Design of Large-Scale Polylingual Systems

Mark Grechanik, Don Batory, and Dewayne E. Perry
UT Center for Advanced Research In Software Engineering (UT ARISE)
University of Texas at Austin
Austin, Texas 78712
`{gmark, batory}@cs.utexas.edu, perry@ece.utexas.edu`

Abstract. Building systems from existing applications written in two or more languages is common practice. Such systems are *polylingual*. Polylingual systems are relatively easy to build when the number of APIs needed to achieve language interoperability is small. However, when the number of distinct APIs become large, maintaining and evolving polylingual systems becomes a notoriously difficult task.

In this paper, we present a simple, practical, and effective way to develop, maintain, and evolve large-scale polylingual systems. Our approach relies on recursive type systems whose instances can be manipulated by reflection. Foreign objects (i.e. objects that are not defined in a host programming language) are abstracted as graphs and path expressions are used for accessing and manipulating data. Path expressions are implemented by *type reification* — turning foreign type instances into first-class objects and enabling access to and manipulation of them in a host programming language. Doing this results in multiple benefits, including coding simplicity and uniformity that we demonstrate in a complex commercial project.

1 Introduction

Building software systems from existing applications is a well-accepted practice. Applications are often written in different languages and provide data in different formats. An example is a C++ application that parses an HTML-based web page, extracts data, and passes the data to an EJB program. We can view these applications in different ways. We can view them as COTS integration applications where a significant amount of code is required to effect that integration. Or we can view them as instances of architectural mismatch, specifically as mismatched assumptions about data models [1]. Or we can view them, as we do in this paper, as instances of *polylingual interoperable* [2] applications that manipulate data in *foreign type systems (FTSs)* i.e., type systems that are different from the host language.

Consider an architecture for polylingual systems as shown in the directed graph in Figure 1. Graph nodes correspond to programs P_1, P_2, …, P_n that are written in different languages and may run on different platforms. Each edge

$P_i \rightarrow P_j$ denotes the ability of program P_i to access objects of program P_j. $P_i \rightarrow P_j$ is usually implemented by a complex API that is specific to the language of the calling program P_i, the platform P_i runs on, and the language and platform P_j to which it connects. (In fact, there can be several different tools and APIs that allow P_i to access objects in P_j). Note that the APIs that allow P_i to access objects in P_j may be different than the APIs that allow P_j to access objects in P_i.
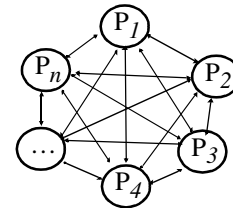

Figure 1: Architecture of Polylingual Systems

The complexity of a polylingual program is approximately the number of edges in Figure 1 that it uses. That is, when the number of edges (i.e., APIs needed for interoperability) is miniscule, the complexity of a polylingual system is manageable; it can be understood by a programmer. But as the number of edges increases, the ability of any single individual to understand all these different APIs and the system itself rapidly diminishes. In the case of clique of n nodes (Figure 1), the complexity of a polylingual system is $O(n^2)$. This is not scalable. Of course, it is hard to find actual systems that have clique architectures. In fact, people want them, but these systems are too complex to build, maintain, and evolve. A *large-scale polylingual system* is a polylingual system where the number of edges (APIs) is excessive. Such systems are common (we consider one in Section 6) and are notoriously difficult to develop, maintain, and evolve.

Current approaches do not support large-scale polylingual systems well. They are often limited to specific languages (e.g., typical CORBA platforms allow Java, C++, etc. programs to interoperate, but there are no facilities for accessing HTML or XML data or objects in C# programs). This leads to a proliferation in tools and their APIs used, which noticeably increases the accidental complexity [3] of the resulting code, loss of uniformity in the way programs are written,

thus rendering resulting systems extremely difficult to maintain and evolve.

Object-oriented researchers have developed frameworks as a technique for eliminating this kind of complexity. A *framework* is an abstraction that underlies a number of similar programs, and is represented by a set of abstract classes. A particular implementation of this framework is a set of concrete classes that customizes the framework's abstract classes for a designated application. The benefit of a framework is that it defines a single API that all programmers can use; so instead of having $O(n^2)$ possible APIs for achieving program-to-program communication, a single, standard, and clear API is used. New framework implementations are easy to add, and consequently, this is a scalable approach.

In this paper, we define a framework that presents a single API for the interoperability of programs in a polylingual system. Our idea is to abstract instances of an FTS as a graph of objects and to provide language-neutral specifications based on path expressions, coupled with a set of basic operations, for traversing this graph to access and manipulate its objects. We implement traversals by dynamically converting foreign types into first-class host language objects so that we enable access to and manipulation of their instances. This is the concept of *type reification*. Reification by reflection eliminates the need for generating potentially huge numbers of conversion classes, and allows us to access and manipulate semi-structured data that have no schemas. In this respect, this is superior to existing approaches because it does not require programmers to generate potentially large number of corresponding types, explicitly define common interfaces using an IDL language, or use different low-level APIs. Among its many benefits, we provide a type checking mechanism that enables a tractable method for checking program behavior with respect to reified foreign types. We call our approach the *Reification Object-Oriented Framework (ROOF)*.

