Why IDLs Are Not Ideal

Alan Kaplan

Department of Computer Science
Clemson University
Clemson, SC 29634-1906

John Ridgway and Jack C. Wileden

Department of Computer Science
University of Massachusetts
Amherst, MA 01003-4610

Abstract

The dominant approach to addressing heterogeneity, interoperability and legacy software components at present is based on the use of interface description languages (IDLs) such as the OMG/CORBA IDL. We believe that this approach has serious drawbacks. In this paper we outline our objections to the IDL-based approach, then describe ongoing research directed toward producing a superior alternative, which we refer to as the polylingual systems approach. We illustrate both our objections to the IDL-based approach and also our new polylingual systems approach with examples based on the IWSSD common case study.

1 Introduction

Heterogeneity, interoperability and legacy software are increasingly important issues for software developers. Among the reasons for this are the rapidly expanding role of the internet as a source of software components, the extremely distributed and flexible model of computing encouraged by the World Wide Web and the potential for "write once, run anywhere" software inherent in the widespread adoption of Java. Along with this growth in importance has come increasing interest in conceptual and pragmatic foundations for interoperability and other aspects of component-based software development [1].

Various approaches to heterogeneity, interoperability and legacy software have been proposed and used in the past (see [2] for an overview). At present, however, strong industrial forces (OMG [3]) are pushing an approach (CORBA [4]) whose central feature is the use of a "unifying type model" [2, 5] or interface definition language (IDL) to mediate among interoperating components.

While seemingly sensible, IDL-based approaches actually have some significant drawbacks in practice. In brief, because an IDL inherently represents a least common denominator for the type systems of all the languages in which interoperating components might be written, it imposes an interoperability bottleneck. That is, making a determination as to whether two components are compatible by mapping each to an IDL representation, rather than by comparing them directly, necessarily limits the potential for determining compatibility. It also imposes potentially serious overhead on software developers trying to work around its limitations.

We have been working on an alternative, IDL-free (or IDL-invisible), approach to interoperability that avoids these shortcomings, which we refer to as the *polylingual systems approach* [6, 7, 8]. In this paper, we seek to contribute to an improved understanding of the foundations of interoperability by offering an example-driven comparison of some aspects of these two approaches. Specifically, we illustrate both (some of) the shortcomings of the IDL-based approach and also our new polylingual systems approach with examples based on the IWSSD common case study.

2 Polylingual interoperability

In this section, we briefly outline a partial instantiation of the IWSSD common case study [9] that we will use in illustrating the interoperability problem for component-based software systems. We then provide an overview of a simple classification of interoperability scenarios, which is useful for distinguishing three kinds of interoperability problems. Using our partial instantiation of the case study example we then consider how these kinds of interoperability problems might arise in the development of polylingual software systems.

2.1 The meeting scheduler application

The IWSSD common case study specifies a collection of requirements for a meeting scheduler application. Figure 1 shows some of the classes, along with brief descriptions of some of their properties, that might result from a simple object-oriented analysis of this specification.

2.2 Classification of Interoperability Scenarios

The time the decision is made to share software components is a critical issue in assessing the suitability of different interoperability approaches.[1] Three distinct timing

[1]For a more detailed explanation of this issue, as well as other interoperability-related issues, see [6].

Equipment - encapsulates information about equipment needed at meeting (e.g., overhead, projector, etc.).

Location - encapsulates information about locations of meetings (e.g., cities, countries, etc.).

Date - encapsulates a calendar date (i.e., month, day, year) and a time period (starting and ending time). Some operations on Date might include setDate and setPeriod (for setting its values) and lessThanEqualTo (for comparing two Date objects).

BasicAgenda - encapsulates "basic" meeting agenda information for a participant. Some operations on basicAgenda include setTitle (setting a meeting title), insertIncludeDate, getIncludeDates, insertExcludeDate, and getExcludeDates, (for setting and retrieving preferred and non-preferred dates, respectively), and addEquipment and getEquipment (for setting and retrieving necessary special equipment).

SpecialAgenda - a subclass of basicAgenda, which also encapsulates information specific to "special" or priority participants. Operations include indicateLocation, preferredLocation, indicatePriority and getPriority (for setting and getting values for a preferred location and the priority of a meeting participant).

