Toward Painless Polylingual Persistence *

Alan Kaplan[†] and Jack C. Wileden

Department of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003 USA

{kaplan,wileden}@cs.umass.edu

PREPRINT

February 29, 1996

Abstract

Heterogeneity in persistent object systems gives rise to a range of interoperability problems. For instance, a given object-oriented database (OODB) may contain data objects originally defined, created and persistently stored using the capabilities provided by several distinct programming languages, and an application may need to uniformly process those data objects. We call such a database *polylingual* and term the corresponding interoperability problem the *polylingual access* problem.

While many of today's OODBs support multiple programming language interfaces (we term such systems *multilingual*), none provide transparent polylingual access to persistent data. Instead, present day interoperability mechanisms generally rely on external data definition languages (such as ODMG's ODL), thus reintroducing impedance mismatch and forcing developers to anticipate heterogeneity in their applications, or depend upon direct use of such low-level constructs as the foreign language interface mechanisms provided in individual programming languages. Using such mechanisms make polylingual access *painful*.

In this paper we introduce PolySPIN, an approach supporting polylingual persistence, interoperability and naming for object-oriented databases. We describe our

*Appeared in Proceedings of the Seventh International Workshop on Persistent Object Systems, Cape, NJ, May, 1996.

current realization of PolySPIN as extensions to the TI/Arpa Open Object-Oriented Database and give examples demonstrating how our PolySPIN prototype supports transparent, *painless* polylingual access between C++ and CLOS applications.

Keywords: Persistence, Interoperability, Polylingual, Name Management, OODBs

1. Introduction

Over the years, as information systems applications have grown larger and more complex, various kinds of *heterogeneity* have appeared in those applications. As a result, individuals and organizations involved in developing, operating or maintaining such applications have increasingly been faced with *interoperability problems* – situations in which components that were implemented using different underlying models or languages must be combined into a single unified application. To aid in overcoming such problems, a range of *interoperability approaches* have been employed. As interoperability problems evolve, due in part to evolution of the underlying models and languages used in information systems applications, interoperability approaches must also evolve.

In applications developed using traditional database technology, there have been two primary sources of heterogeneity. One of these is the need or desire to code different components of an application in different programming languages. The other is the need or desire to make use of two or more different databases in a single application. These have given rise to two corresponding classes of interoperability problems, which we refer to as the *multilingual access* problem and the *multiple database integration* problem.

One of the important extensions to database technology that has appeared during the last decade has been the introduction of persistent object systems (POS). By virtually eliminating impedance mismatch, POS technology can be viewed as a significant evolution of the underlying models and languages used in information systems applications and hence has many ramifications. Among these are new possibilities for heterogeneity and concomitant new interoperability problems, which necessitate the evolutionary development of new interoperability approaches. In particular, a given object-oriented database (OODB) may contain data objects originally defined, created and persistently stored using the capabilities provided by several distinct programming languages. We call such a database *polylingual*. This novel kind of heterogeneity induces new interoperability problems, such as the possibility that an application may need to uniformly process the data objects in a polylingual OODB. We term this interoperability problem the *polylingual access* problem. Existing interoperability approaches provide little or no support for polylingual access, so new approaches must evolve to provide such support.

While many of today's OODBs support multiple programming language interfaces (e.g., ObjectStore [6], GemStone [1]), none provide transparent polylingual access to persistent data. Instead, present day interoperability mechanisms generally rely on external data definition languages (such as ODMG's ODL [2] or CORBA's IDL [8]), thus reintroducing impedance mismatch and forcing developers to anticipate heterogeneity in their applications, or depend upon direct use of such low-level constructs as the foreign language interface mechanisms provided in individual programming languages. In addition, many current approaches require that all the data in a polylingual database be stored using a single common representation, and thus force a substantial amount of data translation to precede their use. Others, while avoiding data translation through the use of so-called "wrapper" techniques, often support only a subset of the manipulations that would be available if the data were accessed from its native language. Because they impose significant additional burdens or restrictions on application developers, we consider such approaches *painful*.

