Automated Support for Seamless Interoperability in Polylingual Software Systems

Daniel J. Barrett
Alan Kaplan
Jack C. Wileden
{barrett,kaplan,wileden}@cs.umass.edu

CMPSCI Technical Report 96–64
October 1996

Convergent Computing Systems Laboratory
Computer Science Department
University of Massachusetts
Amherst, Massachusetts 01003

Appeared in Proceedings of the Fourth Symposium on the Foundations of Software Engineering San Francisco, CA., October, 1996

This paper is based on work supported in part by Texas Instruments, Inc. under Sponsored Research Agreement SRA-2837024 and by the Air Force Materiel Command, Phillips Laboratory, and the Defense Advanced Research Projects Agency under Contract Number F29601-95-C-0003. The views and conclusions contained in this document are those of the authors. They should not be interpreted as representing official positions or policies of Texas Instruments or the U.S. Government and no official endorsement should be inferred.

Automated Support for Seamless Interoperability in Polylingual Software Systems

Daniel J. Barrett
Alan Kaplan*
Jack C. Wileden

Convergent Computing Systems Laboratory†
Computer Science Department
University of Massachusetts
Amherst, MA 01003

E-mail: {barrett,kaplan,wileden}@cs.umass.edu

Abstract

Interoperability is a fundamental concern in many areas of software engineering, such as software reuse or infrastructures for software development environments. Of particular interest to software engineers are the interoperability problems arising in *polylingual* software systems. The defining characteristic of polylingual systems is their focus on uniform interaction among a set of components written in two or more different languages.

Existing approaches to support for interoperability are inadequate because they lack *seamlessness*: that is, they generally force software developers to compensate explicitly for the existence of multiple languages or the crossing of language boundaries. In this paper we first discuss some foundations for polylingual interoperability, then review and assess existing approaches. We then outline PolySPIN, an approach in which interoperability can be made transparent and existing systems can be made to interoperate with no visible modifications. We also describe PolySPINner, our prototype implementation of a toolset providing automated support for PolySPIN. We illustrate the advantages of our approach by applying it to an example problem and comparing PolySPIN's ease of use with that of an alternative, CORBA-style approach.

1 Introduction

Large and complex software systems invariably consist of multiple software modules (programs, subprograms, collections of subprograms). For any number of reasons, such as a desire to reuse legacy components, or because particular languages facilitate development of particular kinds of applications, those modules may be written in several different languages. As a result, the problem of cooperation among software modules of different languages – the multi-language *interoperability* problem – naturally arises in such systems.[1]

Suppose two architecture companies, the Frank Firm and Lloyd Ltd., are contemplating a merger. Each company has important software assets, including everything from personnel information to design element descriptions to computer aided architectural design (CAAD) tools, that the merged company, Frank Lloyd, Inc., will wish to integrate to form its own software infrastructure. While Frank's assets are implemented in C++, Lloyd's are implemented in CLOS (Common Lisp Object System). If the merger is to be successful, Frank Lloyd must solve the multi-language interoperability problem, preferably in a way that minimizes its impact on the newly merged software development staff. In particular, a solution that requires substantial translation of existing code or data, or that forces significant retraining of any part of the staff, will threaten the competitive advantage that Frank Lloyd expects the existing assets to provide.

In addressing multi-language interoperability problems, such as those confronting Frank Lloyd, we make several distinctions. First, we differentiate situations in which a component written in one language needs to access one or more components (subprograms, data objects, etc.) written in a single second language from those in which a component written in one language needs to uniformly interact with a set of components written in two or more different languages (which may or may not include the language of the first component). We refer to the former, more homogeneous situation as *multilingual* interoperability, and the latter, more heterogeneous situation, as *polylingual* interoperability.[2] In our Frank Lloyd example, a CAAD tool written in C++ that invoked an aesthetics-assessment subprogram written in CLOS would be an instance of multilingual interoperability. A CLOS program that accessed personnel records, some

*Alan Kaplan is now with the Department of Computer Science; Flinders University; GPO Box 2100; Adelaide, SA 5001; Australia. (kaplan@cs.flinders.edu.au).

†This paper is based on work supported in part by Texas Instruments, Inc. under Sponsored Research Agreement SRA-2837024 and by the Air Force Materiel Command, Phillips Laboratory, and the Defense Advanced Research Projects Agency under Contract Number F29601-95-C-0003. The views and conclusions contained in this document are those of the authors. They should not be interpreted as representing official positions or policies of Texas Instruments or the U.S. Government and no official endorsement should be inferred.

Appeared in the Fourth Symposium on the Foundations of Software Engineering (FOSE), October 1996, San Francisco, CA. Also available as technical report UM–CS–1996–064, Computer Science Department, University of Massachusetts.

[1] Interoperability problems can also arise in systems written in a single language — for example, systems that span several hardware platforms or several dialects of the same language. This paper focuses on multi-language interoperability, which will henceforth be referred to simply as "interoperability."