2 Background

2.1 Principles of Polylingual Interoperability

A framework for the analysis and design of polylingual interoperable systems is given in [4][5]. This framework has two objectives. First, developers should not be constrained in using the type system provided by the host programming language. For example, if a programmer wants to share a Java object, then s/he should be able to do it directly in Java without resorting to some other type system such as an IDL. Second, the design of polylingual programs should not be affected by a decision to share objects. It means that the structure of classes and their interfaces should not be a function of sharing their instances.

Both these objectives define a foundational principle in design of polylingual system, called *seamlessness* [4]. This principle states that developers of polylingual systems need not be aware of language differences between interoperating programs. For example, polylingual programs that include special platform-dependent functions that facilitate interactions between the host and foreign type systems, are not seamless. Violations of this principle of seamlessness lead to complex and nonuniform code that is difficult to understand and reason about and subsequently difficult to maintain and evolve.

When analyzing and designing polylingual systems it is equally important to address three orthogonal requirements. The first is naming. Sharing objects among FTSs often requires elaborate name management mechanism. Suppose that an XML type's name is a keyword in a host programming language, for example, C++. Then this XML type name cannot be used directly when defining a corresponding class in C++. Even worse, if we have an XML and HTML schemas that have identical type names, then what is a naming scheme that resolves this ambiguity when defining corresponding types to C++? Serious effort is dedicated to offering effective name management strategies, however, most carry a significant overhead.

The other requirement is the time when a decision is made to share objects. The decision time is defined by three intervals: *earliest time* when a decision to share objects is made before any software is written, *common time* of making a sharing decision when only a part of polylingual system is developed, and a *megaprogramming* object sharing decision made after the entire system is developed. Clearly, allowing programmers to share objects at the megaprogramming stage is both attractive and quite difficult.

The final requirement is type checking of shared and native objects to ensure that they have compatible types. Different approaches that provide effective type checking can only be used at the earliest stage. Thus, the design and development of interoperable polylingual system is the science of trade-offs among objectives and concepts described above.

2.2 Related Work

Polylingual interoperability is a functional aspect that programmers should be able to add to or remove easily from existing software. If such changes are complex then polylingual software is hard to maintain and evolve. Low-level approaches (e.g. RPC, message passing, Document Object Model) provide APIs that enable programs to cross process boundaries in order to access foreign objects and invoke their methods. This approach is tedious and error prone because it requires the steep learning curve to master various vendor-

dependent APIs that deal, for example, with marshalling and unmarshaling data.

IDL-based approaches (DCOM, CORBA, Mockingbird) require programmers to define interfaces in an *Interface Definition Language (IDL)* that are implementation language neutral and can be translated into language-dependent client and server classes using an IDL compiler. This approach suffers from multiple drawbacks; notably the necessity to deal with an additional type system (the IDL), and to maintain client and server sets of code. In addition, this approach is hardly transparent since programmers are required to use a complex, hard-to-learn platform-dependent API.

IDL-based approaches are also difficult to maintain and evolve because reversing the initial decision to share objects once the client/server wrapper code is generated and implemented requires software to be rewritten. Suppose that a programmer creates a Java program whose objects are not shared by different programs. If this decision is reversed then these previously non-shared Java classes should be recoded as interfaces in an IDL, and then client/server code in implementation languages should be generated using an IDL compiler. Clearly, this can require major rework of the existing code that is laborious and costly when applied to large software projects. However, much larger amount of work is required if a Java class, conversely, is decided not to be interoperable after its IDL-based specification is created and client/server code is generated and implemented. This class has to be recoded by removing its IDL compiler generated code and writing its new implementation. To do this change requires significant programming investment, and is very expensive at the maintenance stage of a project.

When a programmer creates an interface using IDL s/he can select certain types to declare interface members because they may closely map to desired types in the selected implementation language. For example, if an IDL interface is used to generate C++ wrapper code then IDL types that define this interface are likely to be C++-friendly. When the same IDL interface is used to generate Java wrapper code, programmers may replace some IDL types with Java-friendly types. Mockingbird [6] is an IDL-based tool for developing polylingual distributed applications that generates adapter code that reconciles existing friendly IDL-based data types. However, this approach leads to software that is difficult to maintain and evolve. It also suffers from problems with IDL-based approaches described in [7].

PolySPIN [8] did away with the IDL approach by directly mapping types between different FTSs. A tool called PolySPINNER analyzes class definitions written in different languages, matches their structure, and generates code that enables objects of matched classes to interoperate seamlessly,

i.e. if objects of types t_1 and t_2 exist in different FTSs, for example, in Java and C++ correspondingly. Both types have to exist in Java and C++ to begin with. After applying PolySPIN approach a call to method f_1 of an object of type t_1 is translated by the generated code into the call to the matched method f_2 of some object of type t_2. The problem with this approach is that it requires complex matching mechanism to determine isomorphisms between foreign types.

