The Exu Approach to Safe, Transparent and Lightweight Interoperability

John F. Bubba and Alan Kaplan
Department of Computer Science
Clemson University
Clemson, SC 29634 USA
{jbubba/kaplan}@cs.clemson.edu

Jack C. Wileden
Department of Computer Science
University of Massachusetts
Amherst, MA 01003 USA
wileden@cs.umass.edu

Abstract

Exu is a new approach to automated support for safe, transparent and lightweight interoperability in multilanguage software systems. The approach is safe because it enforces appropriate type compatibility across language boundaries. It is transparent since it shields software developers from the details inherent in low-level language-based interoperability mechanisms. It is lightweight for developers because it eliminates tedious and error-prone coding (e.g., JNI) and lightweight at run-time since it does not unnecessarily incur the performance overhead of distributed, IDL-based approaches. The Exu approach exploits and extends the object-oriented concept of meta-object, encapsulating interoperability implementation in meta-classes so that developers can produce interoperating code by simply using meta-inheritance. In this paper, an example application of Exu to development of Java/C++ (i.e., multi-language) programs illustrates the safety and transparency advantages of the approach. Comparing the performance of the Java/C++ programs produced by Exu to the same set of programs developed using IDL-based approaches provides preliminary evidence of the performance advantages of Exu.

1. Introduction

Interest in interoperation among software components developed using two (or more) object-oriented programming languages is growing rapidly. The need to re-use reliable, well-understood software libraries and to interface with legacy systems are two common interoperability problems faced by software developers. Unfortunately, existing interoperability techniques have various shortcomings that preclude transparent interoperation between components defined in different object-oriented languages.

Most object-oriented languages provide only primitive interoperability mechanisms, which are generally based on functions and basic types defined by C. Although appealing from a performance perspective, these mechanisms do not provide direct interoperation between different object-oriented programming languages. They are also heavyweight for software developers, since they require lots of tedious and error-prone coding, and consequently they are far from transparent. In recent years, other solutions (e.g., CORBA, DCOM and .NET) have been pushed by the industrial community. While more directly supporting object-oriented interoperability, these approaches have other serious shortcomings. Although not as heavyweight as the primitive mechanisms, they generally do require the use of an additional type model, such as the CORBA IDL (interface description language), and hence are far from transparent to software developers. These approaches also tend to be heavyweight in terms of run-time performance, since they typically impose a distributed software architecture, even in non-distributed applications.

This paper describes Exu,[1] a technique that helps automate the construction of multilanguage object-oriented programs. Exu represents a middle ground between language- and IDL-based interoperability approaches. The overall goal of this technique is to enable safe, transparent and lightweight interoperability between classes and objects defined in different languages. Similarly to IDL-based approaches, Exu supports direct interoperation between classes; however, Exu does not require the use of an IDL. Exu is based on the extension and exploitation of the object-oriented concept of meta-object. Interoperability capabilities are encapsulated in meta-classes. If a class needs to interoperate with a language that is different from the language used to define the class, the class

[1] In some Afro-Caribbean traditions, Exu is the divine messenger, master of all languages, who acts as intermediary between humans and all divinities and between gods and gods.

can meta-inherit interoperability capabilities from a special meta-class. Exu generates a corresponding class interface in the target language, along with all the necessary interoperability code and configuration management scripts. The generated code uses a language-based, primitive mechanism as an underlying interoperability substrate, which is completely hidden from generated classes and client applications. Like language-based approaches, Exu does not incur the performance overhead of distributed, IDL-based approaches. Finally, Exu supports the reliable application of language-based interoperability mechanisms. Software developers are relieved from the burden of having to develop, understand and maintain code produced by typical language-based, primitive interoperability mechanisms.

The remainder of this paper is organized as follows. In Section 2, we provide an overview of meta-objects and meta-classes, and indicate their role in the Exu approach. We describe a prototype toolset in Section 3. This toolset supports various aspects of object-oriented interoperability between Java and C++. We next provide a preliminary performance evaluation of some representative Java-C++ applications in Section 4. Section 5 summarizes the current status of our work and discusses some future research directions.

