses source-level interface analysis to walk the original code and emit a default XML representation. We differentiate between this 'default' representation and what may be a slightly different final representation due to a lack of sufficient information to properly map an ambiguous type to the corresponding actual type. For example, pointers in languages such as C and C++ introduce ambiguity (see Figure 3), particularly when dealing with multi-dimensional arrays and character strings.

In the Chasm XML representation, high-level array and string types are available that map to basic pointers in C and C++. An automated tool with no semantic information to indicate how parameters are used within the function body, forces it to take the most conservative guess when emitting XML. The user is responsible for updating this XML to reflect the true high-level types to better assist tools that will use the XML later[‖].

Unlike a significant number of tools for source code analysis that use their own parsers, the Chasm XML generation tool is based on the Program Database Toolkit (PDT), which in turn uses commercial

[‖]The next generation of XML generation tools will allow users to provide 'hints' so that it can better identify when low-level pointer types map to high-level array and string constructs.

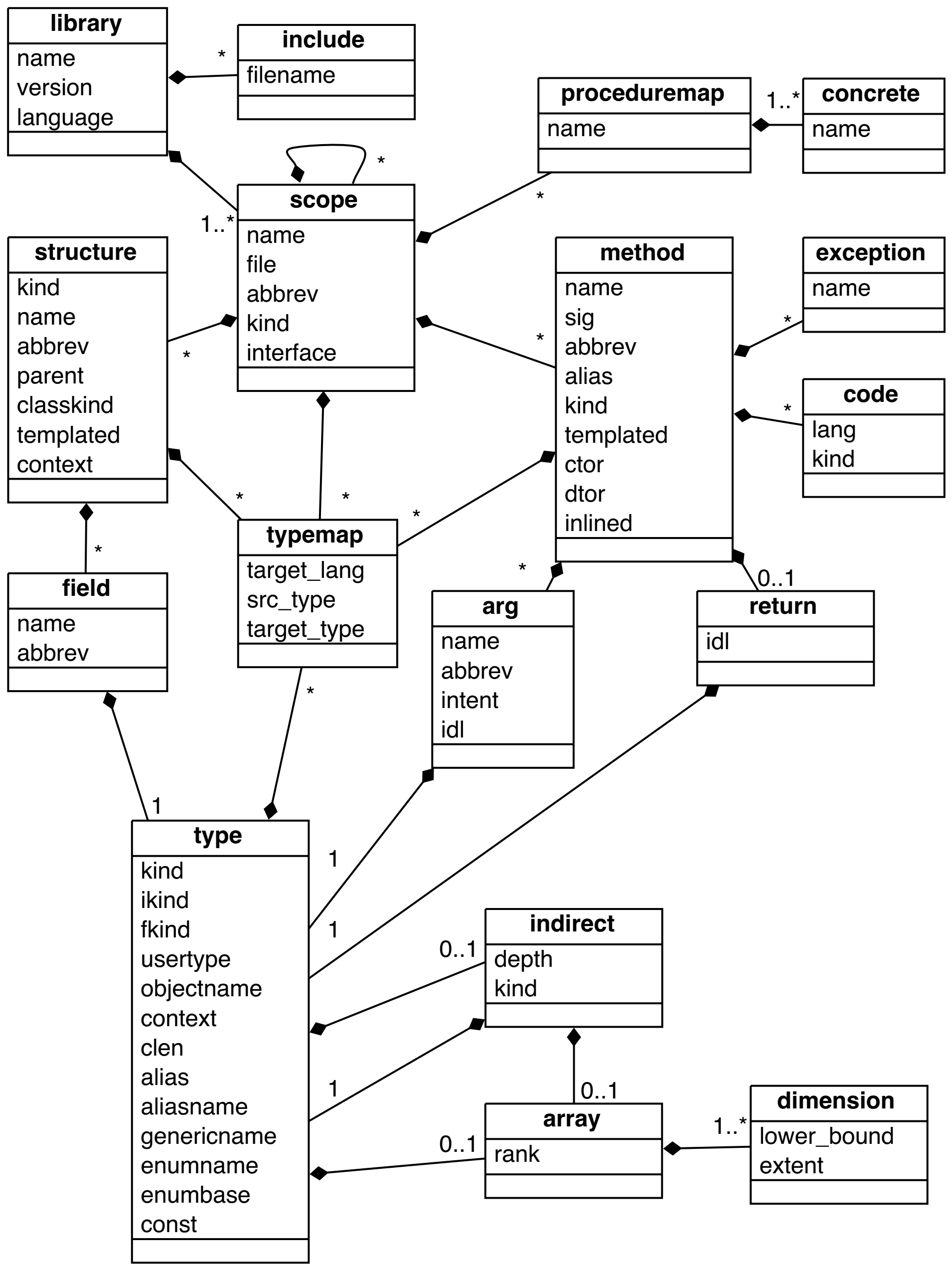

Figure 2. UML representation of the Chasm XML schema.

```
/* ambiguity in C pointers */
 void example(int *1) {
   /* the following are both syntactically correct, but
      semantically different */
   *i = 1;    /* i points to a single value */
   i[4] = 2;  /* i is a ID array */
}
```

Figure 3. An example of the ambiguity in C with a pointer function argument.

grade C, C++, and Fortran 90 compiler front-ends for source parsing and processing. This allows Chasm tools to easily gain access to information for all features in each language, including some that cause custom parsers to break (in particular, heavy usage of C++ templates). The PDT has licenses to freely distribute binary versions of the compiler front-ends that it supports (for a list of supported platforms, see [25]).

CODE GENERATION USING XML TRANSFORMATIONS

The most important portion of Chasm, particularly to end-users, is the code generation facility that takes XML as input and outputs stub and skeleton bridging code for interlanguage procedure calls. Previous systems that the authors worked on for language interoperability either involved large quantities of hand coding [26], or code generation routines that were very difficult to customize [18]. Beginning with the initial prototyping phase for Chasm, a high priority was given to designing a system that was powerful and *highly extensible*, taking into account lessons learned from the projects cited above.

