UCRL-TR-201771

Preliminary Thoughts on Introducing Structs to SIDL/Babel: White Paper

Tom Epperly

January 7, 2004

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

This work was performed under the auspices of the U. S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

Preliminary Thoughts on Introducing Structs to SIDL/Babel
White Paper

Tom Epperly <epperly2@llnl.gov>
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

7 January 2003

1 Introduction

In the past 6 months, there has been increased interest in adding something analogous to C structs to the SIDL language and the Babel language interoperability tool [2, 6]. In particular, Rob Armstrong, of the Common Component Architecture [4], said the lack of structs "is an oft-cited reason that people can't use Babel." Because the interest is high and lack of structs is a barrier to Babel adoption, we must carefully consider the current work around, the motivations for structs, the implications of adding structs, and the alternatives for structs in SIDL/Babel. This document provides the background necessary for a discussion of structs in SIDL/Babel.

For the purposes of this document, I am going to call the potential new language feature a *SIDL struct*. The SIDL struct is analogous to a C struct, a Pascal record, or a Fortran 90 (F90) derived data type. It is a collection of data with no methods or behavior associated with it. Each element of the collection has a name and a type. SIDL structs allow for data abstraction, but they do not provide data hiding. All data is public in a SIDL struct.

2 Data Abstraction Using SIDL Classes

Today's Babel supports data abstraction using SIDL classes. Babel users can create a class as a functional equivalent of a struct. The class has a get and set method for each element stored by the class.

Here is a concrete example. Here is a C struct containing entries for the month, day and year.

```
struct date_t {
  int month;
  int day;
  int year;
};
```

The following SIDL class is a functional equivalent for `date_t`.

```
class Date {
  int getMonth();
  void setMonth(in int month);
  int getDay();
  void setDay(in int day);
  int getYear();
  void setYear(in int year);
}
```

The SIDL class declaration is twice as long as the C struct, and the SIDL class must also be implemented. Overall, the SIDL class is likely to require 10–20 times as many hand written lines to declare and implement as the corresponding C struct. Each access (getting or setting) to a data element incurs the overhead of a Babel method evaluation.

On the positive side, the SIDL class version hides how the data is actually stored. The underlying implementation could be something like the original C struct shown above, or it could be a single integer [7]. In some situations, hiding the underlying storage of the data might be useful.

3 Objectives

There are a variety of motivations behind the interest in adopting SIDL structs. It is important to identify all the objectives for SIDL structs because different alternatives satisfy some objectives at the cost of others. With feedback from the community, we can choose the alternative that satisfies the most important criteria. These objectives are based on discussions with people about structs and our reflection on adding structs to SIDL.

Performance Using a SIDL class/interface instead of a SIDL struct requires a multi-language function call for each get/set operation. For numeric data types, a Babel multi-language function call is roughly 2.6 times the cost of a normal C or Fortran function call [3]. In comparison, accessing an element of a struct is much computationally cheaper and easier for the compiler to optimize. In some cases, SIDL structs may be passable without any marshalling or copying.

Development A SIDL struct does not require implementation like the SIDL class approach does. Structs can reduce the amount of hand written code.

Completeness Struct is a useful concept found in many successful languages. Sometimes it is useful to group data together without the extra effort of making a class. Many languages have special syntax for accessing structs, and programmers prefer using the clean syntax as opposed to making method calls.

Compatibility CORBA [5] and WSDL [8] have the struct concept, and future compatibility with these IDL-based systems may require structs.

4 Implications

Adding SIDL structs has a ripple effect on SIDL. There are a variety of related questions that must be answered. Here, we will try to list the issues and offer commentary where possible.

4.1 Can SIDL struct contain any SIDL type?

If SIDL has structs, I imagine that most programmers will expect it to be a general purpose data structure able to hold any SIDL type. Allowing SIDL structs to contain strings, booleans, object references, interface references or array references will require marshalling in the Fortran 90 binding and perhaps other bindings. If SIDL structs can have object, interface or array references, the struct will need a destructor to release all contained references and perhaps string data.

4.2 Will there be arrays of structs too?

SIDL has arrays of every other type, so it seems that we ought to have arrays of SIDL structs as well.

4.3 Can SIDL structs contain SIDL structs?

This is really a special case of the first question. Babel clients will probably want them.

4.4 Should SIDL structs be reference counted?

I don't know enough at this time to comment on this question. If SIDL structs are reference counted, they will need a virtual function table too.

5 Alternatives

To evaluate each alternative, we need to consider how it maps onto C, C++, Fortran 90 and Java — the Babel supported languages that have native support for the struct concept. I have to drop Java from consideration because I do not know enough about JNI.

For any SIDL struct alternative, the Fortran 77 binding is basically the same. Babel will generate F77 functions to get/set each element of the struct. If SIDL structs have embedded structs (i.e., a struct contained by another struct), it could take multiple calls to get at the embedded data. You would make a call to get a handle to the embedded struct, and then you make a call to get data from the embedded struct handle. Note, the embedded struct handle is dependent on its parent handle. If someone destroys the parent, the embedded handle is a dangling reference. Reference counting could prevent this problem.

