High-Performance Language Interoperability for Scientific Computing through Babel

Thomas G. W. Epperly Gary Kumfert Tamara Dahlgren
Dietmar Ebner Jim Leek Adrian Prantl Scott Kohn

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, California 94551

Abstract—**High-performance scientific applications are usually built from software modules written in multiple programming languages. This raises the issue of language interoperability which involves making calls between languages, converting basic types, and bridging disparate programming models. Babel provides a feature-rich, extensible, high-performance solution to the language interoperability problem currently supporting C, C++, FORTRAN 77, Fortran 90/95, Fortran 2003/2008, Python, and Java. Babel supports object-oriented programming features and interface semantics with runtime-enforcement. In addition to in-process language interoperability, Babel includes remote method invocation to support hybrid parallel and distributed computing paradigms.**

I. Introduction

Babel is a programming language interoperability toolkit for high-performance scientific computing. It was designed to address specific functional and performance needs in the development of large-scale, multi-physics simulations involving the integration of multiple mathematical models, libraries, and solvers implemented in different programming languages. The inherent complexity of the resulting systems requires the aid of software tools for their development, evolution, and maintenance.

Multi-disciplinary, multi-physics, and multi-resolution applications are far too complex to be developed by a single organization. Hence, the parts — models, libraries, and solvers — are developed by code groups with relevant expertise. Each team often relies on different programming languages and development platforms. Some critical codes may have even been developed by experts long retired. These differences exacerbate the integration challenges that must be overcome to successfully create large-scale applications.

Interoperability between languages involving incompatible programming paradigms and type systems is inherently difficult. For example, dynamic memory management may be a feature of one language but left to the programmer in another. Errors may be reported explicitly versus via dynamic exceptions. Arrays may be represented in column- versus row-major order and their indices start at 0 versus 1. These incompatibilities can make building, debugging, and maintaining associated software systems extremely challenging.

Additional features not natively available across languages are typically required by the numerical libraries that dominate scientific applications. Specifically, dynamic, multi-dimensional arrays, array strides, single- and double- precision complex numbers, and structures are common. The heavy reliance on arrays for managing numerical data is a critical aspect of these applications and it can have a significant impact on performance.

Further adding to the challenges faced by the scientific computing community is the need for software tools to run on commercial as well as one-of-a-kind platforms. Computational scientists often develop their codes on desktop platforms then port them to Top 500[1] machines for high performance runs. Machine and language idiosyncrasies present unique portability challenges requiring a detailed understanding of binary interfaces and linkage con-

[1]http://www.top500.org

ventions.

Finally, the often lengthy execution times — on the order of days to weeks — of scientific simulation runs result in the need to minimize the introduction of additional overhead. Babel was initially designed for fast, in-process communication to address this important issue. The project won the prestigious R&D 100 award [23] in 2006 for “the world’s most rapid communication among many programming languages in a single application.” While Babel’s primary focus is efficient interoperability within a single address space, it also fully supports transparent remote method invocation (RMI).

Large-scale, multi-physics, and multi-resolution computational science and engineering applications of today face significant integration challenges due to their use of numerically intensive, long-running codes written in different (including legacy) programming languages for deployment on one-of-a-kind platforms. Babel addresses the functional and performance needs of the community through a high-performance interoperability toolkit. The motivation for and approach taken to develop the technology is described in Sections II and III, respectively. Details of the toolkit are provided in Sections IV through VI. Applications of Babel are presented in Section VII. Section VIII covers the most relevant related work. Future work is presented in Section IX.

II. Motivation

Interoperability solutions at the time Babel was initially conceived, such as CORBA (Common Object Request Broker Architecture) [35] and COM (Component Object Model) [38], tended to be geared more for general and commercial interests. That is, they generally lacked support for the legacy programming languages and native data types commonly used by computational scientists and engineers. Features for aiding cross-language debugging were also missing.

Traditional scientific programming languages generally lack support for object oriented programming (OOP), which has increasingly been adopted by the community. OOP codifies the discipline of data and procedure encapsulation, thereby facilitating the development of re-usable software. The large set of supported languages, shown in Figure 1, emphasizes languages of interest to the community. Support for traditional scientific programming languages

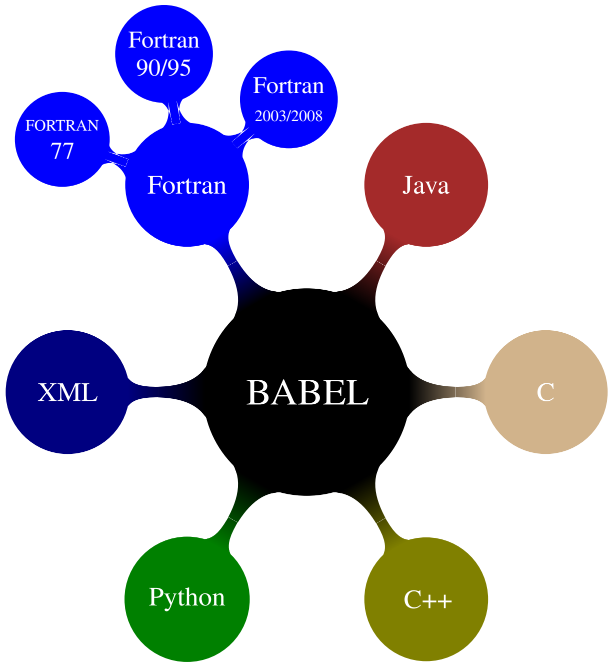

Fig. 1. Programming languages supported in the Babel 2.0 release.

is important due to the significant amount and inherently long lifetimes of legacy codes written in those languages.

Babel, which bridges the gap among the different programming paradigms and languages, enables software written in classical imperative programming languages, such as Fortran and C, to interoperate with interpreted scripting languages, such as Python. To accomplish this feat, Babel must deal with different binary representations, symbol length limitations, and inconsistent rules for identifier declarations (*e. g.*, case sensitivity or reserved symbols).

The numerically intensive computations performed in scientific applications are heavily dependent on the use of array-based and numerical data types. Of particular importance are dynamic, multi-dimensional arrays, array strides, single- and double-precision complex numbers, and structures. Arrays with all these features are not generally native types in modern, general-purpose languages.

Both scientific and general-purpose programming languages lack support for interface contracts, which are a well-known software engineering technique for improving testing and debugging [32]. Interface contracts define and enable the automated enforcement of software behaviors at call boundaries. They

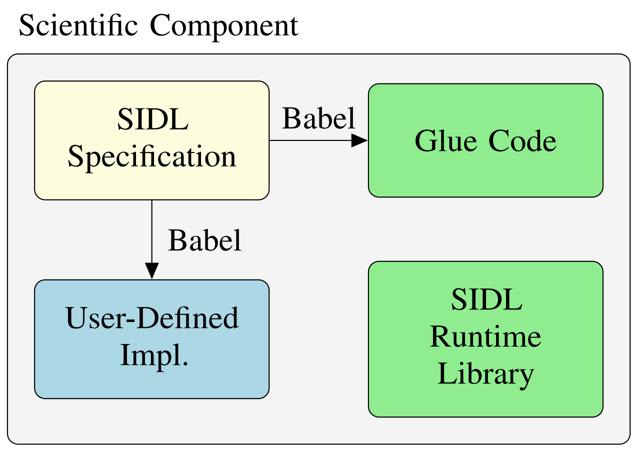

Fig. 2. Babel translates SIDL specifications to glue code and language-specific prototypes.

are also a logical extension of language interoperability solutions for the support of cross-language debugging.

The scientific computing community needs a tailored, high-performance language interoperability toolkit to facilitate the development of complex, large-scale, multi-physics, and multi-resolution applications. Traditional scientific and modern programming languages must be supported to accommodate legacy and new codes. Dynamic, multi-dimensional arrays are critical to the numerically intensive codes that must interoperate. Ensuring these multiple programming language applications work correctly requires mechanisms, like interface contracts, for cross-language debugging.

III. Approach

Restricting our approach to the least common denominator across supported languages would result in the loss of critical features in both traditional and modern programming languages. Combining scientific data types with the discipline of object oriented encapsulation led to the development of technologies based on a scientific object model. The resulting Babel toolkit consists of three parts: a programming language-neutral interface specification language, compiler, and runtime library. The Scientific Interface Definition Language (SIDL), developed at LLNL, is the specification language. The Babel compiler translates SIDL specifications into language-specific glue code, prototypes, and documentation. The supporting library is referred to as the “SIDL Runtime Library”. Figure 2 shows the major parts of the toolkit in relation to their use in a scientific software artifact.