In this paper we focus on the polylingual access problem for object-oriented databases. We begin by discussing heterogeneity and interoperability in OODBs, introducing an example that illustrates various interoperability and heterogeneity issues and identifying some important facets of OODB interoperability problems, particularly the polylingual access problem. We then describe PolySPIN, a framework supporting persistence, interoperability and naming for polylingual object-oriented databases, and its current realization as extensions to the TI/Arpa Open Object-Oriented Database [10]. In addition, we show how PolySPIN can facilitate aspects of interoperability in polylingual object-oriented databases, returning to our earlier example to illustrate PolySPIN's capabilities. It is our assertion that PolySPIN represents an initial step toward *painless* polylingual persistence.

2. OODB Heterogeneity and Interoperability: An Example

As a simple illustration of heterogeneity and interoperability problems in object-oriented databases, consider the following example:

> At Hypothetical University, two colleges have independently developed information systems applications, using object-oriented database technology, for managing personnel information regarding their students and faculty. Although both colleges have in fact utilized the same OODB, the Arts College has built their application on a CLOS API while the Sciences College has built theirs on a C++ API. Figure 1 shows a portion of the C++ schema used by the Sciences College, a portion of the CLOS schema used by the Arts College, and the OODB containing instances of the personnel data object from both colleges implemented in their respective languages.
>
> The central administration at Hypo U would like to develop some applications making use of personnel information from both colleges. Naturally, they cannot hope to convince either college to translate its personnel information to

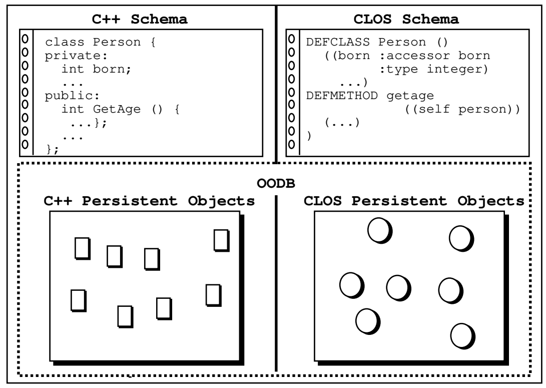


Figure 1: OODB Used By Hypothetical University

a representation corresponding to the other's API. Nor can they expect to convince other colleges, when they develop their own personnel information systems in the future, not to use the API of their choosing (e.g., Ada 95 for the Engineering College, Object-Oriented COBOL for the Business College, etc.). Hence the administrators would like their application to be able to be oblivious to the implementation languages of individual persistent objects. They would also like to be able to employ either navigational access or associative access in processing the personnel information from the various colleges. An example of an OQL-style query (based on [9]) that might be part of a C++ application, in this case seeking candidates for early retirement incentives, is shown in Figure 2. Note that the query should be able to be applied to *all* the personnel data residing in the OODB, i.e., independent of the language used to create the persistent objects.

```
setPerson = ...; // constains Person objects
Set<Person*> result = NULL;
Query {
  result = SELECT *
        FROM Person* matched IN setPerson
        WHERE matched->GetAge() > 45;
}
```

Figure 2: OQL-style Query

Despite the fact that the two college's personnel information schemas are clearly equivalent, existing OODBs, even those such as the TI/Arpa Open OODB that provide multiple APIs, do not support the kind of polylingual access desired by the Hypo U administrators. Several aspects of current OODB technology stand in the way of polylingual access. In the next section, we briefly discuss interoperability goals and issues in general. Later sections then indicate how these goals and issues are addressed in our PolySPIN approach.

3. OODB Heterogeneity and Interoperability: Goals and Issues

Our work on interoperability is, and has for several years [11] been, motivated by a primary concern for the impact of an interoperability approach on applications developers. In our view, among the most important objectives for any approach to interoperability are the following:

- Developers should have maximum freedom to define types of objects that their programs manipulate. In particular, they should always be able to use the type systems provided by the language(s) in which they are designing and developing components of their application.

- Whether a data object is to be shared among an application's components should have minimal impact on the components' developers. In particular, making (or changing) a decision about whether, or with what other components, a data object may be shared should not affect the definition of, or interface to, the object. As a corollary, interoperation should not result in an unnecessary reduction in the ways in which the (now shared) data objects can be manipulated.