[2] This terminology is inspired by the analogy between "uniform processing independent of data type" (polymorphic) and "uniform processing independent of language" (polylingual).

stored as C++ objects and others stored as CLOS objects, to assign employee office space would be a case of polylingual interoperability.

Second, we distinguish approaches by their level of *transparency* to software developers. Most current approaches force developers to be aware that multiple languages are involved and to build their modules to surmount the language boundaries. Multilingual interoperability may be achieved, for example, by using heterogeneous remote procedure calls, while polylingual interoperability can be realized by enforcing the use of a common, foreign type model among the modules (e.g., as in CORBA [ClPCC+93]).

If developers of a multi-language software system need not be aware of language differences between the software modules, we call the interoperability approach *seamless*. For example, suppose that two new CAAD tools under development at Frank Lloyd, called *Form* and *Function*, are being written in different object-oriented programming languages (say C++ and CLOS, respectively) and they must share objects. In a typical multilingual system, the developers of *Form* and *Function* must write special code for accessing each others' objects across the language boundary, either directly (using low-level foreign function calls, for example) or through an intermediary (e.g., a CORBA ORB). In a seamless approach, however, *Form* and *Function* could each access C++ and CLOS objects as if there were no language barrier. To *Form*, for instance, there would be no discernable difference when invoking the methods of a C++ or a CLOS object.

Such seamlessness has long been a highly desirable property of interoperability, but it has rarely been achieved. Our approach, PolySPIN, transparently and automatically modifies a set of objects, implemented in diverse languages, so they can be accessed seamlessly by software modules of different programming languages. That is, PolySPIN allows a programmer to proceed as if the languages of the object and the accessor were the same. All software modules use only their native type systems to access these objects, not a common (but foreign) type system like CORBA's IDL, and existing modules need not be modified to do it. A prototype toolset supporting the approach currently automates implementation of seamless interoperability for C++ and CLOS objects.

The remainder of the paper is organized as follows. In Section 2, we discuss some foundations for polylingual interoperability. Section 3 reviews existing approaches to interoperability, focusing particular attention on CORBA-style approaches. Section 4 outlines our approach to seamless interoperability in polylingual systems and describes the toolset automating its use. Section 5 provides an example application of our approach and compares it to a CORBA-style approach. Section 6 summarizes the current status of PolySPIN and discusses some future directions for this work.

2 Polylingual Interoperability

In this section, we provide an overview of polylingual interoperability concepts. We first offer a simple but useful classification of interoperability situations. We then consider several dimensions of interoperability, which we summarize in a model representing a generic polylingual interoperability situation. These foundations provide a basis for subsequent discussion of various approaches to interoperability: both existing approaches (Section 3) and PolySPIN (Section 4).

2.1 Classification of Interoperability Situations

The decision to cause two software modules A and B to interoperate can be made at three different times in the software lifecycle: before A and B have been written, after A but before B has been written, or after both A and B have been written. The first scenario is the *easiest case*: since neither A nor B exists yet, a developer can specifically design them to interoperate. Developers are seldom fortunate enough to be able to design both interoperating modules from scratch, however. The second scenario is a more *common case*, in which a new software module B must be designed to interoperate with legacy system A. In this case, it is desirable that legacy module A need not be modified; but unless A was designed with future interoperability in mind, this is unlikely.[3]

Our approach to automated support for interoperability in polylingual systems addresses the third and most difficult of the three scenarios, sometimes called *megaprogramming* [BS92, WWC92]. Since both A and B exist, we want to modify them as little as possible to make them interoperate. Our approach allows A and B to interoperate with no modifications visible to the developer nor the modules. Existing approaches such as CORBA are problematic in this scenario because they are intrusive, requiring both A and B to be modified in a significant way: translating some of their data type definitions into another language (IDL), and changing the protocol by which those data are accessed (ORB calls instead of native language constructs).

2.2 Dimensions of Polylingual Interoperability

As a basis for understanding, comparing and developing interoperability approaches, we have found it helpful to view interoperability as having several dimensions. Our model of interoperability, depicted in Figure 1, illustrates this view.[4] In this model, one or more software modules, called *accessors*, need to access other software modules, called *objects*. (A given software module can be both an accessor and an object.) The accessors can be written in diverse languages and, since we are modeling polylingual interoperability, the objects are implemented in at least two different languages.[5]

The four central subcomponents in Figure 1 represent dimensions of interoperability:

Interlanguage naming (Locator): Interoperability cannot occur without some means for accessors to locate or reference objects. Thus, an approach to polylingual interoperability must include some mechanism for interlanguage naming (or its equivalent). In the model, that mechanism is termed a Locator.

Language information (Language Arbiter): Although language differences between accessors and objects are hidden in polylingual systems, they must be addressed at some level. Our model includes a mechanism for transparently associating language-specific information with a given object. This mechanism is termed a Language Arbiter.

[3] Moreover, even if legacy module A were designed with interoperability in mind, that is no guarantee that it is an appropriate kind of interoperability to use with future module B.

[4] We stress that this is a conceptual model, not to be interpreted as prescribing an implementation strategy. As will be seen in Section 4, however, our prototype implementation closely follows the model.