SIDL provides a declarative description of the *public* methods of the calling interface as extensions of the scientific object model. The model is defined through base classes, interfaces, methods, exceptions, and built-in types. SIDL, like the CORBA Interface Definition Language (IDL) provided by the Object Management Group (OMG) [35], [10], is programming language-neutral. Both IDLs support the modular packaging of full method definitions specifying the type (*e. g.*, integer, float) and mode (*i. e.*, in, out, inout) of each parameter. Both also support enumerations, arrays, and multiple inheritance of interfaces. Unlike CORBA IDL, SIDL provides basic types for numeric complex and multi-dimensional, multi-strided arrays. Another distinguishing feature is complete support for polymorphism across programming language boundaries. For example, Python may be used to overload a specific method of a Fortran module, throwing an exception implemented in C++. Interface contract clauses with a rich set of expressions are also supported as an aid to testing and debugging.

The Babel compiler translates SIDL descriptions into wrappers used to map between programming language-specific types and the common representation layer. Native language features, such as built-in data types and method overloading, are leveraged in the generated code, whenever possible, with reasonable alternatives used in the remaining cases. Babel maps features to the particular native language ecosystem in order to impose minimal requirements for existing software packages. There is no major feature in Babel that cannot be supported in all existing language bindings. To the user, a component using Babel always appears to be implemented in the particular native language, no matter which combination of languages is actually used. The common representation layer depends on features of the SIDL runtime library to support the object model and contains interface contract enforcement features, when needed. The layer is also instrumented with interface contract enforcement checks, when contracts are defined, that also depend on features in the runtime library.

The SIDL runtime library essentially provides language-specific implementations of language-independent SIDL constructs. It provides basic operations and capabilities associated with SIDL types, such as casting, reference counting, reflection, and implicit exceptions. The runtime also supports object and library management as well as interface contract enforcement.

Babel bridges the gap among different scientific programming paradigms and languages, combining native support for key scientific data types, object orientation, and interface contracts, to produce a tool tailored for scientific language interoperability. The approach involves the generation of source code wrappers from specifications of the calling interface, in SIDL. The generated code is based on an object model (defined in SIDL and implemented in the SIDL Runtime library) with built-in scientific data types and optional interface contracts. The toolkit, therefore, combines the benefits of object orientation and programming by contract [32] with scientific language interoperability.

IV. Scientific Interface Definition Language

The Scientific Interface Definition Language (SIDL) is a programming language-neutral specification language used to define the calling interface. SIDL specifications are formed from eight main elements: packages, interfaces, classes, methods, exceptions, contract clauses, types, and comments [12]. Figure 3 shows an example SIDL specification. The remainder of this section describes each element.

A. Packages

Packages are used to define name space hierarchies. All SIDL entities are required to be part of a package. They consist of a combination of types, primarily in the form of interfaces and classes, and allow for software composition. Packages may be nested and versioned, with the latter — identified through major and minor numbers — being inherited by class and interface declarations. Types defined within an external package may be referenced by importing the package. Package version restrictions can be enforced by import statements.

```
import sidl;
package sort version 1.0 {
  interface Comparator {
    /**
     * Returns −1 if i1  < i2,
     * 0          if  i1  = i2,  and
     * 1          if  i1  > i2.
     */
    int compare(
      in BaseInterface  i1,
      in BaseInterface  i2);
  };

  interface Container {
    int getLength();
    void swap(
      in int i,
      in int j);
  };

  abstract class Algorithm {
    abstract static string getName();
    abstract void sort(
      in Container c,
      in Comparator cmp);
  };

  class List implements−all Container {
    void pushBack(BaseInterface elmnt);
  };

  class QuickSort extends Algorithm {
    static string getName();
    void sort(
      in Container c,
      in Comparator cmp);
  };
}
```

Fig. 3. SIDL specification for a simple class hierarchy. Only List and QuickSort are concrete classes instantiated at runtime. Package `sidl`, which is the basis of Babel's runtime library, provides functionality associated with abstractions like `BaseInterface`.

B. Interfaces

Interfaces define a set of methods a caller can invoke on an object of a class implementing the methods. Figure 3 illustrates the definition of two interfaces: `Comparator` and `Container`. Interfaces are akin to Java interfaces or pure abstract base classes in C++. All member functions of an interface are implicitly abstract. Like Java and C#, only single inheritance and multiple implementation of interfaces is supported to avoid major complications and ambiguities caused by multiple inheritance.

C. Classes

Classes define a set of methods a caller can invoke on an object. Unless explicitly abstract, classes have implementations of each member function. Abstract

```
class Output {
  /* No conflicting declarations. */
  static void print();

  /* User-specified long names are used
   * only for languages without support
   * for method overloading.
   */
  static void print[Int](in int v);
  static void print[String](in string v);
  static void print[Double](in double v);
};
```

Fig. 4. SIDL fragment illustrating overloaded methods.

classes, however, have at least one unimplemented method in order to preclude their instantiation. Figure 3 illustrates the definition of one abstract class — `Algorithm` — and two concrete classes — `List` and `QuickSort`. Data members cannot be declared in SIDL; therefore, the only way to pass data in or out of objects is through methods.

D. Methods

Methods define routines available for invocation by a caller. They represent the *public* interface of an object, therefore all methods implicitly have public visibility. There is no notion of private or protected member functions. SIDL methods are by default *virtual*. This means that the actual method being called is determined at runtime, based on the concrete type of an object. Methods may be declared `final` to prevent them from being overridden.

SIDL also supports the usual notions of `static` methods (*i. e.*, conceptually part of the class but do not execute in the context of a concrete object) and overloaded methods (*i. e.*, methods with the same (base) name but different arguments). The specification of the method name must include an extension, as shown in Figure 4, to disambiguate names in languages that do not support overloading. For languages with built-in support for method overloading, such as C++ and Java, the short name is used while for the remaining languages, such as C, Fortran, or Python, the user-specified suffix is appended to provide an unique identifier.

All methods inherited from an abstract base class or interface remain abstract unless they are re-declared in the class definition. In order to keep class declarations concise, the `implements-all` keyword can be used to revert this behavior for interfaces.

Similar to CORBA IDL, each parameter declaration is preceded by an explicit *mode* specifier, which may either be `in`, `out`, or `inout`, to declare whether a parameter will be read-only or also written to by the method. Implementations are explicitly allowed to return an object different from the one being passed for `inout` parameters, except for special cases such as raw arrays.

Remote Method Invocation (RMI) adds three additional modifiers: `local`, `oneway`, and `nonblocking`. Caller and callee of a `local` method have to share the same address space (*i. e.*, they cannot be invoked on a remote object). A method declared `oneway` can only have `in` arguments and be implemented using unidirectional network messages. Non-blocking methods implement *asynchronous* call semantics. These methods return a so-called ticket that can be used to retrieve return values and out arguments.

SIDL methods define the calling routines of interfaces and classes through method signatures specifying the arguments and return type, if any. Standard object oriented mechanisms for controlling method inheritance, such as *virtual*, *final*, and *static*, are either implicit or defined with method qualifiers. Parameter declarations must indicate the *mode*. Finally, RMI is supported through additional modifiers identifying communication semantics.

E. Exceptions

Exceptions are used to indicate errant behavior. All methods implicitly throw a `sidl.RuntimeException` should there be errors in the Babel generated code or communication. Additional exceptions have to be specified explicitly using a `throws` clause, as shown in Figure 5. Exceptions are mapped to native language features whenever possible; otherwise, they are communicated using an additional generated `out` parameter that must be explicitly checked by the user.

F. Contract Clauses

Contract clauses, which are optional, define constraints on properties of objects as well as argument and return values. Babel supports three clauses: preconditions, postconditions, and class invariants [14].

```
/**
 * Return the dot (, inner, or scalar)
 * product of the specified vectors.
 */
double vuDot(in array<double> u,
             in array<double> v,
             in double tol)
  throws
     sidl.PreViolation,
     sidl.PostViolation;
  require
     not_null_u : u != null;
     u_is_1d : dimen(u) == 1;
     not_null_v : v != null;
     v_is_1d : dimen(v) == 1;
     same_size : size(u) == size(v);
     non_neg_tolerance : tol >= 0.0;
  ensure
     no_side_effects : is pure;
     vuAreEqual(u, v, tol) implies
       (result >= 0.0);
     (vuIsZero(u, tol) and
         vuIsZero(v, tol))
       implies
         nearEqual(result, 0.0, tol);
```

Fig. 5. SIDL fragment illustrating precondition and postcondition clauses for a vector dot product operation. The preconditions for `vuDot` require the two normal SIDL arrays, `u` and `v`, both be non-null, one-dimensional arrays of the same size. The tolerance value must also be non-negative. The postcondition clause indicates all implementations of the method must ensure the following, assuming the preconditions are satisfied: 1) if `u` and `v` are equal then the result of calling `vuDot` will be non-negative and 2) if `u` and `v` are both zero vectors then the result will be within the specified tolerance of `0.0`. The `is pure` assertion within the postcondition clause indicates implementations should be side-effect free so they can be specified within the contract clause of another method; however, the assertion is not executable since Babel does not statically analyze associated implementation(s).