2. Overview of Meta-objects

Starting with the advent of C++ in the mid-1980s and accelerating with the emergence of Java in the mid-1990s, object-oriented (OO) programming has rapidly become the dominant approach to software development. In an OO program, an *object* typically represents some component of a modeled system. Thus, for instance, an OO implementation of an autonomous vehicle controller might represent each sensor as an object.

In OO programming, a *class* is a template that is used to create objects. That is, a class determines the properties of all the objects it is used to create. Thus, for example, all the sensor objects in an OO implementation of an autonomous vehicle controller would be *instances* of a sensor class, and the properties used to represent sensors in the control software would be determined by the definition of that class. Figure 1 illustrates this situation. The class sensor defines the properties (attributes[2]) and behaviors (operations[3]) by which all sensors will be represented in the autonomous vehicle controller. The individual objects (instances of that class) will then be distinguished by the particular values of their attributes, and the possible behaviors of all the sensor objects will be those specified by the class. Because they are one level of abstraction above objects, and because they are

[2]Called data members in C++ or fields in Java.

[3]Called function members in C++ or methods in Java.

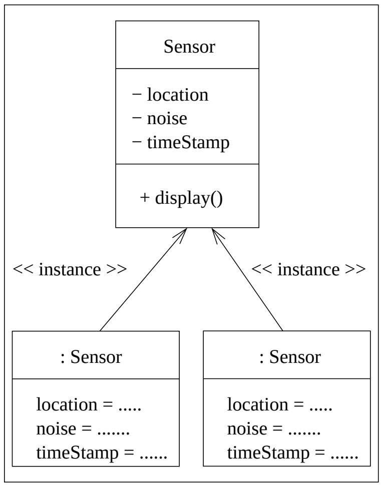


Figure 1. UML Diagram of a Sensor Class and Its Instances

repositories of meta-information about sets of related objects, classes are the most rudimentary form of *meta-object*.

The class meta-object provides some very powerful features to OO software. In particular, it means that there is a single place in which meta-information about an entire set of objects is maintained and through which it can be uniformly obtained or even modified. From the perspective of multilanguage programs, this means that classes provide an ideal means for describing meta-models, i.e., the meta-information needed for reasoning about, modifying, or achieving interoperation among components in a multi-language program. Thus, while most tasks in an OO application are appropriately carried out by manipulating objects, other tasks, notably those related to the overall structure of the application, fundamentally involve manipulation of meta-objects. Representing, examining or modifying properties related to the component interoperability in a multilanguage program are all examples of this latter category of tasks. Various alternative, *ad hoc* approaches to individual tasks in this category have been proposed (e.g., CORBA [11] or .NET [17] for integration or XML [4] for meta-modeling), but a meta-object-based approach has the potential to provide a unified basis for addressing all of them.

An important role of classes in OO programming is to represent relationships among, and to support combining of, properties of different sets of objects. The notion of *inheritance* is one mechanism used for this. Thus, for example, as shown in Figure 2, a class of radars might be a specialization of the sensor class. Instances of the radar class will have all the properties of the sensor class as well as whatever additional properties are needed to distinguish radars