[5] If all the objects were implemented in a single language different from that of the accessors, this would be a model of multilingual interoperability.

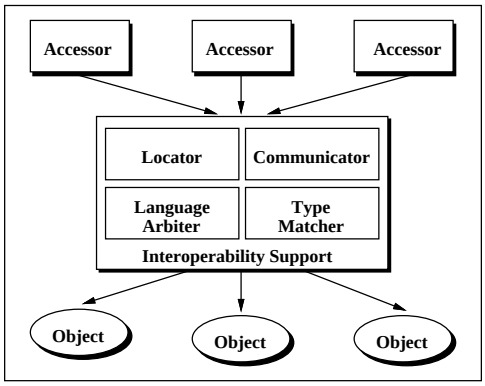

Figure 1: Conceptual Model of Polylingual Interoperability

Interlanguage invocation (Communicator): If the accessor and object are implemented in different languages, an approach to interoperability must provide a mechanism for invoking operations across the language boundary, including marshalling and unmarshalling of arguments. In the model, this mechanism is termed a Communicator.

Type compatibility (Type Matcher): In order to have uniform access to objects, the data types utilized by an accessor and its accessed objects must be sufficiently compatible that communication between them is sensible and meaningful. In the model, satisfying this requirement is the role of the Type Matcher.

2.3 Scope of Polylingual Interoperability

Our polylingual interoperability research focuses on interlanguage issues. It does not directly address the full problem of schema mismatch, in which models of objects having significant semantic differences, implemented in the same or different languages, must be reconciled. Rather, our approach provides an organized mechanism for dealing with schema mismatch problems that arise solely from language differences.

Our approach is designed to work best with languages that encapsulate operations within types, i.e., abstraction-oriented languages (e.g., Ada 83) and object-oriented languages (e.g., CLOS, C++ or Ada 95). In order to apply this approach to older procedural languages that do not support encapsulation, such as C and FORTRAN, some mechanism like that provided by Polylith [Pur94] would be needed for associating operations with datatypes.

3 Existing Approaches to Interoperability

A variety of approaches to interoperability have been developed over the years [WWRT91]. Existing approaches tend to fall short of our goal of supporting seamless interoperability in polylingual systems, however. In this section, we describe several major categories of existing approaches in terms of our four dimensions of interoperability, and assess their effectiveness in achieving seamlessness. We then give more detailed consideration to CORBA-style approaches, since they not only come the closest of any existing approach to supporting seamless interoperability in polylingual systems, but also are quite popular. We apply one such approach, ILU, to an interoperability problem that might face Frank Lloyd, and then assess its level of seamlessness.

3.1 Overview of Existing Approaches

Low-level approaches: Some languages provide a foreign function call interface, allowing a program p written in one language to invoke a subprogram s written in another. The object code modules of p and s are linked, and the foreign function call is accomplished within a single address space. For calling across address spaces, remote procedure calls [BN84] have been extended to cross language boundaries (e.g., [MHO96]). Both foreign function calls and remote procedure calls represent relatively low-level approaches to interoperability, and as a result they are generally far from seamless because application programmers frequently must deal with details of parameter marshalling and because complex data types (pointer-based data structures, ADTs) are not supported. In terms of our model, such approaches assist software developers with interlanguage invocation but provide little support for the other dimensions of interoperability.

Messaging systems: Some systems accomplish interoperability by allowing software modules to send messages to each other or to a central message server that routes messages to their intended recipients. Well-known examples are FIELD [Rei90] and Polylith [Pur94]. These systems primarily support interlanguage invocation and may also offer some, typically primitive, assistance with interlanguage naming. They seldom, however, provide much support for the remaining dimensions of language information and type compatibility. Approaches in this category are usually far from seamless, since software modules must be created or modified to adhere to the messaging interfaces supported by these systems. Significant overhead can be incurred if software modules must continually translate data from their native type systems into messages and back again.

CORBA-style approaches: While the preceding classes of approaches focus on interlanguage invocation, CORBA-style approaches have emphasized the type compatibility dimension. They require all interoperating software modules to adhere to a single type model, separate from that of the modules' languages. This approach is typified by CORBA [ClPCC+93] and ILU [JSS95]. Software modules are considered to be of two kinds: objects, which provide public interfaces, and clients, which invoke the methods of objects. These objects may be written in numerous languages, but a wrapper must be created for each object before it can be accessed by clients. Wrappers defined in an interface language (e.g., CORBA's IDL or ILU's ISL) provide a language-independent interface to the object. This interface can then be translated automatically into a client's language, allowing the client to interoperate with the wrapped objects.

In addition to addressing type compatibility, object wrappers generally encapsulate interlanguage invocation, interlanguage naming, and object implementation language information. Thus, these approaches offer some support for all dimensions of polylingual interoperability. They fall short with respect to seamlessness, however, because clients use their language's type system for accessing local data and another (the interface language) for accessing objects. This shortcoming will be discussed further in Section 3.2.