The syntax is borrowed from Eiffel [32]. Figure 5 shows the SIDL specification of precondition and postcondition clauses associated with a vector dot product method. As illustrated, clauses consist of a list of assertions optionally preceded by individual labels used for documentation.

Each clause corresponds to a different set of enforcement points. *Precondition* and *postcondition* clauses, indicated by the `require` and `ensure` keywords respectively, apply to methods. A *precondition* declares constraints on invocation of a method while a *postcondition* constrains its effects. In some cases, there may be properties that need to hold throughout the life of an instance of a class. Rather than require the assertions be specified in the precondition and postcondition clauses of every method, the *class invariant*, which is specified at the interface and class level, is used for these properties. It is important to keep in mind that interface assertions only need to hold at the method call boundary. Depending on the nature of the algorithm, it may be necessary for the implementation to temporarily violate the contract during method processing. However, as long as the corresponding assertions hold at the call boundary, the contract is not technically violated.

Babel supports a variety of operators used to specify the boolean expressions making up contract clauses. Basic operators found in most common programming languages are supported. Conditional operators `iff` (i.e., if-and-only-if) and `implies` enable more expressive contracts. A case where `implies` is useful is ensuring a null pointer is not passed within a contract to a method incapable of supporting it. For example, the null check in the assertion `(outHandle != null) implies (size(outHandle) >= 0)` is needed because the built-in `size()` function does not gracefully handle a null array argument.

In order to support assertions on object properties, which are not visible to the interface, function calls may be specified in contract clauses. It is very important that such methods be side effect-free because the implementation cannot assume contracts will actually be enforced during runtime. The functions may be user-defined methods that are in scope or any of twenty built-in functions listed in Table I. Built-in array accessor and simple numeric value comparator function operate in constant-time. Other array-based operations, including existential and universal quantifiers, are linear in the size of the arrays. Contracts may also include user-defined functions (i.e., methods returning a value) in scope whose contract includes the `is pure` annotation in its postcondition clause. The annotation indicates all implementations of the method are not (supposed to) have any side effects. At this point, there are no tools in the Babel toolkit to verify this property.

Babel interface contracts support the specification of Eiffel-inspired precondition, postcondition, and class invariant clauses. Each clause, when specified,

[2]The $expr$ can be one of: $u\ r\ v$, $u\ r\ n$, and $n\ r\ v$, where $u, v \in$ *SIDL arrays*, $n \in$ *Numbers*, and $r \in \{<, >, <=, >=, ==, !=\}$. The relation $u\ r\ v$ is equivalent to $\forall i \in 0\ ..\ (size(u) - 1),\ u[i]\ r\ v[i]$; $u\ r\ n$ to $\forall i \in 0\ ..\ (size(u) - 1),\ u[i]\ r\ n$; and $n\ r\ v$ to $\forall i \in 0\ ..\ (size(v) - 1),\ n\ r\ u[i]$.

Function	**Returns**
all($expr$)	$True$ if the expression $expr^2$evaluates to $true$ for each element in the specified array(s). For example, `all(`$u < v$`)` returns $true$ if the value of each element in array u is less than the value of the corresponding element in array v.
any($expr$)	$True$ if at least one element in the specified array(s) satisfies the expression $expr^2$. For example, `any(`$u = 0$`)` returns $true$ upon encountering the first element in array u whose value equals zero but returns $false$ if none of the element values is zero.
count($expr$)	The total number of array elements satisfying the expression $expr^2$.
dimen(u)	Dimension of array u.
irange(x, n_{low}, n_{high})	$True$ if x falls within the integer range of $n_{low}..n_{high}$.
irange(u, n_{low}, n_{high})	$True$ if all elements in array u fall within the integer range $n_{low}..n_{high}$.
lower(u, d)	Lower index of the d^{th} dimension of array u.
max(u)	The maximum value of the elements in array u.
min(u)	The minimum value of the elements in array u.
nearEqual(x, y, t)	$True$ if real values x and y are equal within the specified tolerance, t.
nearEqual(u, v, t)	$True$ if the corresponding elements in arrays u and v are equal within the specified tolerance, t.
none($expr$)	$True$ if none of the elements in the specified array(s) satisfies the expression $expr^2$. For example, `none(`$u >= 0.0$`)` returns $true$ if no element in array u has a value greater than or equal to 0.0.
nonDecr(u)	$True$ if the elements in array u are in order by non-decreasing value.
nonIncr(u)	$True$ if the elements in array u are in order by non-increasing value.
range(x, r_{low}, r_{high}, t)	$True$ if the real value x falls within the specified tolerance, t, of the range $r_{low}..r_{high}$.
range(u, r_{low}, r_{high}, t)	$True$ if all elements in array u fall within the specified tolerance, t, of $r_{low}..r_{high}$.
size(u)	Allocated size of array u.
stride(u, d)	Stride of the d^{th} dimension of array u.
sum(u)	Returns the total of the values of all of the elements in array u.
upper(u, d)	Upper index of the d^{th} dimension of array u.

TABLE I
INTERFACE CONTRACT BUILT-IN FUNCTIONS

SIDL Type	**Size [bits]**
`bool`	8
`char`	8
`int`	32
`long`	64
`float`	32
`double`	64
`fcomplex`	64
`dcomplex`	128
`opaque`	64
`string`	varies
`enum`	32
`struct`	varies
`interface`	varies
`class`	varies
`array<Type,Dim>`	varies
`rarray<Type,Dim>` (*index variables*)	varies

TABLE II
FUNDAMENTAL SIDL TYPES

contains one or more assertions. Assertions can be as simple as basic variable expressions consisting of arguments, operators, and numeric or boolean literals. Alternatively, methods — built-in as well as user-defined — may be used in the expressions. Overall, Babel supports a rich variety of assertion expressions within SIDL interface contracts.

G. Types

Types constrain parameter values, exceptions, and return values associated with methods.

Babel features a complete set of fundamental types, listed in Table II, including single and double precision complex numbers. Basic types like `bool`, `int`, or `float` are fixed size with a native equivalent in most target languages. This is different from languages like C where only a hierarchy among types is defined (*e. g.*, the actual size of type `int` in ANSI C is completely implementation dependent but has to be at most the size of `long`). Whenever possible, more complex types, such as `fcomplex` or `string`, are mapped to native language equivalents. Otherwise, a sensible data structure is provided with a language-dependent runtime library. Strings often involve a high runtime overhead as they are represented differently by practically every programming language.

One of the motivating features that distinguishes SIDL most from related approaches, such as CORBA IDL, is its extensive support for arrays. Babel supports three types of arrays with varying performance implications. A simple, regular array type is provided whose contents can be allocated or borrowed. Generic arrays, which have no type or dimensionality, are more flexible than regular arrays, but have fewer built-in features than regular arrays. Finally, raw arrays (i.e., `rarray`) allow low-level access to numeric array content. Their use is limited, however, to C, C++, and Fortran.

The first, or regular, array type provides the full set of features of a SIDL type including reference counting and automatic (de)allocation. Regular arrays are defined with dimension and base type, which can be any fundamental type except `struct`, including interfaces and classes. They can be passed to and returned from methods in any mode and may optionally be declared `row-major` or `column-major`. The framework will physically copy the array to adhere to these specifications, if necessary, which may involve a large runtime overhead. Due to memory management issues, copying will always occur in some cases in the Python and Java bindings. Arrays may either be allocated or borrowed. The first type owns and allocates memory for its data. Borrowed arrays, on the other hand, reference data from another source and only allocate space for the necessary meta information. The rationale therefore is to wrap data from other sources without requiring the data to be physically copied.

To allow for more flexible interfaces, SIDL supports the notion of *generic arrays*, which are array declarations with no type or dimension information. Meta data such as length and dimension is available using an array API in each of the language bindings. Generic arrays are useful to handle a wide range of arguments (*e. g.*, for (de)serialization or logging).