Exu [9] is an alternative approach to IDL. It enables C++ classes to be accessed from Java classes using *Java Native Interface (JNI)*. For any C++ class Exu generates a corresponding Java proxy classes and JNI-based interoperability code. This approach is limited as it works only for Java classes that interoperate with C++ classes. In addition, it is difficult to maintain and evolve Exu-based systems because generated isomorphic classes may be changed by programmers.

Finally, generator-based approaches (e.g. JAXB, Apigen) can do automatic mapping for individual languages. For example, if an XML schema contains thousands of types then thousands of corresponding classes are generated in a host programming language that map to these XML types. This approach leads to serious problems with evolution and maintenance of generated code, like a complex naming mechanism, and results in a significantly increased compilation time of the system. In addition, as it often happens in commercial development, a schema[1] may not exist to generate corresponding classes at the time of writing an interoperating program. For example, it is customary to write prototype code that manipulates some XML data before a complete XML schema that describes this data is created, without a schema most generator approaches fail. Various generators are used as part of programming environments and as standalone tools to analyze FTSs and generate corresponding types in host programming languages. Most are generators that take XML schemas and generate corresponding classes in Java and C++ [10][11]. This approach requires sophisticated name management software and produces software that is difficult to maintain and evolve.

Polilinguality is implemented by *virtual machines (VM)* such as JVM and .Net CLR. These platforms also have limitations. Each has its own FTS and does not cover all other existing type schemas which exist beyond their scope (for example, there is no virtual machine that would reify HTML types to Java type system). Not only do separate low-level APIs exist for each platform, they are also vendor-dependent. These APIs introduce significant complexity and nonuniformity into

---

1. A *schema* is a set of artifact definitions in a type system that defines the hierarchy of elements, operations, and allowable content.

programming FTSs, let alone the steep learning curve required to master each platform. Extending any VM platform to support a new type system is difficult because it is very complex and fragile.

Various bridges are offered to interwork different polylingual systems [12]. For example, a JVM-COM bridge makes Enterprise Javabeans (EJBs) runs as COM objects. However, bridges offer complicated APIs that are vendor-dependent. Again, using bridges for FTS programming leads to nonuniform and complex code and requires time to learn their APIs.

3 Our Solution

ROOF is designed in light of principles of interoperable polylingual systems described in Section 2. The goals of our solution is to enable easily maintainable and evolvable polylingual interoperability by removing the need for elaborate name management solutions and allowing programmers to make decisions about sharing objects at the megaprogramming stage. The maintainability and evolvability of polylingual systems are achieved by using foreign objects by their names as they are defined in FTSs thereby eliminating the need for creation of isomorphic types in a host programming language and enabling programmers to share objects at the megaprogramming stage. We also provide a comprehensive mechanism for type checking that allows programmers to verify semantic validity of operations on foreign types both statically and dynamically.

Our solution is based on three assumptions. First, we deal with recursive type systems. Even though it is possible to extend our solution to higher-order polymorphic types, such as dependent types, we limit the scope of this paper to recursive types and imperative languages to make our solution clearer. Second, we rely on reflection mechanisms to obtain access to FTSs. Third, the performance penalty incurred by using reflection is minimal since the low-level interoperating mechanisms such as transmission, marshaling and unmarshaling network data has the largest overhead common to all interoperable solutions.

3.1 Type Graphs

Schemas can be represented as graphs whose nodes are composite types, leaves are primitive or simple types, and parent-child relationships between nodes or leaves defines a type containment hierarchy [13]. In order to operate on such graphs, a programmer must be able to reach nodes at arbitrary depth. This is accomplished via path expressions that are queries whose results are sets of nodes. A *path expression* is a sequence of variable identifiers or names of subordinate (or

containment) types that define a unique traversal through a schema.

```
class ReificationOperator
{
    public:
    ReificationOperator &GetObject( string t );
    int Count( void );
};
...
ReificationOperator R;
```

Figure 2: Declaration and instantiation of ReificationOperator class in C++

Suppose we have a handle to an object that is an instance of a foreign type. We declare this handle as an instance R of a ReificationOperator class shown in Figure 2. R enables navigation to an object in the referenced type graph by calling its method GetObject with a path expression as a sequence of type or object names t_1, t_2, ..., t_k as parameters to this method:

```
R.GetObject(t₁)...GetObject(tₖ)
```

Consider a schema that describes the organizational structure of a company shown in Figure 3. It is a directed graph where each node is named after an organizational entity within a company and edges describe the subordination of one entity to the other. Each node has attributes shown as line connectors with filled circles followed by the names of the attributes. The CEO's subordinate is the CTO who in turn supervises two departments shown as Test and Geeks. An instance of this schema may be given in existing markup languages such as HTML, XML, or SGML.