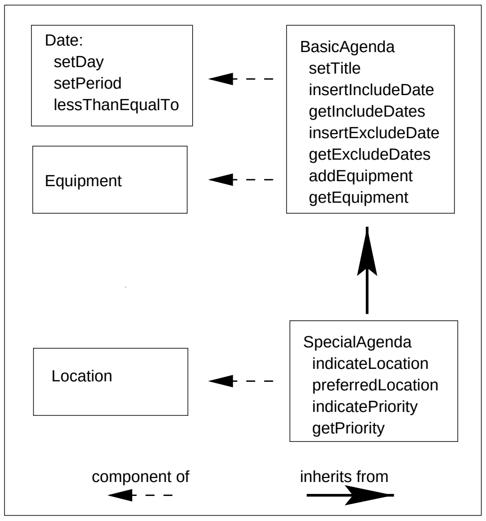

Figure 1: Meeting scheduler classes.

scenarios for interoperability decisions can be characterized by the relationship between the relative times at which the sharing or shared components are developed and the decision to share them is made:

- In the "easiest case," the decision to share is made before any of components have been developed.
- In the "common case," the decision to share is made after some of the sharing components are developed but before others are.
- In the "megaprogramming case," the decision to share is made after the sharing components are developed.

2.3 A polylingual meeting scheduler

Figure 2 shows a *meeting scheduler application* that needs to access and manipulate scheduler-related components (i.e., corresponding to the classes shown in Figure 1) that have been defined in both Java and C++. We refer to

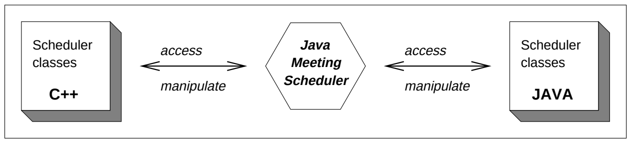

Figure 2: A polylingual meeting scheduler.

the application as *polylingual* since it accesses and manipulates objects that are of the same, or equivalent or compatible, types even though their implementations might be based on different programming languages.

As a step toward understanding the different interoperability problems that a developer might face, we can apply the classification from Section 2.2. In the easiest case, the decision to develop a polylingual meeting scheduler is made before the scheduler-related classes have actually been written. In the common case, a meeting scheduler, perhaps written in C++, may already exist. However, in order to extend its service to also include users who are using a Java-based scheduling system, the C++ application needs to now access Java scheduler-related objects as well as C++ ones. Finally, in the megaprogramming case, both the C++ and Java classes already exist. In this case, a new scheduler application (either in C++ or Java) might then be written so that it can access and manipulate both C++ and Java objects.

In each of these cases, the overall objective is to integrate equivalent or compatible components that have been defined in distinct programming languages. Although various interoperability mechanisms are available that help achieve this objective, we contend that an effective interoperability mechanism should

- allow programmers to use the type systems provided by the language(s) in which they are designing and developing components of their application, and
- have minimal impact on software components, despite the requirement that components need to be shared.

The benefits of achieving these goals are clear, e.g., improved re-use and sharing, improved productivity, reduced development costs, etc. We examine and assess two alternative approaches to this problem in Sections 3 and 4.

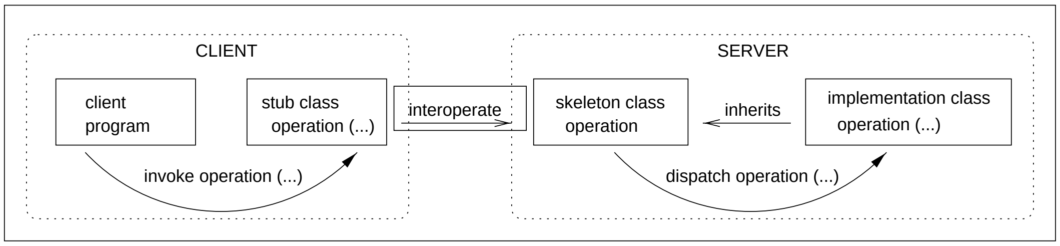

Figure 3: Implementing a component.