SIDL supports the Fortran concept of array strides, which means that data is not necessarily densely packed. Each dimension can be strided arbitrarily. One-dimensional arrays can be declared row-major or column-major in order to specify dense arrays. Otherwise, row- and column major one-dimensional arrays are identical.

Raw arrays provide a lower-level alternative to numeric arrays in some languages (*e. g.*, a one-dimensional raw array may appear as a double pointer and a length parameter in C). To highlight the contrast, a normal SIDL array is represented by a struct in C, a template class in C++, a 64-bit integer in FORTRAN 77, and a derived type in Fortran 90/95. In higher-level languages such as Java or Python[3], raw arrays appear like regular arrays.

Raw arrays provide for low-level data access but suffer a series of limitations compared to regular SIDL arrays. In particular, raw arrays may only be passed in mode `in` or `inout` and cannot be used as return values. They must be contiguous and are implicitly stored in column-major order. Implementations are not allowed to change the shape of the array or return an array different from the one being passed. Also, raw arrays are restricted to fundamental numerical types. As for structs, a major motivation for raw arrays is to match existing legacy APIs, requiring less or no code to translate from and to SIDL interfaces.

The `opaque` type is a language-independent way to declare language-dependent interfaces. Babel only guarantees that opaque arguments are preserved between caller and callee. In practice, opaque types are sometimes used for pointers to resource handles or language-dependent data structures. However, their use is strongly discouraged as `opaque` types have meaning only within a particular process and introduce limitations in combination with RMI.

Enumerations (`enum`) provide a way to declare types with a limited range of values that can be referred by name instead of hard-codded values. C/C++ developers will find the SIDL syntax very familiar. Concrete numeric values can either be provided in the declaration or are assigned automatically in a meaningful way.

A relatively recent addition to SIDL are structural data types (`struct`). These types provide a natural way to group semantically related data together. Figure 6 shows an example SIDL specification. Structural types are more efficient and require less development effort than regular classes. They also often allow for SIDL specifications of existing interfaces and provide compatibility with related systems such as CORBA or WSDL [9]. A SIDL struct may contain fields of arbitrary type, including (raw) arrays and structs. However, there is currently no support for arrays of structs. Structs are fully compatible with RMI and provide for a very efficient way to

[3]In Python code, SIDL arrays appear as NumPy [24] arrays.

```
import sidl;
package structs version 1.0 {
  struct Rarrays {
    int d_int;
    rarray<double,1> d_rarrayFlex(d_int);
    rarray<double,1> d_rarrayFix(3);
  }
}
```

Fig. 6. SIDL specification for a struct containing a fixed-size and a flexible-size raw array. The size of the flexible array is constrained by the integer element `d_int`.

pass data among various programming languages.

H. Comments

Comments are optional annotations for adding documentation to the SIDL and generated files, where appropriate. SIDL supports basic Java and JavaDoc/Doxygen-style comments. Two cases of the former appear in Figure 4, while examples of the latter appear in Figures 3 and 5. These annotations embed class, interface, and method documentation directly in the SIDL file. JavaDoc-style comments can also be used to create interface documentation and are automatically replicated in the generated files, where appropriate, whenever they directly precede the corresponding declaration.

V. Babel Compiler

The infrastructure required to support SIDL elements is obtained through the automatic transformation of specifications into client-server language interoperability source code using the Babel compiler. There are actually four layers generated: stub, intermediate object representation (IOR), skeleton, and implementation.

A simplified view of the layers involved in a local Babel method invocation is shown in Figure 7. On the client side (caller), a so-called stub is generated that converts arguments to Babel's IOR representation, calls the proper method entry point from the object's entry point vector (EPV), and converts eventual return values to the representation used in the original language. On the server side (*skeleton*), the inverse operations are performed, *i. e.*, arguments are converted from IOR to the particular implementation language, the user-supplied implementation is called, and return values are converted back to Babel's IOR. In addition, the skeleton is responsible to catch exceptions thrown in the implementation and convert them to a language-independent representation.

Achieving transparent n-way language interoperability is a non-trivial task. A naïve approach would lead to a quadratic number of mappings among each pair of languages. Instead, Babel defines a C-based intermediate object representation (IOR) that is used to mediate among arbitrary pairs of languages. The IOR is exactly the same, no matter which language has been used to implement or invoke a particular method. Language bindings only need to generate the necessary code to translate to and from Babel's IOR, which is a significantly simpler problem. This approach is further motivated by the fact that most languages provides at least a C interface to build upon (*e. g.*, JNI, Python extension types, or C Interop in Fortran 2003/2008).

Babel supports virtual function calls even on top of procedural languages such as FORTRAN 77. Consequently, it cannot rely on language features. Instead, it implements its own virtual function table and generates the necessary dispatch code for the various supported languages in the client stubs. Dynamic dispatch using virtual function tables was first introduced in Simula [11] and is today the preferred technique for widely used languages such as C++ [19].

A. Stubs

Stubs are pieces of glue code generated by Babel for each member function serving two main purposes: *(a)* they convert arguments and return values between the native language representation and Babel's IOR and *(b)* they dispatch to the skeleton code via an object's EPV. There is a strict separation between client and server code in order to ensure that components can be distributed in binary form together with the corresponding SIDL file. The method entry points stored in the EPV are the only way to "cross" the barrier between client and server code.

In most language bindings, stubs are implemented in C. The C backend itself is a special case as the IOR is already implemented in C. Thus, no argument conversions are necessary. The stub directly dispatches calls to the user-supplied implementation. For C++, stubs are member functions of a wrapper class closely resembling the SIDL declaration.

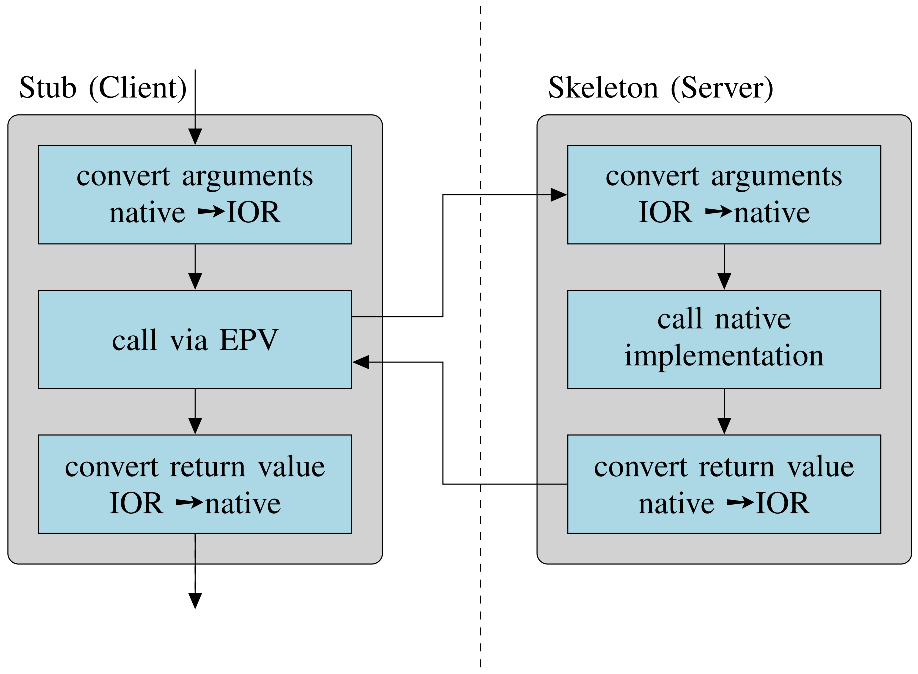
Stub (Client)
convert arguments native → IOR
call via EPV
convert return value IOR → native
Skeleton (Server)
convert arguments IOR → native
call native implementation
convert return value native → IOR

Fig. 7. Arguments and return values are converted to Babel's intermediate object representation (IOR) before being passed.

Wrapper classes are also used for Java and Python. However, the implementation uses the particular C native interface, *i. e.*, JNI in the case of Java and C extension types in the case of Python. FORTRAN 77 and Fortran 90/95 bindings are technically challenging as there is neither a native C interface nor a well-defined binary representation. Instead, it depends on the particular compiler how data structures are laid out in memory. Babel uses a stripped-down version of CHASM [37] to handle compiler peculiarities and initialize and convert complex types such as arrays for Fortran 90/95 and 2003/2008.