The Python binding can also handle any alternative with a single strategy. Babel would generate a Python C extension module to wrap the underlying SIDL struct. The Python wrapper would map the SIDL struct elements to Python object attributes. Python has the same dangling reference problem that Fortran 77 has.

The following example illustrates the dangling reference problem. We start with a SIDL declaration of a struct using syntax that may not bear any resemblance to the syntax ultimately used (if any).

```
package a version 0.1 {
  struct Momentum {
    double weightCarried;
    double averageFlyingSpeed;
  }

  struct Bird {
    string birdType;        // e.g., African or European
    struct Momentum data;
  }
}
```

Let's assume that the syntax for creating a struct in Python is analogous to the syntax for creating objects. This Python code segment demonstrates the dangling reference problem.

```
  swallow = a.Bird.Bird()
  momentum = swallow.data
  swallow = a.Bird.Bird()    # the first Bird is now freed
  momentum.weightCarried = 0 # writes to member that was freed
```

When the second Bird is allocated, the reference count of the Python wrapper object of the first bird goes to zero, and its destructor fires. The destructor deallocates the SIDL struct leaving the momentum Python wrapper pointing to deallocated memory.

The dangling reference problem can be fixed by having the proxy object keep a reference to its parent to keep the parent from disappearing. However, it complicates the implementation.

5.1 Do nothing

In this alternative, we do not add SIDL structs. We encourage users to write classes to perform the function of structs. This alternative does nothing to address any of the objectives.

5.2 Automatically generated IMPLs

We add SIDL structs to the SIDL language using some reasonable syntax that resembles C, C++ or Java. Struct is just a shorthand way to define a class with no behavior other than storing data. The class corresponding to the struct has get and set methods for each data element of the struct, and Babel will automatically generate the implementation for the struct including the code to serialize the object for RMI (when this feature becomes available).

This alternative satisfies the development objective by bringing the number of hand written lines of code to something comparable to what a developer would write in C, C++ or Java. It partially satisfies the completeness objective, but it does not allow developers to use native struct syntax except for Python. We can write a special Python stub to make data elements appear like

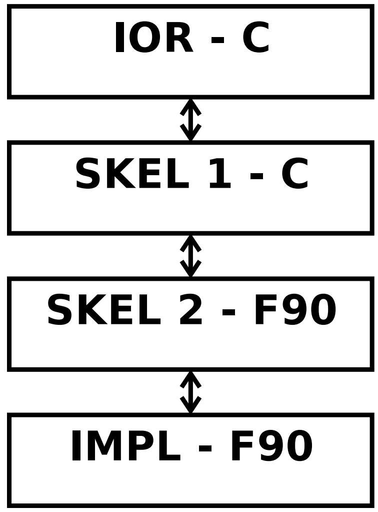

Figure 1: F90 server side stages

attributes. It satisfies the compatibility objective. The performance objective is not improved by this alternative; although, Babel may generate a better C implementation than the developer would have done in the “do nothing” alternative.

5.3 Copy and marshal data approach

SIDL structs map into native structs in C, C++, F90 and perhaps Java. SIDL structs can contain any SIDL type including structs, arrays, objects or interfaces. The IOR uses a C struct, and each of the bindings converts the C struct into a native struct.

For the C++ binding, we assume that the C and C++ compilers map structs to memory in an identical fashion. Babel already makes this assumption with respect to the entry point vector (a struct of function pointers). In the worst case, the Babel user could use the C++ compiler to compile both the C and C++ to ensure structs are laid out identically. If the struct only contains int, long, float and double, it can be used without copying or marshalling. If the struct contains bool, string, fcomplex, dcomplex, array, object or interface values (difficult types), the analogous C++ struct is defined; and the data is copied or marshalled. If a struct contains an embedded struct with difficult types, it must also be copied and marshalled. The marshalling is required for complex types because C++ has a different representation of these types.

For the F90 binding, we do not assume that the compilers map structs and derived data types to memory in an identical manner. The F90 server side has three stages shown in Figure 1. The IOR calls the first part of the skel written in C which calls the second part in F90 which in turn calls the F90 impls. In the skel 1 stage, all the elements of the struct would be converted to function arguments for the call to skel 2. In the skel 2, the individual arguments would be packed into a derived data type before calling the impls.