Database approaches: Some databases provide application programming interfaces (APIs) or query language bindings for multiple languages, thereby allowing programs written in different languages to share data by accessing a common database. The database approach, however, is an ancestor of the CORBA-style approaches and hence suffers from the same lack of seamlessness. In particular, shared objects must be created within the database's type model, usually different from a given language's type model, and the database's query language must be used when accessing those objects. Object-oriented databases, such as the TI/Arpa Open OODB [WBT92], allow accessors to create and access objects using their own language's type model, but typically no facilities are provided for interoperability. That is, data stored in the database via one API cannot be accessed via another.

Compound documents: Microsoft's OLE [Bro94] and Apple's OpenDoc [App94] support interoperability by allowing objects, called compound documents, to contain or refer (point) to other objects. For example, a Microsoft Word document can contain a reference to an Excel spreadsheet object, and when the object is accessed, the operations invoked on it are automatically routed to Excel. As with the previous approaches, software modules must be specifically written or modified to make use of compound documents: e.g., making calls to a compound document manager. Hence, while they may provide or encapsulate all the subcomponents in our model, these approaches also fall far short of seamlessness.

Languages and language extensions: The subject-oriented programming paradigm [HO93] permits objects to be accessed via more than one public interface. Each accessor can potentially use a different interface to access the same object. Subject-oriented programming has been limited to a single language, so it is not currently an approach to interoperability; however, a natural extension is for an object to be accessible from different languages via different interfaces. In this sense, our approach to polylingual systems may be thought of as a generalization of subject-oriented programming, since (as shall be described in Section 4) each language has its own interface to an object, with the selection of interface handled automatically when the object is accessed.

Concert [AR94] is a system that uses an extended C language, called Concert/C, to support interoperability. Like the CORBA-style approaches, Concert uses an intermediate interface language; but unlike CORBA and ILU, Concert's interfaces are automatically generated. So Concert eliminates the need for a human-created wrapper, at the cost of extending the programming language – a cost not incurred by our approach. Although Concert supports only C and C++, a successor to Concert, called Mockingbird [Aue96], is intended to provide interoperability between C++, Java, CORBA IDL, and other languages.

3.2 Example Application of a CORBA-style Approach

We now describe an application of a representative approach to a specific interoperability problem in the Frank Lloyd example. Specifically, we illustrate the use of a particular CORBA-style approach, namely the Inter-Language Unification (ILU) system, an interoperability mechanism developed at Xerox PARC [JS94].

3.2.1 An Example Scenario in Frank Lloyd

As noted earlier, a potential interoperability problem for the new Frank Lloyd company might involve an application that assigns office space based on information contained in personnel data. The new company would like to make use of a CLOS application developed by Lloyd Ltd. Figure 2 shows a simple CLOS function in which personnel with higher office rankings receive priority for office assignments.

```
(defun office-rank (employee)
  ;;; rank = years-of-service * salary / 10000
  (/ (* (YearsOfService employee)
        (Salary employee)
     )
     10000
  )
)
```

Figure 2: Primary Function Used in CLOS Application

Recall that personnel data for employees from the Frank Firm are maintained in C++, while Lloyd Ltd. employee personnel data are maintained in CLOS. The new company would like to make use of the CLOS application without having to translate the C++-maintained personnel information and with only minimal changes to the original CLOS application.

3.2.2 An Overview of ILU

One approach that the Frank Lloyd company might use to solve this problem is provided by ILU. ILU can be viewed as an approach to creating client-server architectures, in which language-specific servers manage instances of classes, and clients access and manipulate objects by invoking requests on these servers. With respect to the Frank Lloyd example, a CLOS server would manage CLOS-defined employee data for Lloyd Ltd. and a C++ server would manage similarly defined data for Frank Firm employees. The CLOS office ranking application would be an example of a client in ILU.

In ILU, interoperating classes are specified using an interface description language called ISL (Interface Specification Language). Classes are described by declaring a class identifier and associating a set of operations with the class. Figure 3 shows the ISL for an Employee class based on our example scenario.

```
INTERFACE Employee;

Type ClassInterface = OBJECT
  METHODS
    Salary () : INTEGER;
    YearsofService () : INTEGER;
  END;
```

Figure 3: ISL for Employee Class

Given the ISL for a class, creating an ILU-based polylingual application requires that the following steps be per-

formed:

1. Create class interfaces for clients. This is accomplished by applying language-specific translators (provided by the ILU development environment) to the ISL class description. For example, an ISL-to-CLOS translator creates a CLOS class interface for employees, while an ISL-to-C++ translator similarly creates a C++ class interface. The ILU translators also produce additional server and client code that is simply compiled and linked into a client and server, respectively, but can otherwise be ignored by the developer. This code serves the purpose of the Language Arbiter and Communicator defined in our model.

2. Construct the servers:

   (a) A separate class, which is a subclass of the class generated in step 1, must be supplied by the programmer. This class contains the actual implementations of the various member functions, as well as any required data members (which are part of the implementation, not defined as part of the class). Thus, the server class is used by the servers, while the client class is used by clients.