Stubs are also used by Babel in order to expose built-in SIDL features, such as reference counting and up-/down-casting, in a way that integrates nicely with the native language ecosystem. For example, wrapper classes for C++ and Python implement smart pointer semantics freeing the user from error-prone explicit reference counting.

B. Intermediate Object Representation (IOR)

The IOR is a well-defined intermediate representation for each of the fundamental SIDL types listed in Table II. For basic algebraic types, Babel uses the corresponding equivalent in C (*e. g.*, the SIDL type `int` is represented by the C99 type `int32_t`). Likewise, basic types such as `bool` or `string` are mapped to their C equivalents. Complex numbers are pairs of single or double precision floating point numbers.

Arrays are more interesting as they are defined for a particular base type. They are represented by a more complex data structure containing meta information and a pointer to the allocated memory. Meta information includes lower and upper bounds, strides, a reference count, and a simple dispatch table for a small number of support routines. Structs are recursively defined on top of these types and may contain arrays, other structs, or interfaces/classes. However, there is no support for arrays of structs or arrays of arrays.

By far the most interesting types are *classes* and *interfaces*. In Babel jargon, each object or interface carries a reference to an entry point vector (EPV) that defines the set of member functions supported by the corresponding type. Figure 8 shows the memory layout for a Babel object with a simple inheritance structure. Solid lines denote generalization while dashed lines stand for implementation of interfaces.

Several things are important to note. First, the memory layout is such that the beginning of an Object of a derived type is always also the be-

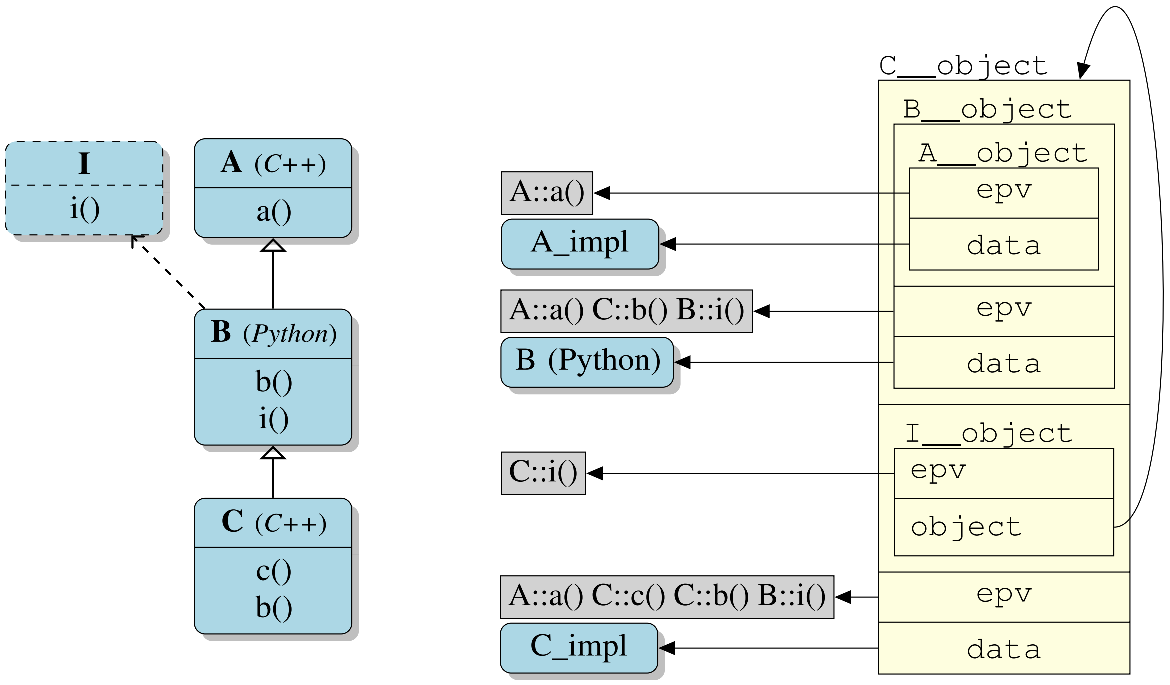

Fig. 8. Babel object layout for a simple inheritance structure.

ginning of a valid object of its base class. Thus, up- and down-casting can be implemented without pointer adjustments. Casting to an interface, however, will usually return a base address different from the original object. The only exception is `sidl.BaseInterface`, which is treated differently.

Next, for any object of a given class, the entries of the entry point vector are exactly the same. Consequently, dispatch tables are initialized only once and require only a constant amount of memory. Each object, however, carries a reference to the entry point vector of its dynamic type.

Furthermore, there is a private data pointer in each object that can be used to hold state. The meaning of this pointer depends on the particular implementation language. In simple cases such as C or Fortran, it is just an opaque data pointer provided to the user. For higher-level languages such as C++, it is usually a reference to an object of a user-modified class.

In reality, objects optionally carry some additional state for profiling, contract enforcement, or "hooks", which are user-supplied methods invoked before and after a particular method invocation. The example also omits implicit base classes and interfaces for simplicity.

Each EPV contains a set of built-ins provided by Babel in addition to user-defined methods. Built-ins always start with an underscore and provide casting, reference counting, and support routines for object construction and destruction. The user invokes the static `_create` built-in in order to create new SIDL objects. This will dynamically allocate memory for the new objects and initiate recursive initialization of the EPV for each class in the inheritance hierarchy. Babel will call a user accessible constructor (`_ctor`) once basic object initialization is complete. Object destruction is implicit once the reference count goes to zero. Again, a destructor (`_dtor`) is provided to the user for implementation-specific de-allocation.

In the presence of contract clauses, the Babel compiler adds check routines to the IOR to support enforcement. Individual assertions within a contract clause results in the generation of a corresponding check in the routine. Preconditions, if any, are grouped together under a single (compound) if-statement to ensure they are only executed when allowed by the current policy. Any invariants are grouped in the same manner before the call to the original skeleton method. Finally, postcondition and invariant checks are generated. Hence, all specified contracts are translated into enforcement checks within the new routines, with enforcement decisions being made based on runtime options.

A key challenge to providing efficient runtime contract enforcement is minimizing performance overhead. This is accomplished in part by the generation of a second function pointer table. While the primary *epv* table contains pointers to functions defined in the *skeleton* layer. The second, or contracts, *epv* table contains pointers to the corresponding check routines.

C. Skeletons

Skeletons are the counterpart to stubs. They convert the IOR to the particular native language representation and transfer control over to the user-supplied member function implementation. For return values and out arguments, the inverse operations are applied, *i. e.*, they are converted from their native language representation to Babel's IOR. Most technical considerations discussed in the context of stubs also apply to skeletons. Except for C++ and Fortran 2003/2008, they are implemented in C and use some form of native language interface for argument conversion.

It is important to note that implementations of SIDL objects can be stateful. This is achieved via the data pointer shown in Figure 8. For procedural languages such as C or Fortran, this pointer is exposed to the user and can be used to hold a private data structure. However, for object oriented languages such as C++, Java, or Python, the user implements a regular class that may contain private data members. In these cases, Babel implicitly manages a reference to an object of the user-provided type using the data pointer. The skeleton is responsible to cast this reference to the appropriate type and invoke the implementation within its context.

Another Babel feature mainly implemented in skeletons are so-called *hooks*. Hooks provide a simple form of aspect oriented programming and allow the user to execute code right before or after method invocations. They can be dynamically enabled or disabled. Hooks are useful for a variety of applications such as logging or profiling. The skeleton executes pre-hooks immediately prior to the usual method dispatch. Likewise, if no exceptions are encountered, post-hooks are invoked immediately after returning from an implementation.

D. Implementations

Apart from glue code generation, Babel provides assistance to develop and maintain implementations for SIDL specifications. This includes the generation of Makefile templates as well as a set of implementation files, which are the only Babel generated files the user is expected to modify. Implementation files serve as a starting point for developers and contain all the necessary declarations and prototypes.

Keeping implementations up to date with evolving interface specifications can be a challenging problem. Babel provides some assistance therefore via so-called *splicer blocks*. Splicer blocks are structured comments that mark the begin and end of user-modifiable sections within a file. Babel will preserve these sections across repeated invocations. All changes outside these splicer blocks may be lost. Babel automatically adds new methods and changes prototypes of existing methods as necessary.

E. Special-Purpose Backends

Babel supports an alternative tool-friendly XML representation for SIDL interfaces. There is both a XML front- and back-end. This means that XML can be used as input language for Babel. Babel can also be used to convert generic SIDL files into XML specifications. XML files retain references to the original SIDL file such as line number information. Both formats are largely equivalent and have uniform support for doc-comments.