Given these objectives, we have noted three major sets of issues regarding interoperability in OODB-based applications. Briefly, these are:

Naming How are objects in the persistent store accessed by applications that wish to interoperate through sharing those objects? Current OODBs typically rely on distinct and often incompatible name management mechanisms for each of the programming languages or application programming interfaces (APIs) they support. This results in disjoint persistent stores segregated according to the language used to define the persistent objects and also leads to inconsistent semantics for the name management capabilities provided by the various language interfaces.

Timing When is the decision to share data objects among an application's components made? This question has a dramatic impact on the suitability of different approaches to interoperability. Three distinct timing scenarios for interoperability decisions can be characterized by the relationship among the relative times at which the sharing or shared components are developed and the decision to share them is made, as illustrated in Figure 3. The salient features of each scenario are:

Easiest case: The decision to share is made before any components are developed. In this case, a common (e.g., IDL) description of the shared data objects can be created prior to development of the components that will share them, language-specific descriptions can be directly created by mapping from the common description, and hence determination of type compatibility is trivial.

Common case: The decision to share is made after one of the sharing components is developed but before any others are. In this case, a common (e.g., IDL) description of the shared data objects can be created by mapping from the language-specific description whose existence predates the sharing decision and then the remaining language-specific descriptions can be directly created by mapping from the common description, so determination of type compatibility is again trivial.

Megaprogramming: The decision to share is made after the sharing components are developed. In this case, common (e.g., IDL) descriptions of the shared data objects can be created by mapping from each of the language-specific descriptions, but determination of type compatibility will then depend upon some kind of comparison of these synthesized descriptions and hence is nontrivial.

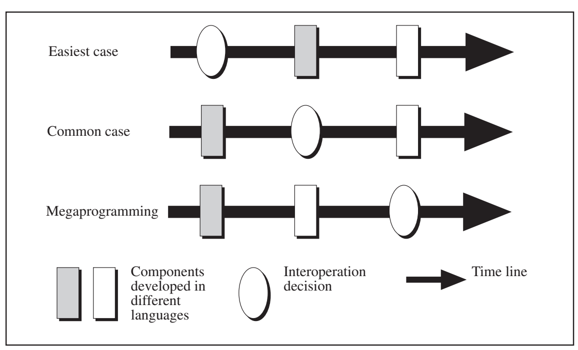


Figure 3: Interoperability Scenarios

Typing How do developers determine whether the types of objects that they wish to share are of compatible types? For object-oriented database technology, most approaches to addressing this question have been based on use of a unifying type model [11], such as the ODMG ODL. While such approaches may suffice for the easiest and common interoperability scenarios, however, they are inadequate for the megaprogramming case. Since, as our example scenario suggests, that case is perhaps the most important and offers the greatest potential rewards, we have focused our research efforts on attempting to handle it.

4. PolySPIN

PolySPIN is a generic, object-oriented framework that unifies persistence and interoperability capabilities in OODBs from a name management-based perspective.[1] PolySPIN, in particular, provides a uniform name management mechanism that not only offers application developers a library of useful abstractions for organizing and navigating object-oriented databases but, as a byproduct, offers an interoperability mechanism providing transparent polylingual access to persistent objects, thus allowing applications to manipulate objects as though they were all implemented in the language of the application. In this section, we begin by briefly describing PolySPIN's approach to name management. Next, we show how an application developer would use PolySPIN to enable interoperability in an OODB. The section concludes with a discussion of the internal features of PolySPIN. Throughout this section, we will refer to the scenario presented in Section 2. as a means of explicating various aspects of PolySPIN. All the PolySPIN features described in this section have been implemented as extensions to the TI/Arpa Open Object-Oriented Database [10], using Sun C++ and the Lucid Common Lisp Object System (CLOS).

4.1 Name Management and Persistence in PolySPIN

While the benefits of orthogonal persistence capabilities offered by OODBs are widely known, relatively little attention has been to paid to how persistent objects should be organized (from an application's perspective) in an OODB. Typically provided by a name management mechanism, existing approaches in OODBs can be characterized as being relatively *ad hoc* and weak [5]. PolySPIN addresses these various shortcomings by providing a uniform, flexible and powerful approach to name management. Although the details of its interface are beyond the scope of this paper, the name management mechanism in PolySPIN allows names to be assigned to objects in binding spaces (where binding spaces are collections of name-object pairs) and names for objects to be resolved in contexts (where contexts are constructed from existing binding spaces) [3]. In addition, binding spaces may be assigned

[1] Name management is the means by which a computing system allows names to be established for objects, permits objects to accessed using names, and controls the meaning and availability of names at any point in time in a particular computation.

names, resulting in the ability to hierarchically organize the name space for objects (similar to directory structures found in almost all modern file systems). Coupled with the persistent store, this approach results in a name-based persistence mechanism where any object (including those in its transitive closure) bound to a name in a binding space reachable from a specially designated root binding space automatically persists. The approach is based on Napier [7], where *environments* correspond to binding spaces. The name management mechanism in PolySPIN is more general, however, since it supports objects defined in multiple languages.