While not the only choice, XSLT (Extensible Stylesheet Language for Transformations) was chosen as the XML processing language. XSLT is a functional language similar to LISP and ML and is easy to apply to the recursive structures found in XML documents. The code generation phase must create several files. In general, there are a C++ header and implementation file for the stubs and a Fortran file for the skeleton (for calling from C++ to Fortran).

Type conversion

The XML transformation engine must declare and implement stub interfaces in C++ and implement the Fortran skeleton interfaces. For each file there is a corresponding XSLT program to generate the file. These stylesheets are fairly generic in that they all contain functions to write file headers and trailers as well as functions to process callee interfaces as they are encountered in the XML file.

At their core, these latter functions continually map types from one language to another as an XML file is processed. The specific mapping depends on context. For instance, is the variable a return type or is it declared within the parameter list of the procedure? Does a parameter type need a variable declared within the body of a stub function or is the variable declaration within the function parameter list sufficient? All of these decisions and type transformations are done by XSLT functions contained in a type-conversions stylesheet. For example, `type-spec` is a function that generates the return type in a C++ stub declaration. If the Fortran callee is a function returning the type, `integer`, then `type-spec` would produce the string `int` on output.

This makes Chasm easy to extend. One only has to override `type-spec` by placing a custom version in a user-conversions stylesheet and have `type-spec` produce the string `int32_t`, for example. This extensibility is particularly useful when passing arrays between Fortran and C++. While a default C++ array class is provided for the array-conversion type used in C++ stub interfaces, Chasm users are able to substitute their own custom-array class for the default.

Problematic types

There are a few types that provide particular challenges to language interoperability. These include Fortran arrays, pointers and character strings. These challenges are briefly considered below.