   (b) A server program must be implemented. The server program essentially creates instances of the class implementation, publishes names for them in a globally available, shared area, and then waits for requests.

3. Construct the clients. Clients access objects by issuing requests in the form of names (or identifiers) to servers. With respect to our scenario, the CLOS office ranking application corresponds to an ILU client. A client interacts with a server through the interface generated in step 1. Thus, the client views an instance as if it were implemented in its own language, even though it may turn out to be implemented in a different language.

4. Invoke the servers and the clients. With respect to the example scenario, each server would manage instances of personnel data, and the client, i.e., the office ranking application, would access the instances via these servers.

3.2.3 An Assessment of ILU

ILU provides support for each of the dimensions in our interoperability model. As mentioned above, a name service, albeit a simple one, allows applications to locate objects. In addition, the ILU ISL translators generate the required language arbiters and communicators. As noted in Section 3.1, CORBA-style approaches emphasize the type compatibility dimension. ILU imposes a language-external type model on application developers, who must use ISL to describe types instead of using the native (CLOS, C++, etc.) type model. This approach is best suited for the easiest case, and perhaps the common case, interoperability scenario. For example, if it is known *a priori* that two (or more) modules may need to interoperate, then an application can of course start with an ISL description of the interoperating classes. More problematic is the megaprogramming scenario, as in the Frank Lloyd example. Using ILU in this case would require the application developer to "wrap" the existing classes with ISL descriptions. In addition, the existing application would need to be modified to use the interface produced by the ISL translators.

In summary, a significant problem with CORBA-style approaches, such as ILU, is that software modules cannot create and access shared objects by using their own type systems. They must use an external, common type model, which may offer only a subset of the capabilities of the native language's type model, so some types cannot be expressed. A related problem is that the common type model is not transparent to the software modules; thus, legacy systems must be modified or retrofitted to use the interface language, and programmers must learn and reason about a separate type model. As we demonstrate in the forthcoming sections, a major feature of our approach is that no common type model is imposed on application developers, and therefore no interface language need be used by developers. Each software module can access shared objects via its language's native type system.

4 PolySPIN and PolySPINner

As indicated in Section 3.1, existing approaches to interoperability are generally not seamless. Because they typically involve use of a different invocation mechanism (e.g., low-level approaches and messaging systems) or a different type model (e.g., CORBA-style and database approaches) from those of the programming languages in which accessors are implemented, it is difficult to imagine how these approaches could be made transparent, even through automation. Our approach, on the other hand, imposes neither different invocation mechanisms nor different type models and hence supports transparent, seamless interoperability. In this section we outline our approach and describe the toolset that automates its use.

4.1 PolySPIN

PolySPIN is an approach to Support for Persistence, Interoperability and Naming in POLYlingual systems [KW96]. The key component of PolySPIN is a language-neutral name management mechanism that allows for uniform name-based access to objects [Kap96]. This mechanism supports, but does not require, a uniform model of orthogonal persistence (see [WWFT88] for an overview) and/or transparent polylingual interoperability.

The interoperability aspects of PolySPIN supplement the name management component (locator) with communicator, type matcher and language arbiter functionality, all of which are transparent to accessors. Language information (language arbiter) is associated with objects as part of the name-object binding. The interlanguage invocation (communicator) functionality is achieved by automatically modifying the implementation of object methods. The modified methods consult the language arbiter at each invocation and transparently select between making a local method call or an automatically generated interlanguage call. Since this communicator mechanism is encapsulated within the original interfaces defined for the object-related procedures and functions, it is entirely transparent to accessors. Similarly, the type matcher functionality is achieved via automated type compatibility checking whose results influence the modifications made to object method implementations. This type matching is transparent to accessors since it does not depend upon explicit use of any non-native type models and does not cause any type definitions or interfaces to be modified.

4.2 PolySPINner

PolySPINner is a toolset automating the application of the PolySPIN technique. Currently, our prototype supports interoperability between C++ and CLOS programs. PolySPINner is extensible because it consists of generic components that can be instantiated for different languages. This section discusses the PolySPINner tool, including its architecture, foundations, and implementation.

PolySPINner operates in the following manner. Given a set of accessors and objects implemented in programming languages $l_1, l_2, \ldots, l_m$, the user supplies PolySPINner with the type definitions (both interface and implementation) of the objects. PolySPINner modifies the implementation of each type (that is, the method implementations) so that its methods become callable from all languages $l_1, \ldots, l_m$. This instrumentation provides all the interoperability support functionality specified in our model (Figure 1).