For a simple SIDL method `void foo(in a.Bird b)`, the skel 1 code would look something like this:

```
static void skel_foo(struct a_Bird_t *b)
```

```
{
  SIDL_F90_STR_LOCAL(_proxy_b_birdType);
  SIDL_F90_STR_COPY(_proxy_b_birdType, b->birdType, 0);
  SIDLFortran90Symbol(a_foo_mi,A_FOO_MI,a_foo_mi)(
    SIDL_F90_STR_LOCAL_ARG(_proxy_b_birdType)
    SIDL_F90_STR_NEAR_LEN(_proxy_b_birdType),
    &(b->data.weightCarried),
    &(b->data.averageFlyingSpeed),
    SIDL_F90_STR_FAR_LEN(_proxy_b_birdType));
  free((void *)SIDL_F90_STR_LOCAL_STR(_proxy_b_birdType));
}
```

The stage 2 skel (in F90) looks like this:

```
recursive subroutine a_foo_mi(bbirdtype, bdataweightcarried,
                    bdataaverageflyingspeed)
  character(len=*), intent(in) :: bbirdtype
  real(selected_real_kind(16,307), intent(in) :: &
    bdataweightcarried, bdataaverageflyingspeed
  type(a_Bird_t) :: b
  allocate(b%birdType, len(bbirdtype))
  b%birdType = bbirdtype(1:len(bbirdtype)) ! not sure
  b%data%weightCarried = bdataweightcarried
  b%data%averageFlyingSpeed = bdataaverageflyingspeed
  call a_foo_impl(b)
  deallocate(b%birdType)
end subroutine a_foo_mi
```

The developer writing the F90 impl sees a normal F90 derived data type coming in. The client-side F90 bindings would require a similar two stage stub where all the structs are flattened and turned into function arguments.

This approach completely satisfies the development, completeness and compatibility objectives, and it provides some increase in performance over previous alternatives. For the C++ binding, there is copying/marshalling where needed, and for the F90 binding, there is copying/marshalling all the time. There is some performance benefit from doing all the copying/marshalling in 1 function call as opposed to a function call per element needed.

This approach does not require any more knowledge about the F90 compiler than Babel already requires. The approach might run into trouble with excessive numbers of arguments causing it to run into a maximum line length, maximum argument length, or maximum stack size limitations.

5.4 Minimal copying and marshalling approach

This approach differs from the previous only in its F90 binding. It requires a test suite or a knowledge database to indicate when F90 lays our derived data types (using sequence) identically to C. The C++ binding is identical to the previous alternative. The F90 binding is set up to be more efficient when possible.

If the configuration determines that F90 has the same layout as C, it sets a preprocessor flag. Babel writes two versions of each method with a struct argument in skel 1 and skel 2. One version copies and marshals everything like the previous alternative, and the other version passes structs without difficult types straight through (similar to the C++ binding). At compile time, it chooses the right version based on the preprocessor flag set during configuration.

Several F90/95 manuals indicate that F90 derived data types are laid out identically to C structs for simple numeric arguments. For example, Sun's documentation indicates that structs can be passed between C and F90 "as long as the corresponding elements are compatible" [11]. The Lahey Fortran 95 User's Guide also indicates compatibility between C structs and F95 derived types [10]. The Absoft documentation says that a `STRUCTURE` can be defined to be compatible with a C struct, but `STRUCTURE` does not appear to be standard F90/95 [1]. For the IBM XL compiler suite, derived types are compatible with C structs for numeric types provided the C is compiled with `-qalign=packed` [9]. These findings suggest that this approach is feasible for a variety of compilers; although, the documentation is brief and may neglect to mention cases where it does not work.

This approach has the best performance assuming an F90 compiler that maps structs identically to C exists. It satisfies all other objectives. It minimizes copying and marshalling, and it presents structs in native data structures for C, C++ and F90. It is also the most difficult to implement. It has the same liabilities as the previous alternative in cases when the F90 compiler does not match the C compiler.

6 Conclusions

There are a variety of feasible alternatives to introduce SIDL structs. Most of the desirable alternatives require a considerable amount of work. If the Babel community indicates that structs are essential, we will seek additional funding to support work in this area.

References

[1] absoft development tools and languages. *Pro Fortran: Linux*, 2003.

[2] Babel home page. `http://www.llnl.gov/CASC/components/`.

[3] David Bernholdt, Wael Elwasif, James Kohl, and Thomas Epperly. A component architecture for high performance computing. In *Proceedings of the Workshop on Performance Optimization via High-Level Languages and Libraries*, 2002.

[4] Common component architecture home page. `http://www.cca-forum.org/`.

[5] CORBA component model. http://www.omg.org/technology/documents/formal/components.htm.

[6] Tammy Dahlgren, Tom Epperly, and Gary Kumfert. *Babel User's Guide*. CASC, Lawrence Livermore National Laboratory, version 0.8.8 edition, October 2003.

[7] Nachum Dershowitz and Edward M. Reingold. Calendrical calcultions. *Software—Practice and Experience*, 20(9):899–928, September 1990.

[8] E. Christensen et al. Web Services Description Language (WSDL) 1.1, March 2001. `http://www.w3.org/TR/wsdl`.

[9] IBM. *XL Fortran for AIX User's Guide*, June 2002.

[10] Lahey Computer Systems, Inc. *Fortran 95 User's Guide*, revision d edition, 2003.

[11] Fortran 90 user's guide. `http://docs.sun.com/db/doc/801-5492`.