

Babel Fortran 2003 binding for structured data types

Stefan Muszala¹, Tom Epperly² and Nanbor Wang¹

¹ Tech-X Corporation, Boulder CO 80301, USA,
muszala@txcorp.com, nanbor@txcorp.com

² Lawrence Livermore National Laboratory, Livermore, CA 94551,
LLNL-CONF-403478, epperly2@llnl.gov

Abstract. Babel is a tool aimed at the high-performance computing community that addresses the need for mixing programming languages (Java, Python, C, C++, Fortran 90, FORTRAN 77) in order to leverage the specific benefits of those languages. Scientific codes often rely on structured data types (structs, derived data types) to encapsulate data, and Babel has been lacking in this type of support until recently. We present a new language binding that focuses on their interoperability of C/C++ with Fortran 2003. The new binding builds on the existing Fortran 90 infrastructure by using the `iso.c.binding` module defined in the Fortran 2003 standard as the basis for C/C++ interoperability. We present the technical approach for the new binding and discuss our initial experiences in applying the binding in FACETS (Framework Application for Core-Edge Transport Simulations) to integrate C++ with legacy Fortran codes.

Key words: Language Interoperability, Fortran 2003, Structured Data Types, Babel, C/C++

1 Introduction

Babel is a language interoperability tool that is part of the Common Component Architecture (CCA) [1], a DOE-sponsored initiative that provides Component-Based Software Engineering (CBSE) [2, 3] support to the High-Performance Computing (HPC) community. Babel allows C, C++, FORTRAN 77, Fortran 95, Java and Python to operate in a single address space [4] through code automatically generated from a programming language neutral specification provided in the Scientific Interface Definition Language (SIDL).

A new area of Babel research is providing support for the structured data type (`record types`, `structs` in C, C++, SIDL and `derived types` in Fortran) as a vehicle of transferring collections of data between languages. The structured data type is relied on heavily in HPC and scientific codes to supply a level of data encapsulation. Providing a SIDL `struct` type allows direct access to data members with better performance than when using a SIDL `class/interface` [5]. SIDL `structs` will also reduce the amount of hand written code and provide increased compatibility with languages that have structured data types [5].

```

package packageAll version 1.0{
  struct typeAll{
    int          intAll;
    double       doubleAll;
    array<double,1,column-major> arrayDoubleOneD;
    rarray<double,2> staticRarray(100,100);
    rarray<int,1> dynamicRarray(iY);
    int iY;
  };
  class classAll{
    void procAllOne(inout typeAll structInOut,
                   in typeAll structIn);
    typeAll funcIllegal(); // Illegal struct as return value
                          // because it contains an r-array.
    void procIllegal(out typeAll structOut); // Illegal out mode
                                             // because typeAll contains an r-array.
  }
}

```

Fig. 1: Example SIDL file showing SIDL primitive, array and r-array type for use in the F2003 binding. Note the illegal SIDL method signatures.

Babel satisfies the specific needs of the HPC community by providing interoperability with scientific programming languages. Other efforts at language interoperability such as SWIG [6] and F2PY [7] offer interoperability solutions for the HPC community but do not provide the generality that Babel does.

The many approaches to Fortran 90/95 interoperability with C/C++ usually depends on a particular Fortran compilers implementation. The Fortran 90/95 **sequence** attribute guarantees order but not padding which can lead to data misalignment with a corresponding C/C++ **struct**. Fortran 90/95 also does not have a built in mechanism to deal with name-mangling issues between Fortran and C/C++. The Fortran 2003 **iso_c_binding** module provides common naming conventions and type compatibility to match those of C/C++ for improved language interoperability. A portion of the **iso_c_binding** module addresses the process of combining codes containing Fortran 2003 **derived types** and C/C++ **structs**. For example, using the **iso_c_binding** module and **bind(c)** directive in **derived type** declarations guarantees member ordering and padding to be consistent with C/C++. Other features such as converting C to Fortran pointers and easing the Fortran name mangling problem are explained in more detail in the Fortran 2003 standard [8] or Metcalf et al., [9].

The paper proceeds as follows. Section 2 presents a technical description of the approach we have taken in our implementation. It includes a discussion of SIDL primitive, array and r-array types as members of SIDL **structs**. We will also show in Section 2 how we use the **iso_c_binding** module that is a part of the

Fortran 2003 [8] standard as the basis for a Fortran 2003 Babel language binding that is interoperable with C/C++. We discuss our initial experiences in using the Babel Fortran 2003 binding when calling legacy Fortran codes from the FACETS (Framework Application for Core-Edge Transport Simulations) C++ framework in Section 3. Performance data that compares Babel to existing approaches in FACETS to language interoperability is shown in Section 4 while conclusions and future directions are presented in Section 5.