In order for PolySPINner to accomplish this, some assumptions are necessary. The primary assumption is that for each object type t_1 written in language l_1 (without loss of generality), the user must also supply corresponding object types $t_2, t_3, \ldots, t_m$ created in languages $l_2, l_3, \ldots, l_m$, that "match" t_1. (Matching is discussed further below.) After PolySPINner has done its work, any call to a method of t_1 that comes from a program in language $l_i, i \geq 2$, is transparently converted into a corresponding call to a method of type t_i. Since none of the type interfaces are modified, application programs (accessors) that previously used these types need no modification in order to interoperate via these types. Thus, for example, even though the Frank Firm and Lloyd Ltd. implemented their respective concepts of "employee" in different languages and with different method calls to access the person's name, age, occupation, and so on, PolySPINner in theory can allow an application developed by the Frank Firm for processing its personnel information to be applied to the merged database of Frank Lloyd with no visible modifications to the objects and no modifications at all to the application.[6]

4.2.1 PolySPINner Architecture

The architecture of PolySPINner is presented in Figure 4. Each set of type definitions is fed through a Parser component to convert it into a language-independent intermediate representation. In this form, types are matched via a Matcher component so that calls to a method in one language are converted into calls to another method in a different language. Finally, the type definitions, matcher output, and other information are fed to a Generator that creates types that are accessible from various languages. These types have interfaces that are identical to the originals.

PolySPINner is extensible because we have implemented generic Parser, Matcher, and Generator components. By instantiating these components with particular parsing requirements, match criteria, or generation information, a developer can tailor PolySPINner for a given set of languages without affecting PolySPINner's overall behavior.

The Parser PolySPINner's generic Parser component is responsible for managing the parsing of type definitions into PolySPINner's language-independent intermediate representation. In its current form, this intermediate representation

[6] As noted in section 2.3, full discussion of database merging issues, such as semantic heterogeneity [SL90], is beyond the scope of this paper.

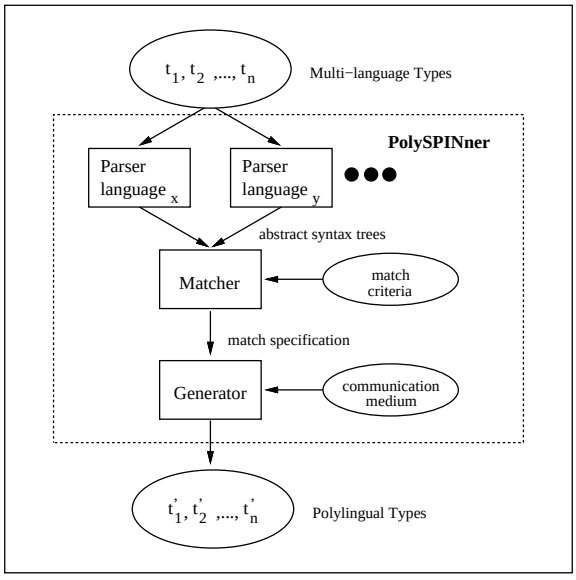

Figure 4: The PolySPINner Architecture

is fairly straightforward, consisting of abstract data types for object types, methods, and parameters. We intend to adopt a more powerful intermediate representation in the near future and are currently examining alternatives in the literature. The PolySPINner prototype incorporates minimal, proof-of-concept parsers for C++ and CLOS.

The Matcher PolySPINner's generic Matcher component is responsible for matching "equivalent" types from different languages. What does it mean for two types in different languages to be equivalent? A strict interpretation could require them to have the same name, abstract specification, and binary representation. Such strictness is unrealistic, however, since interoperability is often desired between software modules whose types are nearly but not exactly equivalent. More lenient, or *relaxed*, matching criteria, are more realistic and useful. Zaremski and Wing [ZW93] have constructed a taxonomy of relaxed type matching criteria within the language ML. Using polylingual system concepts, we have created a prototype extension of their taxonomy that models matching across languages, and PolySPINner currently supports these (and other) matching criteria. In particular, we provide a library of instantiations for the generic Matcher corresponding to common type matching criteria: exact match, Zaremski/Wing matching, intersection match (ignore any methods that the types do not have in common), and others. Since compatibility criteria are represented as C++ functions, users can also create their own using the full power of C++. We provide a library of functions to simplify the process of building these functions, such as generic iterators over all methods of an object type.

The Generator PolySPINner's generic Generator component is responsible for transparently modifying the implementations of types so that they are accessible from multiple languages, without modifying the interfaces of the types. Because we are working only with object-oriented languages, in which the interface and implementation of a type (class)

are separate, we can conveniently modify the implementation and leave the interface unchanged. Thus, to support access from multiple languages, a method's body is modified to query the language arbiter and select the appropriate communicator code.[7] Examples of such modifications as they are performed by our current PolySPINner prototype appear in Section 5.

5 Applying PolySPINner

As illustrated in Section 3, existing approaches supporting the development and maintenance of polylingual applications require some elaborate mechanisms. In contrast, PolySPIN shields the developer of polylingual applications from such complexities. As a point of comparison, we illustrate the use of the PolySPINer prototype by applying it to the example scenario outlined in Section 3.2.