To participate in this mechanism, an object's class definition must inherit from a common base class, designated the NameableObject class. By inheriting from this class, instances of the subclass can be, among other things, named and resolved using the operations supported by the various abstractions that make up the PolySPIN name management mechanism. For example, Figure 4 shows a (partial) C++ class definition named Person, a code fragment showing how a name might be assigned to an instance of Person, and a portion of a persistent store organization based on this approach.

```
// C++ Person class is a
// subclass of NameableObject
class Person : public NameableObject {
public:
  int GetAge ();
...
};
...
BindingSpace* persons;
// Retrieve Persons binding space
// (which is reachble from the Root)
persons = someContext.Resolve ("Persons");
...
Person alan = new Person (...);
...
// Operation signature for Define is:
// BindingSpace::Define (Name&, NameableObject*);
Persons->Define ("Alan", alan);
...
```

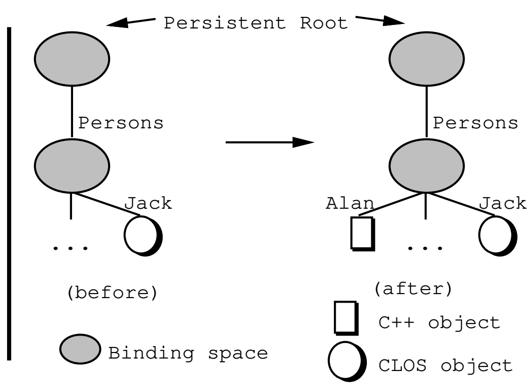


Figure 4: Using PolySPIN's Name Management Mechanism

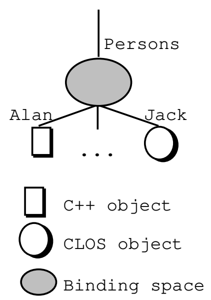
4.2 Name Management and Interoperability

As suggested above, having a class inherit from NameableObject could, and frequently might, be done quite independently of any intention to make objects interoperate. Inheriting from NameableObject does, however, also enable the use of the interoperability capabilities of PolySPIN. First, having a uniform name management mechanism in place results in a language-independent method of establishing visibility paths to persistent objects (i.e., via their assigned names), regardless of the defining language of either the objects or the applications. Second, the name management mechanism serves as a useful place for capturing and recording language-specific information about objects, which can be used to support polylingual access. In particular, once an application has established an initial connection to a persistent object (via its name), the name management mechanism can provide the necessary information permitting the application to create a data path to an object. In other words, when resolving a name of some object (on behalf of some application), the name management mechanism can detect the defining language of the object and initiate the necessary communication medium for manipulating the object. The features supporting this capability are hidden from application developers within the internals of the PolySPIN architecture, which are discussed in Section 4.3.

Given this interoperability mechanism, what is needed to achieve polylingual access is the ability to determine whether two class interfaces defined in different languages can indeed interoperate, and in the event they can, to instrument their implementations (including generating any necessary foreign function interface code) such that the interoperability features of PolySPIN can be employed. As a step toward automating this process, we have developed a tool called PolySPINner. (A more detailed description of PolySPINner can be found in [4].) The overall objective of PolySPINner is to provide transparent polylingual access to objects with minimal programmer intervention as well as minimal re-engineering of existing source code. The current prototype uses an *exact signature matching* rule [12] in determining the compatibility between C++ and CLOS classes. It also encapsulates the foreign function interface mechanism for both Sun C++ and Lucid CLOS, as well as the various internal features of PolySPIN.