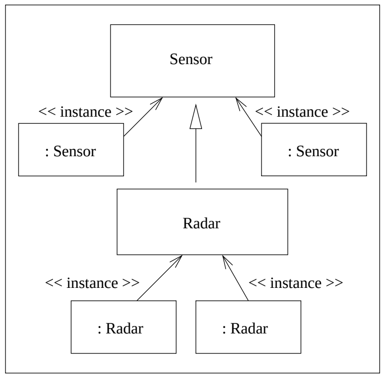


Figure 2. UML Diagam Showing Class Inheritance

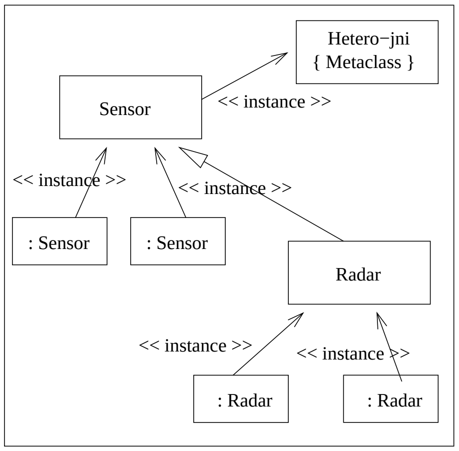


Figure 3. Determining Properties via a Meta-Class

in the application. Thus class-level inheritance provides one meta-object manipulation mechanism that provides a valuable, if limited, set of capabilities for multilanguage programming. Providing uniform interfaces for representing knowledge about shared components, for example, can be facilitated by this mechanism.

Introducing one additional level of abstraction, i.e., one additional level of meta-objects, to OO languages can significantly increase their power and potential value.[4] Treating classes as (meta)objects, which in turn are created using *meta-classes* (see Figure 3 for an example), means that there is a single place in which meta-information about an entire set of classes is maintained and through which it can

[4]While it is possible to continue introducing further levels of meta-objects, this seems to simply increase the complexity of the OO language without appreciably increasing its power.

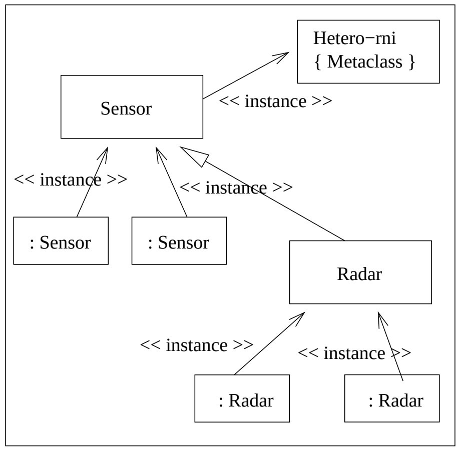


Figure 4. Changing Properties by Changing a Meta-Class

be uniformly obtained or even modified. Under this view, a meta-class determines the properties of all of the classes it is used to create and hence determines (some of) the properties of the (even larger) sets of objects created using all of those classes. So, in the Figure 3 example, the meta-class "Hetero-jni" would determine properties[5] of all the classes, and hence indirectly of all the objects, in the application software system. More importantly, substituting a meta-class "Hetero-rni"[6] for the "Hetero-jni" meta-class (see Figure 4) would alter the properties of all the classes, and hence indirectly of all the objects, in the system without requiring software developers to modify individual objects or even individual classes. Moreover, all the classes, and hence all the objects, inheriting from this meta-class would automatically present a uniform, consistent set of properties appropriate to the interoperation mechanism to which the meta-class corresponds.

3. Exu

The preceding section motivates the Exu approach, specifically how meta-objects are used to encapsulate interoperability details, which in turn enable classes to be reused transparently and safely in multilanguage programs. In this section, we describe a realization of Exu, which we have developed as an extension to the Open C++ Meta-Object Protocol (MOP) [6]. The current Exu prototype allows C++ classes to be accessed from Java applications using the Java Native Interface (JNI) [9] as the underlying in-

[5]In this case, that interoperation in the multilanguage program is to be by means of the Java Native Interface (JNI) [9].

[6]That is, specifying that interoperation in the multilanguage program is to be by means of the Microsoft Raw Native Interface (RNI) [10].

teroperability mechanism. Given a C++ class (or collection of classes), the Exu prototype automatically produces a corresponding Java proxy class (or collection of proxy classes) as well as the JNI-based Java-C++ interoperability code. A significant feature of Exu is that it does not require any changes to existing C++ classes. As a result, the same C++ class can be used in both C++ and Java applications.