We simplify the notation for the ReificationOperator class by introducing array access operators [] that replaces the GetObject method. For example, if R denotes an instance of the schema shown in Figure 3, we can write the C++ program that counts the number of employees in the Geeks department as:

```
int n = R["CEO"]["CTO"]["Geeks"].Count();
```

The method Count returns the number of child nodes under a given path. Such notation is useful since a single line of lucid code is used to replace a lot of hand-written or generated code. No additionally defined types and operations are required. Path expressions symbolize simplicity and uniformity. These are the properties that we inherit from path expressions and they enable programmers to uniformly navigate through instances of foreign types. Further, since type names are used as they are defined in foreign schemas, there is no need to redefine them again in the host language. ***That is, we use the existing names of foreign types; we do not create corresponding types in the host language!*** We will show how this is accomplished shortly.
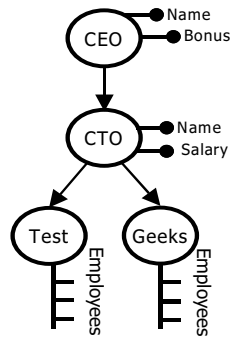
Figure 3: A schema of an organizational structure.

FTS-based applications often change each other's structures. A common example is a C++ application that changes the structure of an XML document. These modifications are complex and require carefully crafted software called transformation engines. However, since all type systems can be represented as graphs, these modifications can be reduced to transformations on graph structures. Thus, we reduce the task of manipulating FTS structures to that of manipulating graphs. A comprehensive set of basic operations used to manipulate FTS graph structures includes copying or moving a node to its new location, appending and removing a node, and logic set and relational operations on graphs.

By implementing these operations using a standardized notation we achieve uniformity of FTS-based code. *Indeed, polylingual programs written in different languages that perform the same operation on the same schema will look the same.* This very important property of uniformity enables effective program evolution and maintenance of and automated reasoning about polylingual systems.

3.2 Type Reification

In this section we show how to reify types. We first explain reification concepts, then their abstraction as reification operators, and then present detailed notes on implementation.

3.2.1 Reification Concepts

Reflection. Reflection is a powerful and common mechanism in contemporary programming languages and programming infrastructures (e.g., reflection in virtual machines). Reflection exposes the type of a given object; it reveals the public data members, method names, type signatures of method parameters and results, and superclasses (if any) of an object's class. Further, reflection enables a program to invoke methods of objects whose classes were not statically known at the time the program was compiled. It also allows a program to navigate a graph of interconnected instances without statically knowing the types of these objects. All of this information and power is available to a program at run-time.

Connectors. A *reification connector (RC)* is an example of an architectural connector [15][16]; it is a communication channel between a host language application and an application with a foreign type system. At the host language end, there are one or more classes corresponding to reification classes which accepts navigation instructions starting from a given foreign object. These instructions are transmitted via the RC to one or more classes in the foreign application, which executes these instructions and returns a reference to the resulting object. This is similar to the way methods of remote objects are executed in CORBA and the result is returned to the calling language. The difference is that there is no need to define explicit CORBA interfaces between the host (or client) application and the foreign (or server) application. Internally, we use a low-level API to transmit names of object attributes, names of methods, and primitive values to execute in the foreign application, and use reflection (on the foreign application side) to generate the appropriate method call.

Combining. Using reflection and reification connectors, a host program in one type system can navigate a graph of objects in a foreign type system. Suppose we are given a foreign object x and a path expression $x.a.b$. That is, starting with foreign object x, we access its "a" attribute to obtain some object y, and then we access the "b" attribute of y, as the result of the path expression.

In more detail given x, we transmit "a" to the foreign executable. Using reflection, we can validate that "a" is indeed a public member of x, and by invoking the appropriate get method (or simply variable access), we can access the "a" value of x. We return the handle of the resulting object y back to the host language, and repeat the process for attribute "b". This is the essence of our implementation.

3.2.2 Reification Operators

We mentioned in the previous section that a host application has a set of classes that hide the details of our implementation. In fact, the handle to a foreign object is the object R of Figure 2. R implements a *reification operator (RO)* that provides access to objects in a graph of foreign objects. We give all ROs the same interface (i.e., the same set of methods) so that its design is language independent; reification operators possess general functionality that can operate on type graphs of any FTS. By implementing R as an object-oriented framework that is extended to support different computing platforms, we allow programmers to write polylingual programs using a uniform language notation without having to bother about peculiarities of each platform. That is, for Java we have separate extensions of the framework that allows Java pro-

grams to manipulate C# objects, another extension to manipulate XML documents, etc. Similarly, for C# we have separate extensions of an equivalent framework that allows C# programs to manipulate Java objects, another extension to manipulate XML documents, etc.

Reification operators are thus nonsymmetrical, i.e. reifying types from FTS_I to FTS_J is not the same as the converse. RO has the transitive property, i.e. by applying RO R_{IJ} to an instance of FTS_I we reify it to the FTS_J. Then by applying RO R_{JK} to an instance of FTS_J we reify it to the FTS_K. The same result is achieved by applying RO R_{IK} directly to an instance of FTS_I. This property is useful since it enables the composition of reification operators to obtain a new RO. Finally, an identity RO reifies FTS types to the same type system, and interestingly, this is useful. Consider a Java program that needs to analyze its own structure. The identity RO enables such a reflective capability and extends it to all poly-lingual systems.