2 Technical Implementation

There are two phases in implementing Fortran 2003 struct support in Babel. They are 1) adding additional grammar to the Babel parser to define structs in SIDL and 2) supplying the code generation that translates the SIDL input files into F03 bindings. Our implementation allows SIDL `struct` members consisting of SIDL primitive types, array types and r-array types.

2.1 SIDL Parser

The SIDL grammar now contains a new `struct` type that allows SIDL primitive, array and r-array types as members. The SIDL array type provides generality and a rich API for accessing data but does not allow direct Fortran array access. R-arrays on the other hand exist to provide more efficient, lower level access to numeric arrays [10]. Using r-arrays within SIDL `structs` is a need that HPC users have requested and allows easier adaptation of Babel to legacy scientific codes. To allow direct array access within structured data types we extend the SIDL `struct` grammar to allow r-arrays (raw arrays) as members.

```

module packageAll_typeAll
  use, intrinsic :: iso_c_binding
  use sidl_double_array
  use sidl_int_array
  type, bind(c) :: packageAll_typeAll_t
    integer(c_int32_t) :: intAll
    real(c_double) :: doubleAll
    type(c_ptr) :: arrayDoubleOneD
    real(c_double) :: staticRarray(100,100)
    type(c_ptr) :: dynamicRarray
    integer(c_int32_t) :: iY
  end type packageAll_typeAll_t
end module packageAll_typeAll

```

(a) Babel generated Fortran 2003 derived type corresponding to the SIDL definition shown Figure 1.

```

struct packageAll_typeAll__data {
  int32_t intAll;
  double doubleAll;
  struct sidl_double__array*
    arrayDoubleOneD;
  double staticRarray[100][100];
  int* dynamicRarray;
  int32_t iY;
};

```

(b) Babel generated C/C++ derived type corresponding to the SIDL definition shown in Figure 1.

Fig. 2: Babel code generation corresponding to the SIDL definition shown in Figure 1.

R-arrays introduce memory management issues that are not present when using the more general SIDL array. Memory creation and deletion associated with r-arrays within `structs` must be handled by the user. The r-array data is not managed by Babels reference counting scheme (like the SIDL array) and hence memory can be accidentally leaked. To minimize the memory management issues and retain r-array functionality we added parse-time restrictions on SIDL `structs` that contain r-arrays as members. The first restriction is on the parameter passing mode such that only `IN` and `INOUT` modes are allowed. Similarly, such a `struct` may not appear as a return argument in a function call (Figure 1).

The parameter mode restriction implies that memory allocation should occur on the caller side of a multi-language boundary and is the paradigm we recommend when dealing with r-arrays as `struct` members. Unfortunately, there is no practical way to enforce the caller side memory-management on an r-array that is a member of an `INOUT struct` and in this case all of the burden of memory management is placed on the end-user. There are no such restrictions on `structs` containing only SIDL primitives and SIDL array types as members.

The Fortran 2003 standard and `iso_c_binding` module dramatically improves interoperability with C/C++ but a small number of shortcomings remain. For example, the `c_ptr` kind provided by the `iso_c_binding` module is only applicable when converting C pointers to Fortran allocated arrays and may not be used as a pointer to a `derived type`. Babel requires access to all array metadata including stride information. In cases where we need additional functionality (particularly when dealing with SIDL arrays) we have relied on Chasm [11] for interoperability.

SIDL Type	F2003 type(kind)
BOOLEAN	logical(c_bool)
CHAR	character(c_char)
INT	integer(c_int32_t)
LONG	integer(c_int64_t)
FLOAT	real(c_float)
DOUBLE	real(c_double)
FCOMPLEX	complex(c_float_complex)
DCOMPLEX	complex(c_double_complex)
OPAQUE	integer(c_int64_t)
STRING	type(c_ptr)
ENUM	integer(c_int64_t)

Fig. 3: SIDL primitive types mapped to their corresponding F2003 `iso_c_binding` type and kind.

2.2 SIDL Types

Once we decided how to extend the SIDL grammar and parsing rules for `structs` to take advantage of the Fortran 2003 standard we could concentrate on adding the relevant code generation required to support SIDL primitive types, array types and r-array types as structured data type members.

An overview of example code generation and type implementation in SIDL `structs` that are useful in the context of this discussion are shown in Figure 2a (Fortran 2003 `derived type`) and Figure 2b (C/C++ `struct`). The codes in Figure 2 are based on the SIDL `struct` definition presented in Figure 1.