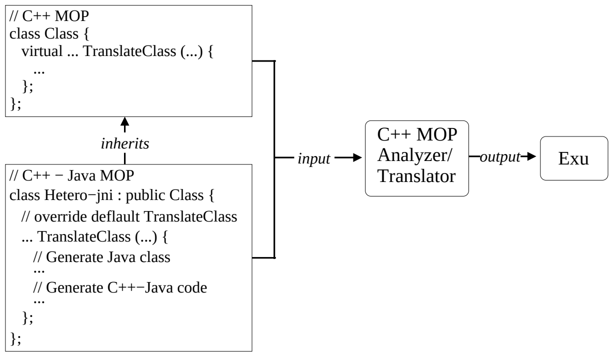
```
class Sensor {
  . . .
  public:
    Sensor ();
    virtual void reset ();
    virtual void setThreshold (int value);
    virtual int getThreshold ();
    virtual char∗ display ();
    virtual void noiseLevel (float∗ amount);
    virtual void copy (Sensor& source);
};
```

Figure 5. A C++ Sensor Class

To help illustrate the Exu prototype, and its relationship to both traditional, language-based approaches and contemporary approaches, consider the C++ Sensor class (interface) shown in Figure 5. Suppose a software developer would like to use this class inside a Java application. A traditional approach to this problem would involve manual application of a low-level interoperability mechanism, such as the JNI (which allows Java applications to invoke C functions). Although various informal idioms and guidelines can ease the developer's task (e.g., [7, 13, 14]), such approaches are extremely tedious and prone to error. Moreover, the resulting interoperability code is application specific, that is, it is customized to an individual class or application and cannot be re-used to support multilanguage interoperability needs for other C++ classes and/or applications.

Contemporary alternatives to traditional approaches are based on the use of intermediate languages (e.g., CORBA's IDL [11], Microsoft's MIDL [15], ODMG's ODL [5], and ILU's ISL [8]). Thus, a developer would need to create a separate specification of the C++ Sensor class using an intermediate language, and then retrofit the Sensor class into the code generated from the intermediate language. Not only is this process extremely cumbersome, but these approaches generally impose a distributed architecture, which may not be compatible with the architectural requirements of a multilanguage application.

The Open C++ Meta-Object Protocol (MOP) provides a framework that supports extending and/or changing the default semantics of various C++ constructs. The framework itself consists of a library of meta-classes that correspond to, among other things, the definition of C++ classes and methods. These meta-classes are processed by the Open C++ MOP analyzer, which produces a C++ translator extended with the behavior as specified in (Open C++ MOP) meta-classes. For example, the Open C++ MOP defines a meta-class called Class. This meta-class provides a method called TranslateClass, which is invoked (by a C++ translator produced by the C++ MOP analyzer) when a C++ class is recognized in an input source. By default, the TranslateClass method simply returns the source representing the C++ class. By extending the meta-class Class, developers can provide customized class translations.



```
// C++ MOP
class Class {
  virtual ... TranslateClass (...) {
    ...
  };
};
```

```
// C++ − Java MOP
class Hetero−jni : public Class {
  // override deflault TranslateClass
  ... TranslateClass (...) {
    // Generate Java class
    ...
    // Generate C++−Java code
    ...
  };
};
```

Figure 6. Generating the Exu Translator

Figure 6 shows how the Open C++ MOP toolset is used to create Exu. A meta-class, called Hetero-jni, specifies Java interoperability semantics for C++ classes. This meta-class is a subtype of the meta-class Class and is processed by the Open C++ MOP analyzer, which generates the Exu toolset. The Hetero-jni meta-class overrides the default definition of the TranslateClass method; thus, the Hetero-jni version of TranslateClass is invoked by the Exu toolset when a C++ class has been parsed and analyzed.

The meta-class Hetero-jni serves two primary functions. First, it encodes a model of *multilanguage class compatibility*. More specifically, this model is used to construct a Java *proxy class* that is compatible with a given C++ class. By compatible, we mean that the Java proxy class provides an interface that mirrors the class interface of its C++ counterpart. Moreover, the visible interface of the Java proxy class does not expose the fact that the proxy class is actually implemented by a C++ class.