To help illustrate how an application developer might use PolySPINner, we return to the scenario presented in Section 2.. In this example, the OODB contains instances of a Person class, where some of the instances have been developed in C++ and others have been developed in CLOS. To take advantage of the naming facilities offered by PolySPIN, we further assume that the original class definitions for each class already inherit from the NameableObject class, as defined in their respective languages. Thus, prior to any decision to interoperate, the objects resident in the OODB might be organized as shown in the left hand portion of Figure 5. In this scenario, the central administration at Hypo U wished to develop an application supporting queries of the kind shown in the right hand portion of Figure 5.[2] Specifically, the C++ OQL-style query shown here is embedded in a fragment accessing both the C++- and CLOS-defined objects and performing the desired query. Note that the implementation of each object is completely transparent to the C++ OQL-style

[2] Although a query is given in this example, an update could be applied to the objects in a similar manner.




```
Set<Person*> setPerson;
Person* p;
Context all = ...;// set context
...
p = (Person *)all.Resolve("Alan");
setPerson.Insert (p);
p = (Person *)all.Resolve("Jack");
setPerson.Insert (p);
...
Set<Person*> result;
Query {result =
    SELECT *
     FROM Person* matched IN setPerson
     WHERE matched->GetAge() > 45;
}
...
```

Figure 5: PolySPIN-based OODB

query. That is, from the application's perspective both objects are instances of the C++ Person class, even though one is obviously implemented as a CLOS object. To accomplish this, the application developer would take the following steps:

1. Apply PolySPINner to the interfaces and implementations of both the Person C++ and CLOS classes. For example, Figure 6 shows the class definitions for the C++ and CLOS Person classes, where the plain face type represents the original source code and the boldface type represents the code generated by PolySPINner.

   (a) PolySPINner first determines whether or not the two Person classes are compatible by comparing the class interfaces. Using an exact match rule, it should be clear that the C++ and CLOS class interfaces shown in Figure 6 are compatible with one another.

   (b) Since the C++ and CLOS Person classes are deemed compatible, the tool next generates foreign function interface code corresponding to each of the operations associated with each of the classes. This permits calls from C++ to CLOS and vice versa. For example, foreign function interfaces corresponding to each of the "GetAge" operations provided by each of the classes must be generated.

   (c) The tool also modifies the implementations of each of the operations defined by a class. The modifications essentially wrap each operation with switching logic that determines the language in which an object is actually implemented and makes the callout to the code generated in the previous step, if need be. For example, if the C++ application invokes the "GetAge" operation on what is in reality a CLOS object, the CLOS "GetAge" operation should be invoked; otherwise the original C++ code implementing the C++ "GetAge" operation should be executed.

2. Re-compile the modified class (method) implementations and the generated source code.

3. Re-link the application.

```
class Person : public NameableObject {
private:
  int born;
public:
  int GetAge ();
};
// GetAge member function
int Person::GetAge () {
  if (this->language == CLOS)
    return
       (__Callout_CLOS_Person_GetAge(this->tidForObject));
  else {
    int result;
    result = 1995 - born;
    return (result);
   }
}
// Callout CLOS Person GetAge
extern "C" int __Callout_CLOS_Person_GetAge (TID this);
// Callout from CLOS into C++
extern "C" int __Callout_CPP_Person_GetAge (TID self ) {
  Person* object = (Person *) TidToCid (self);
  return (object->GetAge());
}
```

C++ Person Class

```
(defclass Person (NameableObject)
  ((born :accessor born
         :type Date
         :initform "MM/DD/YY")
  )
 )
;; GetAge method
(defmethod  GetAge ((this Person))
  (declare (return-values Integer))
  (cond   ( (EQUAL (language this) CLOS)
            (- Today (born this))
          ( (EQUAL (language this) C++)
            (__Callout_CPP_Person_GetAge (tid this)))
  )
);; Callout C++ Person GetAge
(DEF-ALIEN-ROUTINE ("__Callout_CPP_Person_GetAge"
                     __POLYSPIN_CPP_Person_GetAge)
  int   (self TID )
)
;; Callout from C++ into CLOS
(DEF-FOREIGN-CALLABLE
                    (__Callout_CLOS_Person_GetAge
                    (:language :c) (:return-type int))
                    ( ( this TID) )
  (GetAge  (tid-to-cid this))
)
```

CLOS Person Class

Figure 6: Results of Applying PolySPINner

As should be evident, neither the class interfaces nor the persistent data are modified by the PolySPINner tool. Only the class implementations must be re-compiled, along with the generated source code. In addition, the original application remains unchanged, although it must be re-linked to accommodate the changes made to the class implementations. Note that, in Figure 6, some of the PolySPINNer generated code contains references to *CIDs* and *TIDs*. Although transparent to applications, these abstractions enable polylingual access in PolySPIN. In the remainder of this section, we describe these and other internals of PolySPIN that enable polylingual access.