3 IDL-based approach

The IDL-based approach (e.g., [4, 10, 11, 5]) has become prevalent in recent years. The basic idea behind this approach is that shared components are described in a universal formalism (namely, an interface description language or IDL). Given an IDL specification for shared components, a tool then generates programming-language-specific source code. For example, a CORBA IDL translator will create code that allows components to interoperate with one another (invoke methods, pass parameters, etc.) as well as creating client and server interfaces for components (referred to as *stubs* and *skeletons*, respectively, in CORBA terminology). The developer is responsible for providing implementations for components. For object-oriented programming languages (such as C++ and Java), the implementation technique relies on polymorphism and dynamic type binding. To implement a server object, the developer defines and implements a class that inherits from a class corresponding to a skeleton, where an implementation for an operation specified in an IDL interface resides in this new class. When a client invokes an operation on an object, it actually calls an operation on the (generated) stub. The stub then invokes the corresponding operation on the (generated) skeleton, which in turn dispatches to the actual operation in the implementation class. Figure 3 illustrates the process.

In the remainder of this section, we illustrate how a representative of the IDL-based approach, namely CORBA, can be used to address each of the scenarios described in Section 2. In the interest of space, we omit discussion of the "common case" and instead focus attention on the two extremes.

3.1 The easiest case

In the easiest case, a developer would first specify the IDL for the scheduler classes, as shown in Figure 4. Although the details of IDL are beyond the scope of this paper, an IDL interface corresponds to a class. IDL allows the specification of operations for interfaces and an interface may inherit from another interface. Finally, IDL provides mappings from primitive CORBA types to language-specific types.

```
module Scheduler {
  interface Equipment {...};
  interface Location {...};
  interface Date {
    void setDate (in long day, in long month, in long year);
    void setPeriod (in long fromHour, in long fromMinutes,
                    in long toHour, in long toMinutes);
    boolean lessThanEqualTo (in Date value);
  };

  typedef sequence<Equipment> EquipmentList;
  typedef sequence<Date> DateList;
  interface BasicAgenda {
    void insertIncludeDate (in Date includeDate);
    DateList getIncludeDates ();
    void insertExcludeDate (in Date excludeDate);
    DateList getExcludeDates ();
    void addEquipment (in Equipment need);
    EquipmentList getEquipment ();
  };
  interface SpecialAgenda : BasicAgenda {
    void indicateLocation (in Location preferLocation);
    Location preferredLocation ();
    void indicatePriority (in boolean flag);
    boolean getPriority ();
  };
```

Figure 4: IDL for easiest case.

Given the IDL shown in Figure 4, a developer would use IDL tools such as "idl2c++" and "idl2java" to generate C++ and Java code, respectively, for the meeting scheduler classes. The developer would then write C++ and Java implementations for these classes, and a meeting scheduler, written in C++ (and/or Java), which could access and manipulate both C++ and Java scheduler objects.

The IDL-based approach is clearly suitable for the easiest case. It provides developers a specification language for describing interoperating components and tools that can automatically generate code in various programming languages. Thus, in polylingual software applications where the decision to interoperate can be made *a priori*, IDL-based approaches appear to facilitate the development of polylingual software systems.

A closer examination, however, reveals several shortcomings. The most serious drawback is that the requirement of an IDL (to specify a shared component) introduces impedance mismatch. Since IDL can be translated into a variety of programming languages (C, SmallTalk, C++, Java, etc.), it inherently represents a least common denominator for all type systems in those programming languages. Thus, important features and capabilities that are found in many type systems (e.g., overloading, constructors, and access visibility) cannot be expressed in IDL.

Another shortcoming is due to the implementation technique used in IDL-based approaches. In situations where interoperability is not required, a class represented in a design generally has a direct correspondence to a class in an implementation. In IDL-based approaches, interoperating components in a design must be decomposed into three separate classes in an implementation — a client class (a stub), a server class (a skeleton) and an implementation class (a subclass of the skeleton class), as shown in Figure 3. Subclasses that exist in an IDL specification present additional difficulties. Specifically, an implementation of a subclass must inherit from its corresponding skeleton class and its parent's implementation class. Such techniques are not only overly complex, but clearly widen the semantic gap between the conceptual specification or design of a component and its implementation and as a result produce code that is arduous to write, difficult to reason about and prone to error.