SIDL interfaces can also be used to automatically generate documentation using a HTML backend similar to Javadoc [27]. Babel itself uses these capabilities to generate consistent documentation for its runtime library. User-provided doc-comments are properly maintained.

VI. Runtime Library

Babel's runtime library introduces some support for reflection, in particular runtime type information (RTTI). This allows the programmer to determine the dynamic type and SIDL version of a given object. Types can be identified by name or via a class info data structure.

Reference counting is implemented using atomic compare-and-swap operations instead of global locks. Babel uses non-standard compiler intrinsics if available and small chunks of inline assembler otherwise.

There is also extensive support for dynamic loading and symbol resolution. Symbols can be resolved ahead of time or as needed (lazy). The SIDL runtime systems manages a library search path and keeps track of all libraries loaded through its interface. Dynamic SIDL libraries are identified by a SIDL class file (SCL). These files contain meta data for an arbitrary number of dynamic libraries, allowing the loader to locate and identify them as needed. There is optional support for `md5` and `sha1` message digests to verify that libraries have not been modified or replaced.

A. Remote Method Invocation

Remote method invocation [31] in Babel is fully transparent to the user. This means that the user's code stays exactly the same, no matter if an object is local or remote. Babel provides the built-ins `_isRemote` and `_isLocal` to distinguish among the two cases.

A reference to a remote object can either be obtained by creating a new remote object via `_createRemote` or by connecting a stub to an existing remote object via `_connect`. Remote object stubs differ from local objects mainly in that their EPV will not directly point to the skeleton, but to a Babel-generated function that marshals and un-marshals arguments and performs the necessary network transfer. All objects are identified by a protocol-specific URL. Babel itself ships with a simple TCP/IP based protocol. However, users are expected and encouraged to provide their own implementations to make use of machine specific high-speed interconnects or particular communication patterns [30], [31]. The same mechanism can be used to provide compatibility with related systems such as CORBA.

Objects can be passed remotely either by reference or by copy. A call by reference requires the existence of a so-called Babel object server (BOS). Call by value, on the other hand, requires an object to be serializable and creates an additional *local* copy on the remote end. The user is responsible to implement (de)serialization functionality for a particular class by implementing the `sidl.Serializable` interface. For derived types such as structs, Babel provides automatically generated support routines.

Distributed reference counting can be a challenging problem. Babel keeps partial reference counts in the local stubs. Only when the local reference count reaches zero, communication occurs to adjust the reference count on the object itself. This approach has many advantages since it significantly reduces the amount of network traffic. However, it may also lead to resource leakage in case clients are physically disconnected. As for communication protocols, more fault-tolerant schemes can be provided by the user if necessary.

B. Contract Enforcement

The ability to enforce interface contracts, described in Section IV-F, at runtime requires library support. Minimally, there are contract-specific exceptions raised when clauses are violated. Support for a variety of enforcement options across programming languages also requires option management. Finally, although earlier work pursued distributed enforcement decision processes [15], [16], the goal of trying to better control enforcement overhead lead to the current release containing a centralized enforcement manager [13].

There are three interface contract exceptions supported in the runtime library, one per contract clause. Violations in the assertions within a clause result in a clause-specific exception being raised at runtime. For example, an assertion evaluating to false within a precondition clause results in a `sidl.PreViolation` exception. The contracts are checked and exceptions raised automatically by the generated middleware. The client side needs to be aware of and appropriately handle these exceptions.

Contract Classification[4]	**Enforcement Frequency**
All	*Adaptive Fit*
Constant	*Adaptive Timing*
Invariants	*Always*
Linear	*Never*
Method Calls	*Periodic*
Postconditions	*Random*
Preconditions	
Results	
Simple Expressions	

TABLE III
BASE CONTRACT ENFORCEMENT OPTIONS

Babel supports traditional and experimental contract enforcement through runtime options combining the classification of contract clauses to be enforced with the frequency of their enforcement. This approach was considered to be the most flexible for not only enforcement but also classification purposes [14]. Table III lists the more common options for each criteria.

Contract clause classification options are based on either the type of clause or clause contents. Traditional enforcement options are either all-or-nothing for precondition, postcondition and/or invariant clauses. Babel supports those as well as options based on clause contents (e.g., presence or absence of method calls, output and return arguments, or the complexity of assertions). Determination of the complexity of actual contract clauses is inferred by the Babel compiler based on the complexity of array arguments in the corresponding assertions. The clause is tagged according to the highest complexity encountered.

Frequency options further restrict enforcement of the clauses satisfying the classification option. The traditional all-or-nothing approach is supported through the *Always* and *Never* options, respectively. Simple sampling strategies are reflected in *Periodic* and *Random* sampling, where the period (or number) is specified at runtime. The final options focus on performance-driven adaptive enforcement with the goal of reducing the contract enforcement overhead. The two basic adaptive strategies are: *Adaptive Fit* and *Adaptive Timing*. Both rely on execution time estimates for contract clauses and methods. *Adaptive Fit* checks clauses whose execution time will not result in exceeding the desired overhead limit when compared to the execution time of the method. *Adaptive Timing*, on the other hand, factors in the cumulative estimates of time spent in methods and contract checking, enforcing contracts so long as the clause does not result in exceeding the desired limit.

[4]Additional options, specified in `sidl.sidl`, enable enforcement of two of the three clause types or clauses with higher complexity assertions.

Enforcement decisions are actually made by an *Enforcer* class based on information about the clause under consideration and the enforcement policy options in affect. If an assertion within a clause is determined to be violated, then the appropriate exception is raised. Statistics on enforcement decisions and violations are maintained. This is all handled automatically by the IOR making the necessary calls to the *Enforcer* class at the appropriate time.

Babel supports a variety of traditional and experimental contract (clause) enforcement options. Traditional all-or-nothing approaches are supplemented with selective and sampling-based enforcement techniques. Selective enforcement options can be used to gather data on the nature of clauses actually encountered during testing and, potentially, deployment. Sampling-based enforcement techniques are intended to enable reduced contract enforcement during deployment in performance-constrained environments.

VII. APPLICATIONS

Babel was born out of a larger approach to manage the rising software complexity of scientific applications. The Common Component Architecture (CCA) [3] is a joint effort by researchers from both academia and U.S. national laboratories to establish and adapt component technology for high-performance scientific computing. The CCA mediates how components interact with each other and with the underlying framework, using Babel as its language interoperability framework. Babel can be used stand-alone or as part of the full CCA framework.

The CCA specification itself is written solely in SIDL. Thus, each of the languages supported by Babel is a first class citizen and can be combined with components written in any other language. Components interact with each other via *ports*, following the provides/uses design pattern. Each component specifies which ports it uses from others and for which it provides an implementation. Several CCA compliant frameworks focusing on different programming models are currently available. Ccaffeine [2] focuses on SPMD-style programming and supports a native C++ interface; XCAT [22], [41] focuses on distributed programs; SCIRun [36], [43] features bridging technologies between CCA, CORBA, VTK, and shared-memory models. Babel itself ships with a simple reference implementation called Decaf [28].

Asynchronous Babel RMI has been used to develop COOP - a new parallel MPMD (multiple program multiple data) programming model that allows applications to distribute work in a flexible way. Thus, COOP effectively allows applications to scale to larger and larger machines. COOP has been successfully used in several high-performance applications (*e. g.*, in material science [25], [6]).

Another very common use case for Babel is to provide bindings to scientific libraries for various languages. One example therefore is *hypre* – a suite of scalable parallel linear solvers and preconditioners for sparse linear equation systems [20]. The user is not even necessarily aware that he is using Babel. Language independent SIDL specifications have also proven to be a valuable tool for stable interfaces. Furthermore, hypre developers were able to consolidate multiple versions of algorithms that only differed in implementation details for matrix multiplication using polymorphic interfaces. This shows that Babel can be used to provide benefits of object oriented design while avoiding portability problems of real object oriented languages such as C++.

Experiments for course-grained interface descriptions [26] clearly show that the overhead of Babel is well within measurement imprecision and usually below 1%. A more detailed performance study [5] shows that the call overhead is usually very small for most data types. Large overheads can only be observed for a small number of types such as strings or arrays with explicit order specifications, as Babel is forced to allocate new memory and physically copy data.