3.2.3 Reification of Methods

So far, we have described how host programs can navigate a graph of foreign objects. But in addition to navigation, we would like to invoke methods on foreign objects as well.

In order to reify methods we instruct the RO to set all parameters for a desired function and execute it in its native type system, and then reify the returned result.

We propose a reification model where operators << and >>, used to set and get values of reified type instances, and constitute the basis for operations on reified types. In the notation shown below, we use parentheses to specify attribute name a_r of type t_k.

$$R_{IJ}[t_1]\ldots[t_k](a_r) << E_j$$
$$R_{IJ}|t_1]\ldots[t_k](a_r) >> E_j$$

The operation << takes the value of a variable E_j and instructs the RO to set the particular attribute of a foreign object to this value; conversely the operation >> instructs the RO to obtain the value of the particular attribute of a foreign object and assign it to some variable E_j.

Consider setting the `Salary` attribute of the CTO to the value of integer `salary` and retrieving the value of the `Bonus` attribute of the type CEO into the variable `bonus` for an instance of the organizational schema shown in Figure 3.

```
int salary = 10000, bonus;
R["CEO"]["CTO"]("Salary") << salary;
R["CEO"]("Bonus") >> bonus;
```

Our notation can be used to reify methods in FTSs. Since a method has a unique type determined by its name, its signa-

ture types, and its return type, we treat a method name as a type t_k and its parameters as a set of attributes $\{a_r\}$. We set values for each parameter using the operation <<. Then by applying the RO to the typed operation t_k we execute it. The result of the execution is an instance of some type that is stored in some internal representation of the reification operator.

```
int j, rv;
string s;
RO_Java rj;
..............................
rj["SomeClass"]["mthd"]("ip") << j;
rj["SomeClass"]["mthd"]("sp") << s;
rj["SomeClass"]["mthd"] >> rv;
```

Figure 4: Example of a method reification.

A fragment of C++ code that provides an example of reification of a Java method is shown in Figure 4. A declaration for the reified Java method is shown in Figure 5.

```
class SomeClass
{
  public:
    int mthd( int ip, String sp {...}
}
```

Figure 5: Declaration of a Java class.

We declare RO `rj` that reifies Java type instances to C++. The method `mthd` declared as a member of Java class `SomeClass` has two parameters. We navigate to the method `mthd` and set the values of its parameters using the attribute semantics. Then we call this method, retrieve the return value and set it to the local variable `rv`.
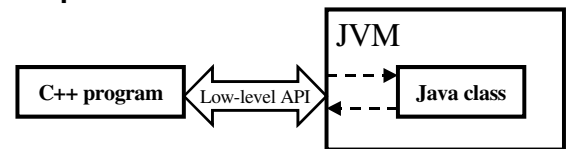
3.3 Implementation Details

Figure 6: A C++ program interacting with a Java class via JVM low-level API

Here is how a reification operator `RO_Java` that reifies Java types to C++ programs can be created. A C++ program uses low-level API to invoke a JVM and load it in the memory as shown in Figure 6 with a block arrow. Then, using the *Java Native Interface (JNI)* APIs as `FindClass`, `GetStaticMethodID` and `CallStaticVoidMethod`, a Java class can be loaded into the JVM, executed, and the results of its execution can be passed back to the C++ program. The interaction between the JVM and the Java class is shown in Figure 6 with dashed arrows.

```
class RO_Java
{
    public:
      RO_Java &Load( string name );
      RO_Java &GetMethod( string name );
      RO_Java &SetParam( string name, int v );
};

RO_Java rj;
```

Figure 7: Declaration and instantiation of RO RO_Java

An example of RO `RO_Java` is implemented as a C++ class declared and instantiated in Figure 7. For example, if we want to load Java class called `j.class` and execute its method `A` by setting its parameter `In` to integer value 5, we write the statement below:

```
rj.Load("j").GetMethod("A").SetParam("In", 5);
```

Now consider an RO `RO_CPP` that reifies C++ types to Java. It is an inverse operator to `RO_Java`. Using JNI, an instance of `RO_CPP` in a Java program invokes a C++ library and executes methods as shown in Figure 8. Even though the implementation of these operations is different from the ones of `RO_Java`, the class declaration is pretty much the same as shown in Figure 9.