3.2 The megaprogramming case

In the megaprogramming case the C++ and Java scheduler-related classes already exist. The problem here is to develop a meeting scheduler that can access and manipulate instances of both the C++ and Java classes. Figure 5 shows example C++ and Java classes. (For the sake of simplicity, we do not include all required operations and private class members. In addition, we have intentionally used the same names and signatures where possible.)

In order to use an IDL-based approach in this situation, a developer is faced with the daunting task of translating from C++ and Java class specifications to IDL. The resulting IDL specifications would then need to be compared, mediated, negotiated, etc., in order to produce a single unified IDL specification. This resulting IDL would next be translated into language-specific code as described in Section 3.1. The resulting code could then be used as wrappers around the original classes (from Figure 5).

Clearly such a process, even with the aid of automated translation tools (i.e., tools that translated programming language source to IDL), is relatively complex and error-prone. For example, the Java and C++ classes in Figure 5 clearly have similar interfaces. There are, however, various differences that somehow need to be reconciled. First, the C++ component defining a meeting agenda is actual represented in a class hierarchy (class BasicAgenda is a superclass of SpecialAgenda), while the Java component uses a single class (Agenda). Second, there are operations that are available on the C++ component that are not available on the Java component and vice versa. Third, there are (primitive) types that exist in Java (boolean) that do not exist in C++. Moreover, since IDL represents a least common denominator for type systems, there are various constructs that may exist in an actual class description that do not exist in IDL. For example, overloaded operations (printValue), constructors (BasicAgenda, SpecialAgenda, Agenda), and method access visibility (public, private) are specified in the C++ and Java classes in Figure 5; these constructs, unfortunately, cannot be expressed in IDL.

```
// Sample C++ Classes
class BasicAgenda {
  public:
    BasicAgenda () {...}
    void insertIncludeDate(Date& includeDate){...};
    Date* getIncludeDates (){...};
    char* printValue () {...};
    char* printValue (char* msg) {...};
    private: ...}
class SpecialAgenda : public BasicAgenda {
  public:
    SpecialAgenda () {...}
    void indicatePriority (int flag){...};
    int getPriority (){...};
  private: ...}

// Sample Java Class
class Agenda {
    public Agenda () {...}
    public void setTitle (String title){...}
    public String getTitle (){...}
    public void insertIncludeDate(Date includeDate){...}
    public Date[] getIncludeDates (){...}
    void indicatePriority (boolean flag){...}
    boolean getPriority (){...}
    private ...
}
```

Figure 5: Existing C++ and Java classes

4 The PolySPIN approach

In contrast with the IDL approach the PolySPIN approach does not involve a unifying type model. Rather it is based on directly mapping between types of different languages. The PolySPINNER tool parses class definitions written in different languages, analyzes them, attempts to

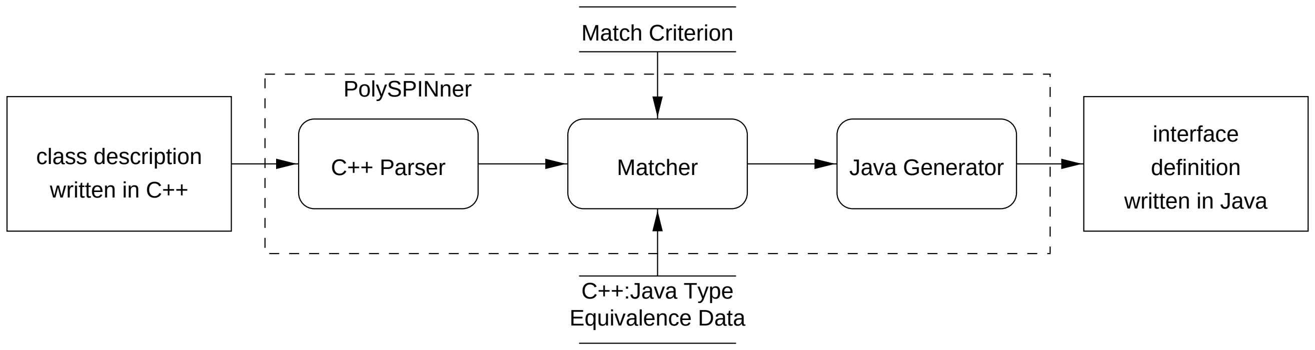
Match Criterion
PolySPINner
class description written in C++
C++ Parser
Matcher
Java Generator
interface definition written in Java
C++:Java Type Equivalence Data

Figure 6: Translating interface definitions with PolySPINNER.

unify them, and produces new code fragments capable of seamless interoperability. The matching criteria used for unification are currently *ad hoc* but we are currently developing an implementation based on a formal model of the matching [8].