The other primary function of the meta-class Hetero-jni is that it encodes details of the Java-C++ interoperability mechanism. In this case, the interoperability mechanism employed is based on the JNI and automates techniques similar to those described in [13, 14]. To employ a different Java-C++ interoperability mechanism, such as Microsoft's Raw Native Interface (RNI), we could simply substitute a different meta-class encoding the details associated with that mechanism (e.g., RNI).

To produce the actual Java proxy class and associated Java-C++ interoperability code for a given C++ class, a C++ class must meta-inherit from the Hetero-jni meta-class. Once this relationship is established, the Exu toolset can process a C++ class and automatically generate the appropriate Java proxy class and interoperability code.

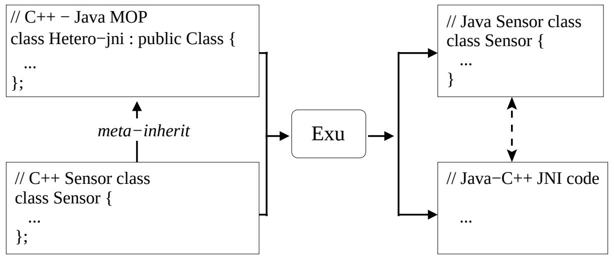


Figure 7. Using the Exu Translator

Figure 7 illustrates how the Exu toolset is used to allow the C++ Sensor class, shown in Figure 5, to be re-used in Java applications. The Sensor class meta-inherits from the Hetero-jni meta-class. As described above, this meta-class extends the default behavior of C++ classes with its model of Java-C++ class compatibility and Java-C++ interoperability. The Exu translator then processes the C++ Sensor class and the Hetero-jni meta-class, and produces a compatible Java Sensor proxy class. It also produces JNI-based Java-C++ interoperability code such that method calls (including constructors) are dispatched to the C++ implementation of the Sensor class.

Figure 8 shows the visible interface of the Java proxy class that is produced by Exu. (The details of the Java-C++ JNI-based interoperability mechanism, which are encoded in the private section of the Java proxy, are not shown.) Notice that the signatures of the methods in the Java Sensor class are nearly identical to those of the corresponding methods in the C++ Sensor class. In particular, primitive types in C++ can, in general, be mapped to primitive types in Java. In addition, class types appearing in C++ methods can be mapped to corresponding Java proxy class types. A notable exception concerns C++ pointers (or references) to primitive types, for which there are no corresponding types in Java. For example, the C++ method noiseLevel has a single parameter of type float*. Exu maps this type to a newly defined Java *wrapper* class called FloatWrapper (see Figure 8), which provides two accessor methods, i.e., get and set methods, the former of which declares a return type of float and the latter of which declares a single float parameter. Unfortunately, not all C++ primitive pointer/reference types can be mapped to Java wrapper classes. For example, as shown in Figure 8, the C++ char* type is more appropriately mapped to the Java StringBuffer class. As noted above, these mappings are automatically determined by the Exu cross language type compatibility model.

```
class Sensor extends JNIBase {
   // public interface
   public Sensor ( ) {...}
   public void reset () {...}
   void setThreshold (int value) {...}
   public int getThreshold () {...}
   public StringBuffer display () {...}
   public void noiseLevel (FloatWrapper amount) {...}
   public void copy (Sensor source) {...}
   // Java-C++ JNI code
   private ...
}
...
class FloatWrapper {
   public float get () {...}
   public void set (float value) {...}
   private float value;
}
```

Figure 8. Generated Java Proxy Class

4. Evaluation

The current Exu prototype has been developed and built as an extension to the Open C++ MOP 2.10 [6] and supports the development of Java-C++ applications, where Java applications may interoperate with C++ classes. Exu generates interoperability code based on the JNI, which is part of the JDK 1.3. The experiments described in this section were performed on a SPARC Ultra 10 with 128Mb of memory running under Solaris 2.7. Although the C++ code generated by Exu is in general independent of any particular compiler, our experiments were conducted using the Sun WorkShop C++ compiler.