Fortran passes an array via a descriptor (sometimes referred to as a dope vector) that contains array meta-data, such as the number of dimensions and the number of elements in each dimension. Array descriptors are not specified by the language and are vendor specific [23]. A C library is provided with the Chasm distribution to create and access array-descriptor information. This library allows users to create, for example, a multidimensional array in C and pass it to Fortran as an assumed-shape array. Fortran pointers are also passed via a descriptor.

The convention for strings is somewhat different. Normally, a Fortran compiler passes a pointer to the character buffer, as do C compilers, although Fortran strings are not terminated with a 0. Rather, the string length is explicitly passed as a hidden parameter at the end of the formal parameter list.

Fortunately, Chasm users do not need to be concerned with the exact nature of how a specific Fortran vendor chooses to pass arrays, pointers, or character strings. The stub and skeleton bridging code hides these compiler-dependent details.

CONCLUSIONS

Chasm is a toolkit providing language interoperability between Fortran 95 and C++. Its equivalent of an IDL is created by a compiler pass that outputs an XML file. Stub and skeleton interfaces are machine generated from the XML by XSLT stylesheets. Chasm stylesheet functions are easily extensible to provide custom interlanguage type conversions by overriding the default XSLT functions. For C++ to Fortran interfaces, normally Chasm does not require human intervention, so the entire language interoperability process can be automated in normal project makefiles. C-based languages have more ambiguity (recall the C pointer example discussed above), so minor modifications to XML files may be required.

Several advantages are seen to using Chasm over the traditional method of hand-modifying user code and making the interlanguage function calls directly from the caller's code. First it removes the shear drudgery of hand coding and removes compiler dependencies from user code. However, perhaps more importantly, it means that bridging code will not become out of sync with changes in user code, because normally a simple make will regenerate the bridging code. Chasm generated code is less susceptible to errors than hand-generated code.

Furthermore, because Chasm is easily extensible, it can be modified on a project-wide basis to meet project needs. This allows projects to impose project-wide language transformation standards so that individual project programmers need not 'invent' their own, thus leading to surprises and ultimately to errors. There need not be any long-term dependencies on Chasm, because each project 'owns' the generated code and it can be distributed with project libraries. The open source license of Chasm allows source code for the small Chasm run-time library to be distributed with a user's source distribution.

While Chasm [24] is 'bilingual', providing point-to-point interoperability between Fortran and C++, Chasm stylesheets can easily be modified to provide calls from other languages to Fortran or C++. Experimental versions have been created that automatically create bridging code for calls from Ruby to Fortran. More creatively, proof-of-concept stylesheets have been created that automatically wrap Fortran modules as Common Component Architecture (CCA) [27] components.

Chasm is continually being developed. It has good support for users seeking to call Fortran from C++, however work is currently underway to improve the Fortran compiler front-end distributed with the PDT. Fortran primitive types and array types are well supported. Future improvements include better support for Fortran derived types and more complete support for generating bridging code for calls from Fortran to C++. While the Fortran to C++ XSLT stylesheets are immature, they have been successfully used to produce Fortran interfaces to the C++ Visualization ToolKit (VTK) [28].