While some overhead is often unavoidable, converting to and from the IOR is unnecessary if caller and callee are implemented in the same language. Recent versions of Babel implement an experimental feature that allows us to bypass the IOR in this special case. Experiments for C++ show that the costs for native Babel calls can be reduced to roughly the costs of native virtual function calls, effectively eliminating the overhead for native method invocations.

Babel and/or the CCA has been successfully used in a large number of projects in various areas such as chemistry, geomagnetics, sparse linear algebra, fusion, or nuclear plant simulation. Discussing these projects is beyond the scope of this paper. A recent overview can be found elsewhere [29].

VIII. Related Work

Babel is an Interface Definition Language (IDL)-based tool similar to industry's CORBA/CCM [35], [10], Microsoft's (D)COM [7] and .Net [33], Mozilla's XPCOM [42], and Sun's JavaBeans [40]. There are several existing projects providing language interoperability among a limited set of languages – the most important are shortly discussed in the following. In many respects, the design considerations are very different from the requirements of the scientific computing community. The main differences and limitations of each approach will be discussed individually.

SWIG [4] is a software development tool that connects C and C++ libraries to a large set of scripting languages (*e. g.*, Perl, Python, PHP, Ruby, or Tcl). Recent versions also include bindings for non-scripting languages such as C#, Lisp, or Java.

As most approaches, SWIG lacks the support for Fortran and scientific data types. Its primary purpose is to provide language-specific bindings for C/C++ libraries. Thus, there is a strong asymmetry between the native implementation and the supported client languages. SWIG uses so-called "interface files" that contain C/C++ style declarations as its input. SWIG can also be used to package structures and classes into proxy classes for the particular target language, exposing data structures in a more direct way.

CORBA [35] is a distributed object specification maintained and developed by the Object Management Group (OMG) – a consortium of several hundred industry partners. Like Babel, CORBA uses an interface definition language (IDL) to specify the interface exposed to the outside. It supports interaction among objects written in different languages with a large set of supported bindings.

CORBA mainly targets distributed systems communicating over a physical network. Communication is managed by a so-called Object Request Broker (ORB). An abstract protocol (GIOP) is used to communicate among different ORBs. Implementations exists on top of plain TCP/IP as well as higher-level protocols such as SSL or HTTP.

As all the remaining approaches, CORBA lacks essential features for high-performance computing such as Fortran-style arrays, strides, or complex numbers. While it provides robust interoperability among various languages, it is usually far too inefficient for performance sensitive in-process method invocations [5]. CORBA also provides a rather limited object model with no support for polymorphism.

Microsoft's Component Object Model (**COM**) and its distributed version DCOM [38] are Windows-based interoperability frameworks. Their main focus is on business and internet applications.

For various reasons, (D)COM is hardly a candidate for high-performance computing. One major problem is portability to platforms other than Windows. It also lacks the necessary abstractions for parallel data organization and scientific data types. Furthermore, COM uses a very limited object model with no support for polymorphism.

XPCOM [42] is a similar approach used by Mozilla to connect JavaScript and C++ components for their products, most notably the Firefox web browser. Because of its high overhead for data marshaling, due to the selection of supported languages, it is also not well suitable for high-performance computing.

Sun's **JavaBeans** and Enterprise JavaBeans (EJB) are Java-specific cross-platform architectures. It does not address the problem of language interoperability. In a limited way, the Java Native Interface (JNI) [39] is a way to integrate C and C++. However, the user has to deal explicitly with argument conversion and acquire and release global and local references for garbage collection support.

Protocol Buffers [21] are Google's approach to object serialization and is used for remote procedure calls in Googles internal software. Protocols are specified in a file format similar to SIDL and a compiler automatically generates the (de-)serialization code. They do not support Fortran and also do not include an RPC mechanism. **Apache Thrift** [1] is a very similar product developed at Facebook. It includes an RPC mechanism but also lacks support for Fortran. The main difference to Babel is that Babel is highly optimized for in-process language interoperability, where code in another language is directely called without having to serialize the arguments to transmit them over a network. Both formats are, however, very interesting alternatives for the Babel RMI protocol.

CHASM [37] is a research effort to automatically create components from existing Fortran modules. CHASM generates adapter classes that mediate among the two languages. Babel is a more general and flexible tool but builds on a stripped-down versions of CHASM for some language backends.

PIDL [17] is an experimental extension to SIDL that can describe interactions between distributed parallel components.

IX. Summary and Future Work

Babel is an open source scientific language interoperability toolkit, distributed under the LGPL license, with open bug tracking and version control systems. The latest stable Babel release is version 2.0. Babel is written in Java using the JavaCC parser generator for the SIDL frontend and Xerces for XML parsing. The build and configure system is based on autoconf, automake, and libtool.

Babel is under active development and serves as testbed for a number of exciting research projects. We actively work on reducing the overhead of language interoperability to allow for more fine grained interfaces. More challenging is support for emerging parallel PGAS (partitioned global address space) languages such as Chapel [8], Unified Parallel C (UPC) [18], or Co-array Fortran [34]. We are also considering support for GPGPU-oriented languages such as CUDA and OpenCL.

Acknowledgments

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. In particular, this work has been supported by the program for Scientific Discovery through Advanced Computing (SciDAC).

REFERENCES

[1] Aditya Agarwal, Mark Slee, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, April 2007.

[2] Benjamin A. Allan and Robert Armstrong. The Ccaffeine framework: Composing and debugging applications interactively and running them statically. Compframe, 2005. Extended Abstract.

[3] Rob Armstrong, Gary Kumfert, Lois Curfman McInnes, Steven Parker, Ben Allan, Matt Sottile, Thomas Epperly, and Tamara Dahlgren. The CCA component model for high-performance computing. *International Journal of Concurrency and Computing: Practice and Experience*, 18(2), 2006.

[4] David M. Beazley and Peter S. Lomdahl. Building flexible large-scale scientific computing applications with scripting languages. In *Proceedings of the 8th SIAM Conf. on Parallel Processing for Scientific Computing (8th PPSC'97)*, Minneapolis, Minnesota, USA, March 1997. SIAM (Philadelphia).

[5] David E. Bernholdt, Wael R. Elwasif, James A. Kohl, and Thomas G. W. Epperly. A component architecture for high-performance computing. In *Proceedings of the Workshop on Performance Optimization via High-Level Languages (POHLL-02)*, New York, NY, June 2002.

[6] Joel V. Bernier, Nathan R. Barton, and Jaroslaw Knap. Polycrystal plasticity based predictions of strain localization in metal forming. *Journal of Engineering Materials and Technology*, 130(2):021020, 2008.

[7] Nat Brown and Charlie Kindel. Distributed component object model protocol-DCOM/1.0. Technical report, Microsoft Corporation, Redmond, WA, November 1996.

[8] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel language. *IJHPCA*, 21(3):291–312, 2007.

[9] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (WSDL) 1.1. Technical report, World Wide Web Consortium, March 2001. http://www.w3.org/TR/wsdl.

[10] CORBA website. http://www.corba.org.

[11] O.J. Dahl and B. Myhrhaug. Simula implementation guide. Technical Report Publication S47, Norwegian Computing Center, Oslo, Norway, March 1973.

[12] Tamara Dahlgren, Thomas Epperly, Gary Kumfert, and James Leek. *Babel User's Guide*. Lawrence Livermore National Laboratory, July 2006. version 1.0.0.

[13] Tamara L. Dahlgren. Performance-driven interface contract enforcement for scientific components. In *Component-Based Software Engineering*, volume 4608 of *Lecture Notes in Computer Science*, pages 157–172, Berlin/Heidelberg, 2007. Springer.

[14] Tamara L. Dahlgren. *Performance-Driven Interface Contract Enforcement for Scientific Components*. PhD thesis, University of California, Davis, June 2008.

[15] Tamara L. Dahlgren and Premkumar T. Devanbu. Adaptable assertion checking for scientific software components. In *Proceedings of the Workshop on Software Engineering for High Per formance Computing System Applications*, pages 64–69, Edinburgh, Scotland, May 24, 2004.

[16] Tamara L. Dahlgren and Premkumar T. Devanbu. Improving scientific software component quality through assertions. In *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, pages 73–77, St. Louis, Missouri, May 2005.

[17] Kostadin Damevski and Steven G. Parker. M x N data redistribution through parallel remote method invocation. *IJHPCA*, 19(4):389–398, 2005.

[18] Tarek A. El-Ghazawi and Lauren Smith. UPC - UPC: unified parallel C. In *SC*, page 27. ACM Press, 2006.

[19] M. Ellis and B. Stroustrop. *The Annotated C^{++} Reference Manual*. Addison-Wesley, 1990.