The Exu prototype can generate compatible Java classes and JNI-based interoperability code for C++ classes that have the following properties:

- Single inheritance
- Primitive types passed or returned by value, pointer or reference (int, float, double, bool, char)
- C-style strings (char*)
- Class types passed or returned by value, pointer or reference
- Overloaded methods

- Virtual methods

Conventional language-based approaches to multilanguage software development are generally not type safe. The Java Native Interface and Raw Native Interface specifications are two primary examples of approaches supporting the development Java-C programs. Such approaches are not type safe since they require that developers ensure that types are used and mapped correctly across languages.

In comparison, Exu is a more type safe approach to multilanguage program development. The mapping of types between Java and C++ is handled entirely by the Exu toolset. In addition, Exu relieves developers the burden of having to hand craft multilanguage interoperability code.

Existing approaches to multilanguage program development are far from transparent. In contemporary approaches, the choice of a specific interoperability mechanism often becomes inextricably intertwined with the application. This makes it extremely difficult, if not infeasible, to change the interoperability mechanism at a later point during the application's lifetime. For example, replacing a JNI-based interoperability substrate for an application with an RNI-based one (or vice versa) generally requires a major rewrite of the application.

Similarly, IDL-based approaches (e.g., CORBA, DCOM, ILU and ODMG) are not very transparent. Developers must use an additional intermediate language (i.e., an IDL) to specify classes that are to be shared across language boundaries. This means that an additional "type" must be managed and maintained in order to interoperate with a class in multiple languages. In addition, these approaches impose a distributed software architecture, which is exposed in the various software modules that must be implemented.

In comparison, Exu neither requires an intermediate language nor does it impose a distributed software architecture. The interoperability code is generated directly from class specifications. As demonstrated in Section 3, the Exu toolset directly processes the C++ Sensor class and creates a corresponding Java (proxy) class. The actual implementation of the Java class (i.e., the C++ class and the Java-C++ interoperability code) is hidden from clients of the Java Sensor class.

The preceding comparisons indicate the safety and transparency advantages of Exu. We have also conducted several preliminary performance experiments involving the Exu approach. The primary objective of these experiments was to determine the overhead of invoking methods (including constructors) across programming language boundaries. These experiments compare a sample Exu-developed application to similar applications implemented using ILU [8] and CORBA [11]. We used ILU release 2.0beta1, which supports both distributed and JNI-based Java-C++ interoperability mechanisms. We also used the OOC Orbacus version 3.3 [12] implementation of CORBA. We note that the distributed programs (i.e., the server and client programs) executed as separate processes on a single host.

```
#include <math.h>
typedef double Number;
class fcomplex {
 public:
  fcomplex (Number r, Number i);
  fcomplex ();
  Number r ();
  Number i ();
  Number Cabs ();
  fcomplex Conjg ();
  fcomplex Csqrt ();
  fcomplex Cadd (fcomplex);
  fcomplex Csub (fcomplex);
  fcomplex Cmul (fcomplex);
  fcomplex Cdiv (fcomplex);
  fcomplex RCmul (Number);
 private:
  Number the_r;
  Number the_i;
};
```

Figure 9. A C++ Complex Number Class

Figure 9 shows the C++ interface for a complex number class, which we downloaded from a C++ class repository [3].[7] Using this class, we developed four applications, each of which has a Java driver that interacts with instances of the C++ class. Figure 10 shows the general structure of the Java driver, which creates several complex number objects and then performs various calculations on the objects. (The actual structure of the driver is slightly different depending on the interoperability approach used to implement the driver.)

In each experiment, we measured the time (in microseconds) required to execute the main driver.