4.1 The easiest case

In this section we outline a simple extension to the PolySPIN framework and the PolySPINNER tool to assist in "easy case" polylingual programming. The PolySPINNER tool is already capable of examining two classes written in different languages and verifying that they match.[7] We now propose to allow PolySPINNER to accept an input class written in one language and to output the equivalent class definition in another language. This change will not require adding any new intelligence to PolySPINNER, since the interlanguage type equivalences are already known.

Given the problem as stated earlier, a programmer would generate the interface definition in a chosen language, then run it through PolySPINNER to generate the interface definition in any other required language. This process is illustrated in Figure 6. Effectively we have eliminated the need for an IDL in this case by using one of the programming languages as an IDL. This is clearly easier for a programmer to work with (no need to learn "yet another language"), and it allows the programmer to use any constructs that are mappable between the two languages, rather than being restricted to those that are available in IDL.

4.2 The megaprogramming case

The PolySPIN approach is well-suited, and indeed was originally conceived, for the megaprogramming environment. In this case C++ and Java scheduler-related classes already exist (see Figure 5). The problem is to develop a meeting scheduler that can access and manipulate instances of both the C++ and Java classes. Given this information the programmer would examine the classes in question, decide that the C++ class SpecialAgenda was equivalent to the Java class Agenda, create matching criteria, and run PolySPINNER to verify the matching and create interoperating code. This process is illustrated in Figure 7. In the indicated example, with no additional information, PolySPINNER would identify that an intersection match exists between the indicated classes and generate code that would interoperate seamlessly. Details of this process as it has been carried out using a version of PolySPINNER that accepts C++ and CLOS inputs are described and illustrated in [7].

5 Status and directions

In this paper we have sought to contribute to an improved understanding of the foundations of interoperability by offering an example-driven comparison of some aspects of two interoperability approaches. Specifically, we have illustrated some aspects of both the IDL-based approach and our polylingual systems approach with examples based on the IWSSD common case study.

We are currently developing extensions to the PolySPINNER tool that supports our approach. The existing version takes type definitions in C++ and CLOS, checks their compatibility and produces transparently modified implementations that allow their instances to be accessed from components written in either of the two languages. The new version will add Java to the set of languages that PolySPINNER supports in this way. It will also, as described in section 4.1, allow PolySPINNER to accept an input class written in one of the three supported languages and to output the equivalent class definition(s) in either or both of the other languages.

To provide a principled foundation for the cross-language type compatibility checking implemented in our PolySPINNER tool we have developed an initial version of a formal model of cross-language type matching [8]. Improvements and enhancements to that model are also presently underway. Some of these are necessitated by the addition of a third language (Java) to the set supported by