4.3 The Internal Features of PolySPIN

The fact that objects themselves may be implemented in different languages is completely hidden within PolySPIN's name management mechanism. To support this level of transparency in applications, the PolySPIN framework utilizes the following components:

- A three-level object identifier hierarchy.
- A common base class encapsulating language-specific information for transient objects.
- A universal object representation encapsulating language-specific information for persistent objects.

As we illustrate in the remainder of this section, these abstractions, together with their interactions with one another, form a suitable foundation for providing transparent polylingual access.

4.3.1 The Object Identifier Hierarchy

A common solution to the interoperability problem involves converting between data representation formats. For example, to achieve interoperability in the scenario described earlier, it might be possible to simply translate C++ Person objects into CLOS objects (and vice versa). Unfortunately, even when hidden from users and applications, such techniques can be prone to error and computationally expensive, especially for large and complex objects.

An alternative approach involves utilizing object references (or L-values) for identifying objects. This solution has the obvious benefits in terms of efficiency and maintainability. One drawback, however, is that different programming languages use distinct and incompatible object reference mechanisms. For example, native references to objects in C++ can not be interchanged with references to CLOS objects (and vice versa). Instead, a distinct mechanism must be used in a CLOS application to identify a C++ object. Languages supporting garbage collection (e.g., CLOS) present further complications since the value of an object identifier may change over the course of a computation. Although transparent to CLOS applications, garbage collection may cause subsequent accesses by a C++ application using a native CLOS object identifier to result in invalid or dangling references. The addition of persistence yields yet another identifier mechanism that must be managed despite the fact that persistent identifiers are generally hidden from applications. In particular, when an object is designated as being persistent, a persistent identifier is assigned to the object, where the persistent identifier is typically bound to some user-level name. When the object is retrieved from the database, the persistent identifier is first used to locate the object. A reference (i.e., an L-value) must then be created for the object so that the application can access and manipulate the object.

As a step toward relieving application developers from managing separate object identifier mechanisms or building special-purpose ones, the PolySPIN framework maintains a three-level identifier hierarchy, as shown in Figure 7. The hierarchy, in order of increasing lifetime, consists of:

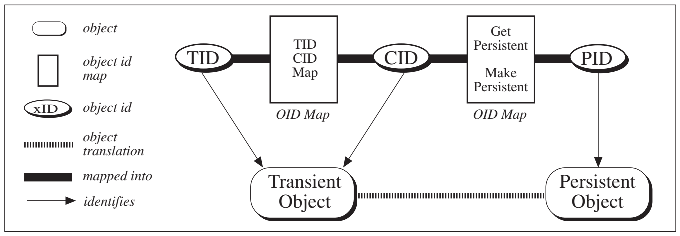


Figure 7: PolySPIN's Three Levels of Identifiers

- A *computation identifier* (or CID), which is an L-value or reference used by applications for identifying, accessing and manipulating objects defined in the same language. A CID for a particular object may change over the the course of a computation, although such changes are intended to be invisible to programmers. An object's (virtual) memory address is an example of a CID.

- A *transient identifier* (or TID), which is an active-computation-unique identifier for an object. Once assigned during some active computation, it is assumed that a TID does not change over the course of that computation's lifetime.

- A *persistent identifier* (or PID), which is a globally unique identifier for a persistent object. When a object is made persistent, a PID is assigned to the object. A PID is assumed to be immutable over the course of the object's lifetime.

In addition, for each language, PolySPIN provides two-way TID↔CID and CID↔PID mapping mechanisms, where the former map can be implemented using a traditional hash table, and the latter map is often supplied by the underlying persistence mechanism provided by the OODB. PolySPIN's maintenance of these identifiers, along with functions for mapping between them, means that applications can simply use CIDs to manipulate objects. Any required identifier translations are handled by PolySPIN. For example, when an application retrieves an object from the persistent store (i.e., via name resolution), a CID identifying the object is returned to the application. As we will show in the following sections, the CID points to an object that, from the application's point of view, looks and behaves as if the object were defined in the same language as the application.