Our first experiment consisted of a single run of the Java driver shown in Figure 10. This essentially provides "cold start" timing data for this sample application. Table 1 shows the results of our first experiment. The times shown represent the average of ten runs of each program. The overhead in calling the methods in the Exu-generated application is clearly orders of magnitude less than both ILU implementations and the CORBA implementation. This result can be directly attributed to the significant marshalling and demarshalling that is performed by the other three interoperability mechanisms. These results give us some confidence that

[7]The only modification we made to the source involved separating the class into its interface (".h") and implementation (".C").

```
a = random.nextInt() + random.nextDouble();
b = random.nextInt() + random.nextDouble();
c = random.nextInt() + random.nextDouble();
d = random.nextInt() + random.nextDouble();
x = new fcomplex(a,b);
y = new fcomplex(c,d);
z = x.Cadd (y);
z = x.Csub (y);
z = x.Cmul (y);
z = x.Cdiv (y);
a = z.Cabs ();
z2 = z.Conjg ();
z2 = z.Csqrt ();
z2 = z.RCmul (b);
z2 = new fcomplex(a,0);
```

Figure 10. Segment of Java Code

the overhead imposed in Exu is much less than for the other approaches.

Technique	Experiment 1	Experiment 2
EXU JNI	207.34	106.64
Orbacus	17399.59	3582.93
ILU JNI	7924.38	5367.16
ILU DIST	4838.65	2589.41

Table 1. Results of Experiments In Microseconds

Table 1 shows another set of results for a second set of experiments. In this set, we added a loop to the test driver. Each run of the experiment executed the Java driver ten times. We ran the second experiment ten times and normalized the result to a single execution of the test driver. The results for the second experiment show remarkable improvements in performance across all four applications. These gains can be attributed to the increased availability of opportunities for caching and optimization in the loop-based Java driver.

One interesting result of the two experiments was the extent to which the ILU distributed version outperformed the ILU JNI version. The ILU JNI version was developed using a technique that allowed two different languages, in this case C++ and Java, to share a common address space while still using ILU's distributed interoperability mechanism. Although the two languages are operating within a single address space, method invocations between the client and server are still being made through the ILU marshalling and demarshalling mechanism. We believe that the relatively poor performance of the ILU JNI version can be attributed to an additional level of abstraction that is not found in the distributed ILU implementation, but necessary for ILU JNI.

Not only is Exu lightweight from a performance perspective, as illustrated by these experiments, but from a developer's perspective the cost of using the other approaches is considerably greater than the cost of using Exu. For both CORBA and ILU, we were required to write not only intermediate type definitions corresponding to the complex number, but also implementations that tied together the existing C++ complex number class with the code generated from these type definitions. With Exu, however, we merely needed to invoke the Exu tool on the existing C++ complex number class.

5. Conclusion

This paper introduces Exu, a meta-object based approach to automated support for interoperability in multilanguage programs. An initial prototype implementation of the approach has been successfully applied to automating the use of C++ classes from Java programs via Java's JNI interoperability substrate. As demonstrated by the example usages presented in this paper, when compared to existing alternative approaches Exu appears to offer developers of multi-language programs superior safety, greater transparency and lighter weight. Encouraged by these preliminary results, we are pursuing a variety of activities aimed at further developing and demonstrating the potential of the Exu approach.

In the area of safety, we are actively investigating the difficult issues related to type compatibility in multilanguage systems, a topic in which we have been interested for over a decade ([18, 2]). We are, for instance, considering if and how to safely relax the restrictions listed at the beginning of Section 4, e.g., supporting non-virtual methods, multiple inheritance and aggregate types. A PhD dissertation currently in progress at UMass is aimed at extending formal foundations such as object-oriented type theory (e.g., [1]) and practical techniques such as type checking and type inference to provide a sound basis for the type compatibility across language boundaries.

In the area of transparency, we are working on improving and extending our current prototype implementation that automates use of C++ classes from Java programs via JNI, notably by augmenting it with the complementary ability to automate use of Java classes from C++ programs, again using JNI. To further demonstrate the transparency of the Exu approach, we are also developing the necessary meta-classes to enable both use of C++ classes from Java programs and use of Java classes from C++ programs via alternative underlying interoperability substrates, including Microsoft's RNI or .NET and OMG's CORBA. We anticipate that these meta-classes will enable even more convincing demonstrations of the significant transparency advantage of Exu, by supporting automated production of interoperable classes, using any of several underlying interoperability mechanisms, with virtually no effort by the software developer.