SIDL Primitive Types The Fortran 2003 `iso_c_binding` module provides specific Fortran kinds that enable a direct mapping to C/C++ and SIDL types and are listed in the left column of Table 3. The right-hand column of Table 3 lists the corresponding Fortran 2003 `iso_c_binding` types and kinds that facilitate C/C++ to Fortran interoperability.

SIDL Array Types

The SIDL array type exists in order to generalize the use of many native array types built into various programming languages [10]. The binding we provide for Fortran 2003 and `structs` must adhere to this philosophy and retain the original API's functionality. Figure 4 presents a

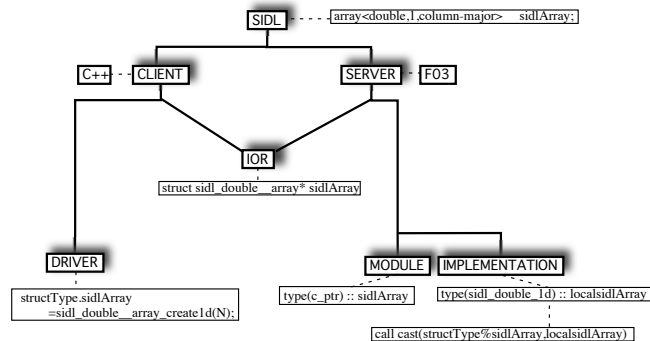


Fig. 4: Overview of the Fortran 2003 Babel binding for the SIDL Array Type.

high level view of our approach. The SIDL definition of `sidlArray` provides the basis for the “interoperability glue” of the client (caller) and server (callee, also implementation). The glue code consists of 1) Intermediate Object Representation (IOR) files where the language interoperability actually occurs 2) Stub code that facilitates the method call from client to IOR and 3) Skel code that facilitates the method call from IOR to implementation.

The C++ client code provides a mechanism that creates the array type (`_create1d()`) within the `struct` and then passes that `struct` as a pointer through the IOR files to the Fortran 2003 side. The Fortran 2003 server code consists of module and implementation files. The Fortran 2003 `derived data type` definition of `sidlArray` in the module files contains the `bind(C)` type `c_ptr` which allows the C++ pointer to be passed directly to the Fortran side. At this point we include a local declaration of the type within the `struct` on which the SIDL array API may be applied. The `c_ptr` type is then cast to a local type using Chasm [11] which provides the underlying Fortran array descriptor that is a part of the SIDL array type.

SIDL R-array Types The SIDL r-array type allows direct access to array data without having to use the accessor functions as required when us-

ing the SIDL array type. We provide two types of r-arrays, static and dynamic. Static r-array support requires a pre-defined size to be declared in a SIDL definition of an r-array (`rarray<int,1> rarrayStructMember(5);`). Since all array or matrix sizes are known at compile time, the C/C++ declaration would be `double rarrayStructMember[5];` while the Fortran 2003 declaration is `real(c_double)::rarrayStructMember(5)`.

Implementing dynamic SIDL r-arrays that allow a user to `malloc`, `new` or `allocate` their data and then pass that data across the language boundary requires a few additional steps. Like the SIDL array type, the r-array is represented as a pointer in the Babel IOR files and the `c_ptr` `bind(c)` kind is once more used in Fortran. At this point the r-array implementation diverges from that of the array type.

Instead of using Chasm as the casting mechanism for the array `c_ptr` the `r_array c_ptr` is converted to a native Fortran array using the `c_f_pointer` routine that is defined by the Fortran 2003 standard [8]. The difference is that the SIDL array type is itself represented in Babel as a structured data type and we need Chasm to set those structured data type members equal to the individual components of the Fortran array descriptors in order for the accessor functions to work. Given the SIDL definition of `rarray<int,1> structRaw(iY)`; the conversion in Fortran occurs as shown in Figure 5 and includes an additional required local type declaration.

```

real(c_double),dimension(:),pointer :: structMember_fptr
...
call C_F_POINTER(CPTR=struct%structMember, &
                  FPTR=structMember_fptr, &
                  SHAPE=(/DIM/))
! =====
! The above is auto-generated by Babel
! and from this point below, the user can
! manipulate the r_array data directly.
! =====
do i=1,struct%N
    structMember_fptr(i)=structMember_fptr(i)+5.0
enddo

```

Fig. 5: Calls required to convert a pointer to a usable Fortran r-array type. DIM is declared in a module and in this case we define N as a struct member as well.