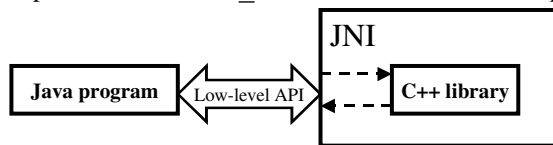The implementation of `RO_Java` is based on the concept of a

Figure 8: A Java program
interacting with a C++ library via Java Native Interface

shared stub described in [14]. It is a native method that dispatches to other native functions and is responsible for locating and loading libraries, passing arguments, calling native functions, and returning results.

```
class RO_CPP
{
  public:
    RO_CPP Load( string name );
    RO_CPP GetMethod( string name );
    RO_CPP SetParam( string name, int v );
};
```

Figure 9: Declaration of RO RO_CPP

Finally, consider an RO `RO_XML` that reifies XML types to C++. Using a DOM XML parser low-level API we load and parse XML data as shown in Figure 10. We can access any type or collection of types and change the structure of this data. We can also execute any method defined in an XLS document associated with any XML type. In this respect a declaration and implementation of `RO_XML` does not differ

fundamentally from `RO_Java` or `RO_CPP` shown in Figure 7 and Figure 9 respectively.
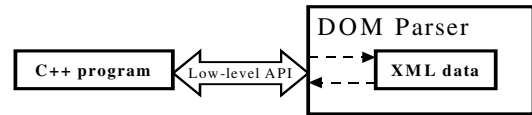

Figure 10: A C++ program
interacting with an XML data instance using DOM

The inverse RO to `RO_XML` that allows an XML instance to access C++ programs can be implemented as a C++ component that is loaded via XSL commands by a DOM parser and serves as a bridge between XML and C++ FTSs. The detailed discussion of this implementation is beyond the scope of this paper.

```
class GenericRO
{
  public:
    GenericRO &operator[]( string name );
    GenericRO &operator()( string name );
    GenericRO &operator<<( int i );
};
GenericRO ro;
```

Figure 11: Declaration and instantiation of
GenericRO.

At this point we can introduce a generic reification operator `GenericRO` shown in Figure 11 that uses [] and () operator overloading to provide uniform syntax and semantics to FTS-based programs. The operator [] allows programmers to access typed objects in FTS programs, for example, XML types and Java member variables and methods, and the operator () provides access to type attributes, for example, method parameters. Using this syntax we can rewrite the statement that loads Java class called `j.class` and execute its method `A` by setting its parameter `In` to integer value 5 as following:

```
rj["j"]["A"]("In") << 5;
```

The generality of RO `GenericRO` operator enables us to introduce a platform that uses `GenericRO` as an abstract parent class to implement different ROs derived from it. This way we provide uniformity to programmers and remove the use of low-level APIs thereby reducing complexity of polylingual systems.

4 Reification Object-Oriented Framework (ROOF)

The *Reification Object-Oriented Framework (ROOF)* targets FTS-based applications and provides a reification model. This model is intended for programmers writing polylingual systems. Unlike many existing frameworks that require a steep learning curve to understand the semantics of hundreds or

even thousands of classes and collaborations among them, ROOF effectively eliminates most of the classes and collaborations that programmers would otherwise have to develop or learn.

ROOF reifies types by providing a framework that is a collection of reusable classes that implement the following functionality:

- establish a channel between FTSs. This channel may be different from interprocess communication channels because it uses a low-level API to initialize a FTS environment and establish denotation of its control and data structures;

- retrieve a collection of instances of a desired type;

- retrieve a specific instance of the given type;

- set and retrieve values from an instance of the given type;

- invoke operations on type instances, and

- implement polymorphic/overloaded operations on the structures of polylingual systems;

Our goal in designing ROOF was to reduce the number of interfaces exposed to programmers to a bare minimum. A single class for each subscript I and J of reification operator R_{IJ} implements operations on reified type instances and structures of FTS-based applications.

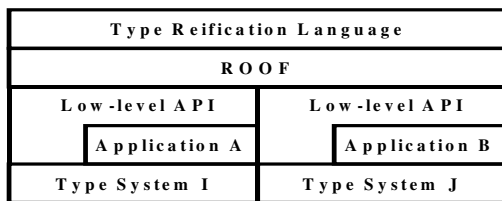| Type Reification Language	
ROOF	
Low-level API	Low-level API
Application A	Application B
Type System I	Type System J

Figure 12: Abstraction layers for a type reification framework.

A layered view of the reification framework is shown in Figure 12. Applications A and B are based on FTSs I and J respectively. For example, A may be based on C# and B is an XML instance. The access to these applications and FTSs is provided by low-level APIs, some C# API and some XML parser API respectively, that provide both control and reflective capabilities. The ROOF unifies FTSs by providing uniform polymorphic operations that are based on their low-level APIs.

5 Type Checking in ROOF

One of the perceived benefits of existing approaches to polylingual interoperability is that type checking is free since host language compilers perform it when compiling generated types and interoperability code. For example, given an XML schema, a generator can produce corresponding classes in Java along with interoperability code. A Java compiler performs type checking of the generated classes. Suppose that during the maintenance phase of polylingual software the XML schema has been changed. The Java compiler is not aware of this change and it would compile the generated classes without producing any warnings. However, the resulting program fails at run time because the interoperability code attempts to access XML objects that are either changed or do not exist. Obviously, this type checking does not meet its primary objective i.e., to produce errors when semantic inconsistencies with polylingual systems exist.

Our approach provides comprehensive type checking at three levels: host language compiler, runtime execution, and static checking across FTSs using our tool based on predefined typing rules. A detailed discussion of typing rules and the tool is beyond the scope of this paper, however, we sketch the basic ideas to understand how our approach works.