[20] Robert D. Falgout and Ulrike Meier Yang. hypre: A library of high performance preconditioners. In *Preconditioners, Lecture Notes in Computer Science*, pages 632–641, 2002.

[21] Google. Protocol Buffers – Google's Data Interchange format. Documentation and open source release. http://code.google.com/apis/protocolbuffers/.

[22] Madhusudhan Govindaraju, Michael R. Head, and Kenneth Chiu. XCAT-C++: Design and performance of a distributed CCA framework. In *The 12th Annual IEEE International Conference on High Performance Computing (HiPC)*, pages 270–279, Goa, India, December 2005.

[23] Arnie Heller. Babel speeds communication among programming languages. *Science and Technology Review*, pages 9–10, October 2006. Available at https://www.llnl.gov/str/Oct06/pdfs/10_06.3.pdf. Last visited April 2010.

[24] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python. http://www.scipy.org/, 2001–.

[25] J. Knap, N. R. Barton, R. D. Hornung, A. Arsenlis, R. Becker, and D. R. Jefferson. Adaptive sampling in hierarchical simulation. *International Journal for Numerical Methods in Engineering*, 76(4):572–600, 2008.

[26] Scott R. Kohn, Gary Kumfert, Jeffrey F. Painter, and Calvin J. Ribbens. Divorcing language dependencies from a scientific software library. In *PPSC*, 2001.

[27] Douglas Kramer. API documentation from source code comments: A case study of javadoc. In *Proceedings of the 7th Annual International Conference of Computer Documentation (SIGDOC-99)*, pages 147–153, N.Y., September 12–14 1999. ACM Press.

[28] G Kumfert. Understanding the CCA standard through Decaf, 2007.

[29] G. Kumfert, D. E. Bernholdt, T. G. W. Epperly, J. A. Kohl, L. C. McInnes, S. Parker, and J. Ray. How the common component architecture advances computational science. In Wiliam M. Tang et. al., editor, *SciDAC 2006, Scientific Discovery through Advanced Computing*, volume 46 of *J. of Phys.: Conf. Ser.*, pages 479–493, Denver, Colorado, USA, June 2006.

[30] Gary Kumfert and James Leek. Writing a protocol to support RMI. Technical Report UCRL-TR-220292, Lawrence Livermore National Laboratory, February 2006.

[31] Gary Kumfert, James Leek, and Thomas Epperly. Babel remote method invocation. In D. K. Panda, editor, *Proc. 21st Int'l Par. Dist. Proc. Symp. (IPDPS'07)*. IEEE Computer Society, March 2007.

[32] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Upper Saddle River, NJ, 1997. Second Edition.

[33] Microsoft Corporation. .NET homepage. http://www.microsoft.com/net.

[34] Robert W. Numrich and John Reid. Co-Array Fortran for parallel programming. *ACM SIGPLAN FORTRAN Forum*, 17(2):1–31, August 1998.

[35] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, February 1998.

[36] S. G. Parker and C. R. Johnson. SCIRun: A scientific programming environment for computational steering. In *Supercomputing '95*. IEEE Press, 1995.

[37] Craig Edward Rasmussen, Matthew J. Sottile, Sameer Shende, and Allen D. Malony. Bridging the language gap in scientific computing: the chasm approach. *Concurrency and Computation: Practice and Experience*, 18(2):151–162, 2006.

[38] Roger Sessions. *COM and DCOM: Microsoft's vision for distributed objects*. Wiley and Sons, 1998.

[39] *Java Native Interface*, 1997. Javasoft's Native Interface for Java.

[40] Sun Microsystems. Enterprise JavaBeans downloads and specifications. http://java.sun.com/products/ejb/docs.html, 2004.

[41] Xcat project. http://www.extreme.indiana.edu/xcat.

[42] Xpcom Website. https://developer.mozilla.org/en/XPCOM.

[43] Keming Zhang, Kostadin Damevski, Venkatanand Venkatachalapathy, and Steven Parker. Scirun2: A CCA framework for high performance computing. In *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, pages 72–79. IEEE Computer Society, April 2004.

Author Biographies

Thomas Epperly, Ph. D., is the Project Leader for the Components Project in the Center for Applied Scientific Computing (CASC) at the Lawrence Livermore National Laboratory (LLNL). His research interests focus on language interoperability and component technology to facilitate large-scale computational modeling of physical systems. At LLNL, he has worked as co-architect of Babel, the R&D 100 award winning language interoperability technology, and as the Computer Science Group Leader. Dr. Epperly, a Department of Energy Computational Science Graduate Fellow, earned his Ph. D. in Chemical Engineering from the University of Wisconsin and his B. S. in Chemical Engineering from Carnegie-Mellon University. His Ph. D. research focused on parallel global optimization algorithms for nonconvex nonlinear programs. As a post-doc in the Centre for Process Systems Engineering at Imperial College in London, he worked on global optimization algorithms for reactor design under uncertainty. His undergraduate research project contributed to ASCEND, an object-oriented language for building large mathematical models. After his post-doc, he worked for Aspen Technology, Inc. on an equation-oriented modeling framework for real time optimization of chemical plant operations.

Gary Kumfert, Ph. D., worked on Babel during his tenure at Lawrence Livermore National Lab from Aug 1999 through Oct 2008. During that time, he conducted research in component technology, parallel model coupling, parallel and distributed programming models, and massively scalable partitioning algorithms. He joined Lawrence Livermore immediately after getting his Ph. D. from Old Dominion University where he did research in combinatoric graph algorithms for sparse matrix computations and object-oriented computing. Currently, Gary is an engineering manager at Conviva Inc., where he works with major broadcasters to optimize their online video ecosystem.

Tamara Dahlgren, Ph. D., is a Computer Scientist at Lawrence Livermore National Laboratory (LLNL). She has worked in the field for more than twenty years on a wide variety of projects, two of which have earned R&D 100 awards. Her research interests focus on software quality improvement. Tamara earned her Ph. D. from the University of California at Davis in 2008 (while employed full-time at LLNL). Her dissertation work included extending Babel to support the specification of and numerous enforcement strategies for interface contracts. Currently, Tamara's time is split between multi-project software quality services and the Components project.

Dietmar Ebner, Ph. D., was a postdoctoral researcher on the Babel team working on performance improvements and support for structs and modern Fortran dialects. He earned his Ph. D. from the Vienna University of Technology for his research on retargetable optimizing compiler backends for superscalar and VLIW architectures. His research interests include applications of mathematical programming to hard combinatorial optimization problems, compiler optimizations and analyses, and large-scale machine learning. Currently, Dietmar Ebner is a Software Engineer at Google Inc.

James Leek, M. S., worked on Babel from 2004 through 2006. James primarily developed the Babel Java bindings and Babel's Remote Method Invocation features. His primary research interest is large-scale parallel computing. James continues to work at Lawrence Livermore National Laboratory (LLNL) on a variety of computing projects including parallel discrete event simulation and parallelization, portability, and integration issues related to large scale computing. James joined LLNL immediately after earning his B. S. in Computer Science from the

University of California at Berkeley, and earned his M. S. at the University of California at Davis while working at LLNL.

Adrian Prantl, Ph. D., is a post-doc at the Center for Applied Scientific Computing (CASC) at the Lawrence Livermore National Laboratory (LLNL). He is working on Babel — a high-performance language interoperability framework. His research interests are in compiler construction, programming languages and code generation. Before coming to LLNL, he worked on static program analysis and verification of real-time systems. Adrian received a Ph. D. from Vienna University of Technology, Austria for his work on high-level compiler support for worst-case execution time analysis. His M. Sc. thesis was on code generation for a VLIW architecture. Currently, he working on a extending Babel to support the upcoming PGAS languages.

Scott Kohn, Ph. D., worked on the Babel language interoperability effort at Lawrence Livermore National Laboratory from 1999 through 2003. Scott started the Component Technologies project in 1999 and helped develop the core Babel architecture and the SIDL interoperability language. Scott has worked at LLNL since 1997, leading various projects in structured adaptive mesh refinement, component technologies, genomics analysis, graph analytics, and cyber security. From 2003 to 2004, Scott took a brief leave from LLNL to work as Director of Information Technology for OpGen, Inc., developing novel techniques for large-scale genomic analysis. Scott received his Ph. D.. in Computer Science from the University of California, San Diego, and his B. S. in Electrical Engineering, Math, and Computer Science from the University of Wisconsin, Madison. He currently leads a cyber security research effort at LLNL.