3 Multicode integration in FACETS

We have started the conversion to Babel and the new F03 binding as the language interoperability tool for a portion of the Framework Application for Core-Edge Transport Simulations (FACETS) [12] project. The FACETS infrastructure is written in templated, object oriented C++ but uses legacy Fortran codes that model transport fluxes of which there are a number of algorithmic and implementation differences. The current method of C++ to Fortran interoperability relies on the FC_FUNC autotools macro for name-mangling and uses a contiguous piece of memory on the C++ side that is manually aligned with the Fortran **derived types** for language interoperability. A key benefit to using Babel is that once a SIDL definition is in place there then exist common interfaces so that different models may be “plugged-in” to FACETS. A domain scientists may easily prototype a model (for example in Python) and test it using the common interface definition without implementing their own glue and integration code.

The transport models contain derived types with primitive types and statically allocated array members and we use SIDL to provide exact mappings of types (recall Figure 3). The one-to-one correspondence (for example statically allocated Fortran arrays map to SIDL r-arrays) retains computational efficiency and facilitates a straightforward conversion to SIDL and Babel. Figure 6 shows parts of the SIDL file for the transport models and how **structs** and **classes** are organized.

Initial prototype implementations that couple the FACETS C++ code with legacy Fortran code indicates that Babel provides a more general solution to the coupling problem than what is currently in place. Babel not only provides multi-platform and multi-compiler support, but we are finding that there are fewer levels of subroutine calls that are exposed to the user when using Babel. In effect the “interoperability glue” code required to combine languages is hidden better than it was in the original implementation. Other benefits include that 1) the cross-language

```

package fmcfmWrap version 1.0{
  struct MagGeom{};
  struct SurfVars{};
  struct Gradients{};
  struct Flags{
    ...
    double          glfCnu;
    rarray<int,1>    glfIflagin(5);
    rarray<double,1> glfXparam(30);
    int              mmmNroot;
    ...
  };
  struct diflux{};

  class fmcfm{
    ...
    void siCalcFlux(in string modelType,
                   inout MagGeom eqMG,
                   inout SurfVars sV,
                   inout Gradients sG,
                   inout Flags genflags,
                   out diflux flux,
                   out int ierr);
    ...
  }
  class fmcfmUtil{} // Wrapping of
                   // getter and setter routines
}

```

Fig. 6: Extracts from the SIDL file used to wrap the transport models for the FACETS integration.

We also executed performance tests in which we place timers around the procedure call of the FACETS C++ framework that calls the Fortran transport modules that are described in Section 3. While not an exact test in that the timing data includes the measurement of the work in the Fortran procedures themselves, it is still useful for comparing the current implementation (Section 3) to the Babel implementation and we find that there is no discernible difference between either of the implementations. The Babel implementation uses the r-array type exclusively and all memory management is handled on the C++ caller side.

All tests, including those shown in Figure 7, were performed on an AMD Opteron. We used gcc 3.4 and the NAG Fortran compiler for the tests in Figure 7 and gcc and gfortran 4.3 for the transport module timing test.

5 Future Directions

Application level work that remains involves further and on-going integration of Babel into the FACETS project as well as the coupling of additional legacy codes. This work will also give us feedback from users and developers as to what features should be added to Babel. Babel implementation involves on-going study of structured data type interoperability for other Babel supported languages (eg. mapping SIDL `structs` to Java final classes) as well as refining the existing Fortran functionality.

References

1. R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski, "Toward a Common Component Architecture for High-Performance Scientific Computing," in *Proceedings of IEEE Symposium on High Performance Distributed Computing*, (Redondo Beach, CA), IEEE, 1999.
2. G. Heineman and B. Councill, *Component-Based Software Engineering*. Reading, Mass.: Addison-Wesley, 2000.
3. C. Szyperski, *Component Software: Beyond Object-Oriented Programming, 2nd Edition*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 2002.
4. S. Kohn, G. Kumfert, J. Painter, and C. Ribbens, "Divorcing language dependencies from a scientific software library," in *10th SIAM Conference on Parallel Processing*, (Portsmouth, VA), March 2001.
5. T. Epperly, "Preliminary thoughts on introducing structs to SIDL/Babel," tech. rep., Lawrence Livermore National Laboratory, 2004.
6. "Swig." <http://www.swig.org>.
7. "F2py: Fortran to python interface generator." <http://cens.ioc.ee/projects/f2py2e/>.
8. Fortran 2003 Final Committee Draft, J3/03-007R2 <http://www.j3-fortran.org>.
9. M. Metcalf, J. Reid, and M. Cohen, *Fortran 95/2003 explained*. Great Clarendon St., Oxford OX2 6DP: Oxford University Press, 2004.
10. T. Dahlgren, T. Epperly, G. Kumfert, and J. Leek, *Babel User's Guide*. CASC, Lawrence Livermore National Laboratory, Livermore, CA, babel-0.9.4 ed., 2004.
11. "Chasm Language Interoperability Tools." <http://chasm-interop.sourceforge.net/>.
12. "Facets web site." <http://www.facetsproject.org>.