As shown in Figure 12, *Type Reification Language (TRL)* is a user interface provided by ROOF to enable programmers to write interoperable polylingual programs. Let us consider a fragment of TRL code embedded in C++ shown in Figure 13.

```
RO_XML R;
int salary;
............
try
{
  R["CEO"]["CTO"]("Salary") << salary;
}
catch( RO_Exception &e ) {...}
```

Figure 13: A fragment of TRL code embedded in C++.

A C++ compiler type checks the compatibility between the operator << expected parameter types on its right and left hand sides. The type of variable salary is integer and the object variable R is of type RO_XML that are consistent with the declaration of operator << given in Figure 11.

The major limitation of the first level of checking using the host language compiler is in not being able to infer the type of object variable R when navigating to a foreign object. For example, the type of object variable R stays RO_XML when code R["CEO"] is executed instead of changing to the CEO foreign type. If the structure of the foreign object changes for some reason (e.g. the CTO object is removed), then the host language compiler will not be able to detect it, and the program produces a runtime error. This situation is corrected by RO producing exceptions at runtime that is shown in Figure 13 using try and catch blocks.

Finally, we provide a mechanism to perform type checking across FTSs statically. Since reflection is the primary assumption for our approach to work, we can use it to retrieve type

information about foreign objects at compile time and use it to verify the validity of operations in host language. To do that we create a tool based on EDG C++ and Java compiler front ends [17] that parses polylingual programs and uses reflection to decorate foreign types in abstract syntax trees, and use our type checking rules to verify the semantic validity of operations on foreign types.

6 A Real-World Project

One of the authors (Grechanik) applied our approach to a real-time component-based semiconductor overlay analysis and control system. The Archer Analyzer is a software package geared for Archer 10 optical overlay metrology systems manufactured by the California-based KLA-Tencor Corporation [18][19].

The purpose of optical overlay measurements is to detect and fix misalignments between layers of semiconductor chips that were put on a silicon wafer using microlithography processes. Overlay or misregistration is a vector quantity defined at every point on the wafer. Ideally, the value of overlay should be zero. When nonzero overlay is detected the tool is stopped and the error is corrected as soon as possible.

The first attempt was started by KLA-Tencor in 1997. A team of forty specialists was assembled; the original system was written in a programming language REXX in the beginning of 80's and became obsolete. The design for the first release of the product was done carefully. The management allocated more than a year to hire people, educate them about the company and existing systems and processes, and to produce design documentation. Each group planned high-level design, functionality and interfaces in great detail. The approach to design of this polylingual system was IDL-based with CORBA and DCOM used as the underlying distributed object middleware platforms. The number of classes, data structures, and various methods and functions was in the thousands, and the complexity of the polylingual system grew to such a degree that communication overhead between groups became excessive. Clearly, the selected approach was not scalable. This situation was compounded by the conflicting and overlapping terminology that led to many syntactically similar types serving different purposes. When the size of code grew to two hundred thousand lines the project became unmanageable.

It took four years and more than $3 million to design and develop the software and take it to beta test at Texas Instruments Corporation. The test failed with MTF $\approx$ 2 hours despite the initial requirement for MTF $\approx$ 2,000 hours. Each time the system went down it was virtually impossible to determine the cause of breakdown since each group claimed that its components performed internal algorithms properly and the algorithms themselves were correct. The management tried to hire more people who would handle the inter-component connectivity, but it realized soon that the project would require significant additional investment as the number of failures increased the further testing went. Therefore, management decided to start from scratch.

The second attempt was started in January 2001. One of the authors who was hired as a consultant to define the strategy for its implementation saw that it was very difficult to extract full and correct information about all aspects of communications among different modules of the existing system. Each group member of the first implementation could clearly explain the programming logic of his/her code and had a clear picture of a schema of data pertinent to the part of the project the s/he owned. However, the understanding of the overall structure of the project, relations among modules and data was vague. When programmers followed the process of creating structures that map FTS types, they created a system based on many wrong assumptions that was extremely difficult to trace in the resulting code. When brought together these group members could not agree upon all details of the big picture of the project.

We applied our approach to target the key problems of the project. *The first was to make each member of the team think about a common schema. The second was to introduce uniformity and reduce the complexity of code. The way to do it was to eradicate the thousands of explicit mappings between FTS types.*

The first problem was addressed with the type graph solution explained in Section 3.1. A single schema and its instance were created by an engineer whose job was to maintain the schema and serve as a single point of reference to define every term. The second problem was solved by enabling programmers to reference each type exactly as it was defined in the schema. This required a common platform that subsumed all other FTSs used in the project. Thus, ROOF was created. Each team member was given a thirty minute presentation of the basic structure of ROOF and its operations. Moreover, each programmer was told that if s/he found the concept and implementation of the ROOF difficult to understand and use they may go back to low-level APIs that they originally planned on using. ***They did not go back***.