4.3.2 The NameableObject Class

As noted earlier, inheriting from the NameableObject class offers applications developers the ability to use PolySPIN's improved name management mechanism. At the same time, the NameableObject class encapsulates various language-specific information for an object including a defining language, a TID, and various type-related information. As shown in

```
// NameableObject class
class  NameableObject {
private:
  LanguageId language;
  TID tidForObject;
  ClassId classInfo;
public:
// Constructor for NameableObject
  NameableObject() {
    language = C++;
    tidForObject = CidToTid (this);
    ...
}
// Constructor for Person class
Person::Person () {
// Implictly invokes NameableObject
// Then set type information
  classInfo = "Person";
  ...
}
```

Figure 8: The NameableObject Class

Figure 8, values for this information can be computed when an object (derived from NameableObject) is instantiated. For example, the constructors for the C++ Person and NameableObject classes in Figure 8 illustrate how this information is computed and recorded. (The same information is computed in an analogous fashion for CLOS.)

When an operation is invoked on an object, the data maintained by the NameableObject can be used to determine the actual implementation of an object. Since this is hidden from users, however, all instances of the class can be viewed and accessed through a single language interface, even though various instances may in fact be implemented in various languages. Returning to the C++ and CLOS classes shown in Figure 8, the "GetAge" operation for the C++ Person class first checks the value of the defining language for the object. If the object is implemented in C++, then the C++ implementation of the "GetAge" operation is used. If, on the other hand, the object is implemented in CLOS, then the corresponding CLOS operation must be invoked (on the CLOS object). This involves making a call-out to a CLOS function (in this example, using the foreign function interface mechanisms of C++ and CLOS) and passing the CLOS object's TID and a value for the increment parameter. On the CLOS side, the TID is first mapped into its CID value and then the actual CLOS "GetAge" operation is invoked.

4.3.3 The Universal Object Representation

PolySPIN unifies the persistent store by permitting the co-existence of objects implemented in different languages. Furthermore, access to the persistent store is provided by a name management mechanism that is uniformly available across multiple programming languages. To support polylingual access to persistent objects, PolySPIN introduces a level of indirection in bindings between names and (persistent) objects called a universal object representation or UOR.

Like the NameableObject class, a UOR encapsulates various language-specific information about objects, including an object's PID. A UOR is created for an object when that object is assigned a name. In particular, the information stored by the NameableObject is transferred to the UOR. If the object is later designated as being persistent (as described above), then a value for the object's PID is also set in the UOR. Later, when the object is accessed (by resolving the name of the object), the name management mechanism can use the information stored in the UOR to return an appropriate object to the application.