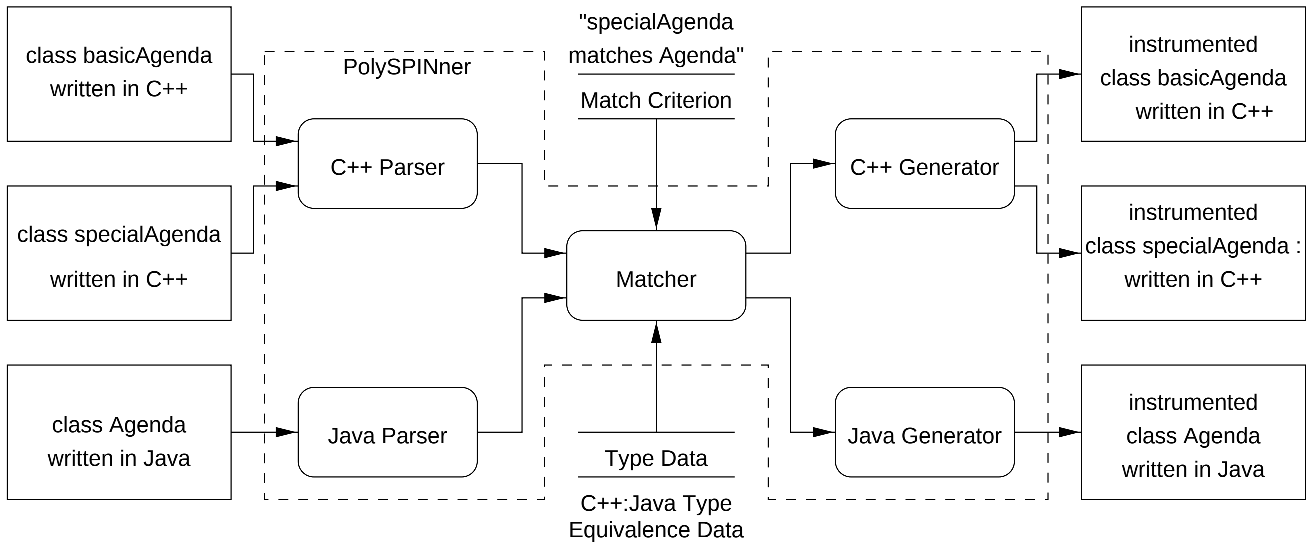
class basicAgenda written in C++
class specialAgenda written in C++
class Agenda written in Java
PolySPINner
C++ Parser
Java Parser
"specialAgenda matches Agenda"
Match Criterion
Matcher
Type Data
C++:Java Type Equivalence Data
C++ Generator
Java Generator
instrumented class basicAgenda written in C++
instrumented class specialAgenda : written in C++
instrumented class Agenda written in Java

Figure 7: PolySPINNER acting on the code fragments from Figure 5.

PolySPINNER. Others seek to expand the kinds of situations to which our approach can apply. For instance, since both CLOS and Java support reflection we now wish to handle reflection in our type compatibility determinations.

In the longer term, we envision extending our approach to encompass both formal foundations and automated support for interoperability that address aspects of heterogeneous, component-based systems going well beyond type definition. We expect, however, that such extensions will share the IDL-free (or IDL-invisible) nature of our existing approach. As the examples in this paper should serve to demonstrate, IDLs are not ideal as a basis for interoperability!

References

[1] Gary T. Leavens and Murali Sitaraman, Eds., *Proceedings of the First Workshop on Foundations of Component-Based Systems*, Zurich, Switzerland, September 1997. http://www.cs.wvu.edu/˜resolve/FoCBS, In conjunction with *European Software Engineering Conference* and the *Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*.

[2] Jack C. Wileden, Alexader L. Wolf, William R. Rosenblatt, and Peri L. Tarr, "Specification level interoperability," *Communications of the ACM*, vol. 34, no. 5, pp. 73–87, May 1991.

[3] Object Management Group, *A Discussion of the Object Management Architecture*, Jan. 1997.

[4] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Aug. 1997, Revision 2.1.

[5] Bill Janssen and Mike Spreitzer, "ILU: Inter-language unification via object modules," in *Workshop on Multi-Language Object Models*, Portland, OR, Aug. 1994, In conjunction with *OOPSLA'94*.

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

[7] Daniel J. Barrett, Alan Kaplan, and Jack C. Wileden, "Automated support for seamless interoperability in polylingual software systems," in *The Fourth Symposium on the Foundations of Software Engineering*, San Francisco, CA, Oct. 1996.

[8] Daniel J. Barrett, *Polylingual Systems: An Approach to Seamless Interoperability*, Ph.D. thesis, University of Massachusetts Amherst, May 1998.

[9] IEEE, *Proceedings of Ninth IEEE International Workshop on Software Specification and Design*, Ise-shima, Japan, April 1998.

[10] Kraig Brockschmidt, *Inside OLE, 2nd Edition*, Microsoft Press, 1995.

[11] R.G.G. Cattell and Douglas Barry, Eds., *The Object Database Standard: ODMG 2.0*, Series in Data Management Systems. Morgan Kaufmann, San Francisco, CA, 1997.