The architecture of the *Archer Analyzer (AA)* is based on many FTSs. AA is created as an open system and is integrated in the production environment and communicates with many other FTSs, such as EJBs, CORBA, and .Net assemblies. The components for AA are created using C++ and different low-level APIs were used for a variety of tasks such as parsing XML data and forming matrices. Since we did not generate or create mapped types in the host programming language it is

not difficult to see how we achieved the reduction of code close to 80% for this project.

The results exceeded expectations. The system was much easier to write and the resulting code was clear. It took over a year with a team of six programmers using our approach to deliver this project to beta test at AMD Corp. that was successfully passed in 2002. This software has since been successfully commercialized.

7 Conclusions and Further Work

In today's enterprise environment it is desirable to make each program interoperate with other programs. However, existing solutions have the significant accidental complexity as there are potentially $O(n^2)$ possible APIs for achieving communication among n programs. We have solved the problem of enabling all-connected architecture graph of polylingual systems by providing a single framework called ROOF with a single, standard, and clear API.

ROOF is a simple and effective way to develop easily maintainable and evolvable large-scale polylingual systems by reifying foreign type instances and their operations into first-class language objects and enabling access to and manipulation of them. By doing so we hide the tremendously ugly, hard-to-learn, hard-to-maintain, and hard-to-evolve code that programmers must write or generate today, i.e. we simplified polylingual code, making it scalable and easier to write, maintain, and evolve.

The capability to write uniform and compact programs that work with applications based on different type systems enables faster development of complex systems at a fraction of their cost. Indeed, if developers concentrate on reasoning about properties of applications without the need to master many low-level APIs that operate on type system elements then it significantly improves their productivity and the quality of the resulting system. We came to this conclusion based on using our approach for a complex commercial software system and for a variety of different smaller projects.

Type checking is an important mechanism to guarantee the safety of ROOF-based programs and improve programmer's productivity. We need a compiler that is capable of checking reified objects against foreign type systems. However, a thorough discussion of type checking is beyond the scope of this paper and is a subject of ongoing work that we conduct in this area.

8 References

[1] D. Garlan, R. Allen, and J.Ockerbloom, "Architectural mismatch: Why reuse is so hard," IEEE Software, vo. 12, no. 6, November 1995, pp. 17-26.

[2] D. Barrett, A. Kaplan, and J.Wileden, "Automated support for seamless interoperability in polylingual software systems," *Fourth Symposium on the Foundations of Software Engineering*, October 1996.

[3] F.Brooks, *The Mythical Man-Month*, Addison-Wesley, August 1995.

[4] A. Kaplan and J.Wileden, "Software interoperability: principles and practice," *ICSE* 1999.

[5] A. Kaplan and J.Wileden, "Toward painless polylingual persistence," *Seventh International Workshop on Persistent Object Systems*, May 1996.

[6] J. Auerbach, C. Barton, M. Chu-Carroll and M. Raghavachari, "Mockingbird: Flexible Stub Compilation from Pairs of Declarations," *19th IEEE International Conference on Distributed Computing Systems*, May 1999.

[7] A. Kaplan, V. Ridgway and J.Wileden, "Why IDLs are not ideal," *Ninth IEEE International Workshop on Software Specification and Design*, April 1998.

[8] D. Barret, "Polylingual Systems: An Approach To Seamless Interoperability," *Ph.D. thesis*, University of Massachusets at Armherst, May 1998.

[9] A. Kaplan, J. Bubba and J.Wileden, "The Exu Approach to Safe, Transparent and Lightweight Interoperability," *25th Annual International Computer Software and Applications Conference (COMPSAC'01)*, October 2001.

[10] Institute for Software Research, University of California, Irvine, *xADL 2.0 project, Apigen for xArch schemas*, http://www.isr.uci.edu/projects/xarchuci/tools-apigen.html

[11] Sun Microsystems, "Java Architecture for XML Binding (JAXB)," http://java.sun.com/xml/jaxb/.

[12] W. Emmerich*, Engineering Distributed Objects*, John Wiley, July 2000.

[13] S. Abiteboul, P.Buneman, D.Suciu, *Data on the Web: From Relations to Semistructured Data and XML*, Morgan Kauffman Publishers, 2000.

[14] S. Liang, *Java Native Interface: Programmer's Guide and Specification*, Addison Wesley, June 1999.

[15] D.Perry and A.Wolf, "Foundations for the Study of Software Architectures", *ACM SIGSOFT Software Engineering Notes* 17(4), 1992, pp. 40-52.

[16] D.Perry, "Software Architecture and its Relevance for Software Engineering", *Keynote at Coordination 1997*, Berlin, September 1997.

[17] Edison Design Group, http://www.edg.com.

[18] Semiconductor Business News, http://www.siliconstrategies.com/story/OEG20020708S0052.

[19] KLA-Tencor, "Archer Analyzer Automated, Real-Time Overlay Metrology Analysis," *Technical Fact Sheet*, http://www.kla-tencor.com/products/archer10/archer_analyzer_tech_factsheet.html