The improvements and extensions to Exu described above will also contribute to demonstrating how lightweight the approach can be. Beyond that, however, we are working toward specializations of the various meta-classes (submeta-classes) that can capture and exploit features of particular implementation environments. Thus, for example, characteristics of the specific hardware or software platform on which an application will be running can be taken into account when the (sub)meta-classes are developed and subsequent usages of those (sub)meta-classes will effectively optimize a set of interoperating classes for use on that particular platform, still with virtually no effort on the part of the application software developers.

Finally, we are actively pursuing more thorough evaluation of the Exu approach by applying it to larger, more realistic example applications. In collaboration with colleagues working in other research areas, we are currently starting to apply the meta-object based interoperability approach to integration of C++ and Java portions of mobile robot control software, to transparent interoperation of diverse client and server modules in multimedia networking and to increasing adaptability in computer aided design and analysis tools used in mechanical engineering [16].

6. Acknowledgments

The work performed at Clemson University work was supported in part by the ERC Program of the National Science Foundation under Award Number EEC-9731680. Alan Kaplan is now located at: Panasonic Technologies, Inc., 2 Research Way, 3^{rd} Floor, Princeton, NJ 08540 USA (kaplan@research.panasonic.com). John Bubba is now located at: DST Interactive, Inc., 9401 ArrowPoint Blvd., Suite 220, Charlotte, NC 28273 USA (jbubba@acm.org).

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, New York, NY, 1996.

[2] D. J. Barrett, A. Kaplan, and J. C. Wileden. Automated support for seamless interoperability in polylingual software systems. In *Fourth Symposium on the Foundations of Software Engineering*, San Francisco, CA, Oct. 1996.

[3] C. A. Bertulani. A class for manipulating complex numbers. http://quark.phy.bnl.gov/˜carlos.

[4] T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible markup language (XML) 1.0 specification. Technical report, World Wide Web Consortium, Cambridge, MA, Feb. 1998.

[5] R. Cattell, D. K. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez, editors. *The Object Database Standard: ODMG 3.0*. Series in Data Management Systems. Morgan Kaufmann, Jan. 2000.

[6] S. Chiba. A metaobject protocol for C++. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 285–299, Austin, TX, Oct. 1995.

[7] E. Gabrilovich and L. Finkelstein. JNI-C++ integration made easy. *C/C++ Users Journal*, pages 10–21, Jan. 2001.

[8] B. Janssen and M. Spreitzer. ILU: Inter-language unification via object modules. In *Workshop on Multi-Language Object Models*, Portland, OR, Aug. 1994. (in conjunction with OOPSLA'94).

[9] S. Liang. *The Java Native Interface*. Addison Wesley, 1999.

[10] Microsoft Corporation, Redmond, WA. *Raw Native Interface Reference*, 1999.

[11] Object Management Group, Framingham, MA. *The Common Object Request Broker: Architecture and Specification*, Feb. 2001. Editorial Revision 2.4.2.

[12] Object-Oriented Concepts, Inc., Billerica, MA. *Orbacus for C++ and Java*, version 3.3.3 edition, 2001.

[13] D. Parson and Z. Zhu. Java Native Interface idioms for C++ class hierarchies. *Software-Practice and Experience*, 30(15):1641–1660, Dec. 2000.

[14] A. Schade. Automatic bridging code generation for accessing C++ from Java. In *Technology of Object-Oriented Languages and Systems*, Melbourne, Australia, Nov. 1997.

[15] R. Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects*. Wiley Computer Publishing, Oct. 1997.

[16] S. Shanbhag, I. Grosse, A. Kaplan, and J. Wileden. A meta-object based approach to finite element modeling. In *ASME International Design Engineering Technical Conferences*, Pittsburgh, PA, Sept. 2001.

[17] D. Watkins. Handling language interoperability with the Microsoft .NET framework. Technical report, Microsoft Corporation, Oct. 2000.

[18] J. C. Wileden, A. L. Wolf, W. R. Rosenblatt, and P. L. Tarr. Specification level interoperability. *Communications of the ACM*, 34(5):73–87, May 1991.