Recall that in this scenario, the new company, Frank Lloyd, wishes to use an application for assigning office rankings (written in CLOS) on employee data objects (maintained in both CLOS and C++). To accomplish this, a software engineer would apply PolySPINner to each of the original class definitions for the personnel data. Figure 5 shows portions the original CLOS and C++ Employee classes. Note that the method interfaces are slightly different for each of the classes. In addition, each of the classes is a subclass of NameableObject, a class defined by PolySPIN, which allows objects to participate in its unified name management mechanism.

```
// C+ Employee Class Interface
class Employee : public NameableObject {
public:
  int Salary ();
  int YearsOfService ();
  Date Birthday ();
  int  Age ();
private:
  int  monthlySalary;
  Date dateStarted;
  Date dateOfBirth;
};
// C+ Employee Class Implementation
int Employee::Salary () {
  return (monthlySalary * 12);
}
int Employee::YearsOfService () {
  int numberOfDays;
  numberOfDays = Today () - dateStarted;
  return (numberOfDays / 365);
}
Date Employee::Birthday () {
  return (dateOfBirth);
}
int Employee:: Age () {
  int numberOfDays;
  numberOfDays = Today () - dateOfBirth;
  return (numberOfDays / 365);
}
```

```
;;; CLOS Employee class
(defclass Employee (NameableObject)
  ((ssn    :accessor ssn
           :type Integer)
   (salary :accessor salary
           :type Integer)
   (years  :accessor years
           :type Integer)
  )
)

;;; CLOS Employee class methods
(defmethod Salary ((this Employee))
  (declare (return-values Integer))
  (salary this)
)

(defmethod YearsOfService ((this Employee))
  (declare (return-values Integer))
  (years this)
)

(defmethod FormatSSN ((this Employee))
  (declare (return-values String))
  (formatter (ssn this) :ssn)
)
```

Figure 5: Original C++ and CLOS Employee Classes

Next the tool determines whether the classes are "compatible" with one another. Compatibility is specified by the user of the tool, as discussed in Section 4. A plausible compatibility specification for this example scenario might be based on an *intersection match* criterion. Under this criterion, two classes are said to be compatible if there exists at least one method from each class such that the method signatures are identical. Specifically, using the classes shown in Figure 5, PolySPINner would deem the C++ and CLOS Employee classes to be compatible since the following methods form an intersection match:

C++ Method	CLOS Method
int YearsOfService()	YearsOfService((this Employee))
int Salary()	Salary((this Employee))

As described in Section 4.2.1, PolySPINner then modifies the implementation for each of the matching methods and generates the necessary communication code enabling the appropriate inter-language references. PolySPINner currently produces code based on constructs provided by PolySPIN [KW96] and the foreign function interface mechanisms of C++ and CLOS. The specifics of these constructs are beyond the scope of the paper (for details see [KW96]). Figure 6 presents example output, in pseudocode, produced for the Salary methods. (Similar code would be generated for the YearsOfService methods.) When the Salary method is invoked on an object, the method first checks the defining language of the object. For example, in the CLOS version, if the object is implemented in CLOS, then the original CLOS logic is executed; otherwise, a foreign function call is made to the corresponding C++ version of the Salary method.

```
int Employee :: Salary () {
  switch (this->language) {
    case CPLUSPLUS:
      // original  code
      return (monthlySalary * 12);
      break;
    case CLOS:
      // call  CLOS method
      int tempSalary =
        ForeignFunctionCallToCLOS (this);
      return (tempSalary);
      break;
    case ...
  }
```

```
(defmethod  Salary ( ( this   Employee ) )
(declare (return-values Integer ))
 (cond ( (EQUAL (language this) CLOS)
           ;;;;; original code
            (salary this)
          (EQUAL (language this) CPLUSPLUS)
            ;;;;; call C++ method
            (foreign-function-call-to-cpp this)
        )
)
)
```

Figure 6: Modified Implementation of Salary Methods

Finally, the generated code (i.e., the modified method implementations) must be compiled and linked with the existing office ranking application. Note that no modifications to the CLOS application or the class interfaces are required by the PolySPINner approach. Only the method implementations are changed.

5.1 A Comparison with ILU

Both PolySPIN and ILU provide support for each of the dimensions in our model. PolySPIN's name management mechanism provides a much richer naming service than does ILU's. More importantly, PolySPIN, unlike ILU, does not impose a language-external type model on application developers. Instead of ISL, PolySPIN permits developers to use their native (CLOS, C++, etc.) type models and hence to define types of shared objects in a style that they find familiar, natural and intuitive. As a result, PolySPIN is better suited than ILU for use in megaprogramming, although it is also very appropriate for the easiest and common cases of interoperability. Finally, CORBA-style approaches, such as ILU, require exact type compatibility between accessor and object, but PolySPIN allows relaxed matches, which are more realistic, particularly for megaprogramming.

[7] In the current PolySPINner prototype, the communicator code consists of foreign function calls for C++ and CLOS, but one could also use message passing, CORBA requests, special-purpose software, etc.

6 Conclusions

Existing approaches to interoperability are not sufficiently seamless. If software modules are required to use invocation mechanisms or type models that are different from those provided in their programming languages, this imposes an unacceptable barrier to integration, particularly in the megaprogramming case. An interoperability approach that forces software developers to modify such fundamental aspects of software modules, that admits only exact match type compatibility, or that is effective only for easiest-case or common-case interoperability is unsatisfactory for meeting the challenges of polylingual interoperability.