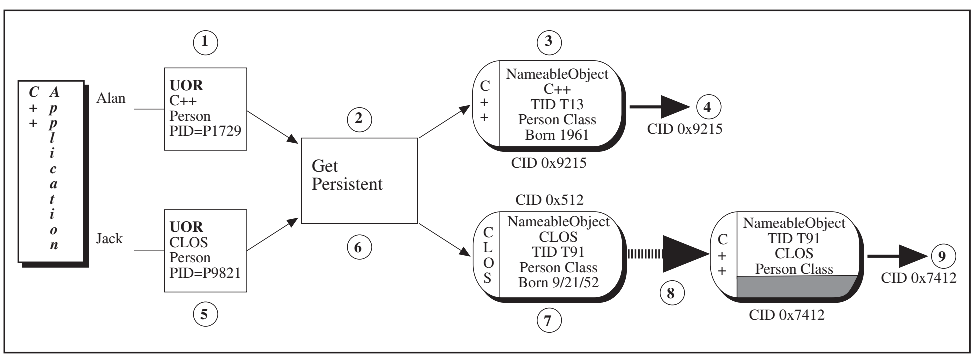


Figure 9: Accessing Objects Via UORs

For example, Figure 9 shows the various tasks PolySPIN performs on behalf of a C++ application accessing two instances of the Person class, where one object, bound to the name "Alan," is implemented in C++ and the other, bound to the name "Jack," is implemented in CLOS. When the object named "Alan" is accessed, PolySPIN examines the UOR for the object (1), determining the object's defining language and its PID. Based on this information, the object is retrieved from the store (2) and a transient, C++ version is constructed (3). Since the language of the application and the language of the object are the same, the object (i.e., its CID) is simply returned to the application (4). When the object named "Jack" is accessed, PolySPIN again examines the UOR for the object (5), determining the object's defining language and its PID. Based on the information, the object is retrieved from the store (6) and a transient, CLOS version is constructed (7). Since the language of the application and the language of the object are in this case different, PolySPIN creates a C++ Person object (8), which acts as a "surrogate" for its corresponding CLOS version. The surrogate's defining language is set to "CLOS" and its TID is set to the TID of the CLOS object. The rest of the data associated with the C++ Person surrogate is simply ignored (as indicated by the shaded portion of the surrogate object in the figure). Finally, the CID of the surrogate is returned to the application (9).

Subsequent accesses to both objects will (eventually) invoke the implementing object (as described in Section 4.3.2). For example, the query shown in Figure 5 calls the "GetAge" operation for both objects. As shown in Figure 6, for the object named "Alan," the original C++ "GetAge" operation is called, while for the object named "Jack," the CLOS "GetAge" operation is invoked. Thus, the query is able to access and process both objects, despite the fact that one object is implemented in C++ and the other in CLOS.

5. Conclusion

In this paper, we have described a new class of interoperability problem for OODBs, namely the polylingual access problem. We have also described PolySPIN, an approach supporting persistence, interoperability and naming in OODBs, and we have shown how PolySPIN can be used to painlessly overcome the polylingual access problem in OODBs. We have briefly described PolySPINner, a tool to help automate the use of PolySPIN and illustrated its capabilities using a simple, but representative, example of a polylingual OODB application. Finally, we have discussed how our approach has been realized in a prototype implementation of PolySPIN and PolySPINner supporting polylingual access between C++ and CLOS, built as an extension to the TI/Arpa Open OODB.

We believe the work reported in this paper represents an important extension to object-oriented database technology. While modern OODBs often provide multiple language interfaces, interoperating among the various languages can be a painful (i.e., cumbersome and complex) process, thus limiting their overall potential. PolySPIN provides transparent, polylingual access to objects (of compatible types), even though the objects may have been created using different programming languages. Thus, application developers are free to work in their native language without precluding the possibility of interoperating with foreign language objects or applications.

REFERENCES

[1] P. Butterworth, A. Otis, and J. Stein. The GemStone object database management system. *Communications of the ACM*, 34(10):64–77, Oct. 1991.

[2] R. Cattell. *The Object Database Standard: ODMG-93*. Morgan Kaufmann Publishers, 1993.

[3] A. Kaplan and J. Wileden. Conch: Experimenting with enhanced name management for persistent object systems. In *Sixth International Workshop on Persistent Object Systems*, Tarascon, Provence, France, Sept. 1994.

[4] A. Kaplan and J. C. Wileden. PolySPINner 1.0: Automating support for interoperability in polylingual software systems. In Preparation.

[5] A. Kaplan and J. C. Wileden. Name management and object technology for advanced software. In *International Symposium on Object Technologies for Advanced Software*, number 742 in Lecture Notes in Computer Science, pages 371–392, Kanazawa, Japan, Nov. 1993.

[6] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, Oct. 1991.

[7] R. Morrison, F. Brown, R. Connor, Q. Cutts, A. Dearle, G. Kirby, and D. Munro. *The Napier88 Reference Manual (Release 2.0)*. University of St. Andrews, Nov. 1993. (CS/93/15).

[8] OMG. Object management architecture guide, revision 2.0. OMG TC Document 92.11.1, Object Management Group, Framingham, MA, Sept. 1992.

[9] Texas Instruments, Inc., Dallas, TX. *Open OODB Query Language User Manual*, release 0.2 (alpha) edition, 1993.

[10] D. L. Wells, J. A. Blakely, and C. W. Thompson. Architecture of an open object-oriented management system. *IEEE Computer*, 25(10):74–82, Oct. 1992.

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

[12] A. M. Zaremski and J. M. Wing. Signature matching, a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2), Apr. 1995.