REFERENCES

1. The Message Passing Interface (MPI) Standard, version 1.1. http://www-unix.mcs.anl.gov/mpi [26 August 2004].
2. Beazley DM. SWIG: An easy to use tool for integrating scripting languages with C and C++. *Proceedings of the 4th Annual Tcl/Tk Workshop*, 1996. USENIX Association: Berkeley, CA, 1996. Available at: http://www.usenix.org.
3. Pope A. *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Addison-Wesley: Reading, MA, 1998.
4. Cutting D, Janssen B, Spreitzer M, Wymore F. *ILU Reference Manual*. Xerox Palo Alto Research Center: Palo Alto, CA, 1993.
5. Kohn S, Kumfert G, Painter J, Ribbens C. Divorcing language dependencies from a scientific software library. *Proceedings of the 10th SIAM Conference on Parallel Processing*, 2001, Koelbel C, Meza J (eds.). SIAM: Philadelphia, PA, 2001.
6. Rémy D. *Using, Understanding, and Unraveling the OCaml Language. From Practice to Theory and Vice Versa* (*Lecture Notes in Computer Science*, vol. 2395). Springer: Berlin, 2002; 413–537.
7. van Rossum G. Extending and embedding the Python interpreter. http://www.python.org/doc/ext/ext.html [26 August 2004].
8. Østerlie T. Ruby. *Linux Journal* 2002; **95**:93–94.
9. Liang S. *Java Native Interface: Programmer's Guide and Specification*. http://java.sun.com/docs/books/jni [26 August 2004].
10. Fortran 2003, ISO/IEC JTC1/SC22/WG5 N1578. http://www.j3-fortran.org [26 August 2004].
11. Nagle D. What's new in Fortran 2000. *ACM Fortran Forum* 2003; **22**(2):9–12.
12. Stutz D, Neward T, Shilling G. *Shared Source CLI Essentials*. O'Reilly: Sebastopol, CA, 2003.
13. Auerbach JS, Russell JR. The concert signature representation: IDL as intermediate language. *ACM SIGPLAN Notices* 1994; **29**(8):1–12.
14. Barrett D. Polylingual systems: An approach to seamless interoperability. *PhD Dissertation*, University of Massachusetts, Amherst, MA, February 1998.
15. Wileden JC, Kaplan A. Middleware as underwear: Toward a more mature approach to compositional software development. Workshop on Compositional Software Architectures. http://www.objs.com/workshops/ws9801/papers/paper061.html [26 August 2004].
16. Kaplan A, Ridgway J, Wileden JC. Why IDLs are not ideal. *Proceedings of the 9th International Workshop on Software Specification and Design*. IEEE Computer Society Press: Washington, DC, 1998; 2–7.

17. Auerbach J, Burton C, Raghavachari M. Type isomorphisms with recursive types. *Research Report RC 21247*, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY, August 1998.
18. Rivenburgh RD, Rasmussen CE, Lindlan KA, Mohr B, Beckman PH. Automatic generation of Perl extensions to C++ and Fortran 90 class libraries. *O'Reilly Open Source Software Convention*. O'Reilly: Sebastopol, CA, 2000.
19. Kaplan A, Wileden JC. PolySPIN: Support for polylingual persistence, interoperability and naming in object-oriented databases. *Technical Report TR-96-004*, Department of Computer Science, University of Massachusetts, Amherst, MA, 1996.
20. Lindlan KA, Cuny J, Malony AD, Shende S, Mohr B, Rivenburgh R, Rasmussen C. A tool framework for static and dynamic analysis of object-oriented software with templates. *Proceedings of SC2000: High Performance Networking and Computing Conference*. IEEE Computer Society Press: Washington, DC, 2000.
21. Fortran 90 front end documentation. http://www.mutek.com [3 March 2001].
22. Compiler front ends for the OEM market. http://www.edg.com [26 August 2004].
23. Rasmussen CE, Lindlan KA, Mohr B, Striegnitz J. CHASM: Static analysis and automatic code generation for improved Fortran 90 and C++ interoperability. *2001 LACSI Symposium*. Los Alamos National Laboratory: Los Alamos, NM, 2001 (CD-ROM).
24. Chasm language interoperability tools. http://sourceforge.net/projects/chasm-interop [26 August 2004].
25. Program Database Toolkit. http://www.cs.uoregon.edu/research/paracomp/proj/pdtoolkit [26 August 2004].
26. Sottile MJ, Malony AD. INTERLACE: An interoperation and linking architecture for computational engines. *Proceedings of EuroPar*, 1999 (*Lecture Notes in Computer Science*, vol. 1685). Springer: Berlin, 1999; 135–138.
27. Armstrong R, Gannon D, Geist A, Keahey K, Curfman-McInnes L, Parker S, Smolinski B. Toward a common component architecture for high performance scientific computing. *Proceedings of the 8th International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press: Washington, DC, 1999.
28. The Visualization Toolkit. http://public.kitware.com/VTK [26 August 2004].