In this paper, we have described the PolySPIN approach and PolySPINner toolkit for automating seamless interoperability in polylingual systems. Since this approach evolved from work on name management in persistent object systems, our prototype toolset relies upon features of the Open Object-Oriented Database (Open OODB) [WBT92] and exploits inheritance of name management-associated capabilities in implementing seamless interoperability. The approach, however, does not depend on Open OODB, persistence in general, nor inheritance, so we plan to explore alternative implementation strategies that will broaden the applicability of our toolset.

Now that we have created a generic framework for experimenting with polylingual systems, we have several other future directions in mind for PolySPINner. Rather than having the user supply matching type definitions in several languages, we would like PolySPINner to automate this process, generating corresponding definitions to as great an extent as possible. We also plan to support additional languages, with Ada95 and Java being prime candidates. As the number of supported languages increases, we expect interesting type compatibility issues to arise. Finally, we envision further tools to assist the construction of polylingual systems, such as a polylingual debugger, static analyzer, and structured editor.

Acknowledgments

We thank our three reviewers for valuable comments that improved this paper, and David Hildum and Keith Decker for timely submission assistance.

References

[App94] Apple Computer, Inc. OpenDoc for Macintosh: An overview for developers. White paper, Apple Computer, 1994.

[AR94] Joshua S. Auerbach and James R. Russell. The Concert signature representation: IDL as intermediate language. *ACM SigPLAN Notices*, 29(8):1–12, August 1994. From the Proceedings of the ACM Workshop on Interface Definition Languages, 1994. Originally IBM Research Report RC19229.

[Aue96] Joshua S. Auerbach. Personal communication, March 1996.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Bro94] Kraig Brockschmidt. Ole integration technologies technical overview. World Wide Web URL `http://www.microsoft.com/TechNet/technol/ole/ddjole.htm`, October 1994. Adapted from an article appearing in Dr. Dobbs Journal, December 1994.

[BS92] B. W. Boehm and W. L. Scherlis. Megaprogramming. In *Proceedings of the DARPA Software Technology Conference*, pages 63–82, Los Angeles, CA, April 1992.

[ClPCC+93] Digital Equipment Corporation, Hewlett Packard Company, HyperDesk Corporation, NCR Corporation, Object Design, Inc., and SunSoft, Inc. *The Common Object Request Broker: Architecture and Specification.* Object Management Group and X/Open, 1993. Revision 1.2, incorporated as part of CORBA 2.0.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428, October 1993. Published as *ACM SIGPLAN Notices* volume 28, number 10.

[JS94] Bill Janssen and Mike Spreitzer. ILU: Interlanguage unification via object modules. In *Workshop on Multi-Language Object Models*, Portland, OR, August 1994. (in conjunction with OOPSLA'94).

[JSS95] Bill Janssen, Denis Severson, and Mike Spreitzer. *ILU Reference Manual.* Xerox Corporation, Palo Alto, CA, 1.8 edition, May 1995.

[Kap96] Alan Kaplan. *Name Management: Models, Mechanisms and Applications.* PhD thesis, University of Massachusetts, Amherst, MA, May 1996.

[KW96] Alan Kaplan and Jack Wileden. Toward painless polylingual persistence. In *Seventh International Workshop on Persistent Object Systems*, Cape May, NJ, May 1996.

[MHO96] Mark J. Maybee, Dennis H. Heimbigner, and Leon J. Osterweil. Multilanguage interoperability in distributed systems. In *Proceedings of the International Conference on Software Engineering*, March 1996.

[Pur94] James M. Purtilo. The Polylith software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, January 1994.

[Rei90] Steven P. Reiss. Connecting tools using message passing in the FIELD environment. *IEEE Software*, July 1990.

[SL90] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[WBT92] David L. Wells, Jose A. Blakely, and Craig W. Thompson. Architecture of an open object-oriented database management system. *IEEE Computer*, 25(10):74–82, October 1992.

[WWC92] Gio Wiederhold, Peter Wegner, and Stefano Ceri. Toward megaprogramming. *Communications of the ACM*, 35(11):89–99, November 1992.

[WWFT88] J.C. Wileden, A.L. Wolf, C.D. Fisher, and P.L. Tarr. PGRAPHITE: An experiment in persistent typed object management. In *Proceedings of the Third Symposium of Software Development Environments*, pages 130–142, September 1988.

[WWRT91] Jack C. Wileden, Alexader L. Wolf, William R. Rosenblatt, and Peri L. Tarr. Specification level interoperability. *Communications of the ACM*, 34(5):73–87, May 1991.

[ZW93] Amy Moormann Zaremski and Jeannette M. Wing. Signature Matching: A Key to Reuse. In *Proceedings of the First ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 182–190, December 1993. Also appeared as Carnegie-Mellon technical report CMU-CS-93-151.