POLYLINGUAL SYSTEMS:
AN APPROACH TO SEAMLESS INTEROPERABILITY

A Dissertation Presented

by

DANIEL J. BARRETT

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 1998

Department of Computer Science

POLYLINGUAL SYSTEMS:
AN APPROACH TO SEAMLESS INTEROPERABILITY

A Dissertation Presented

by

DANIEL J. BARRETT

Approved as to style and content by:

Jack C. Wileden, Chair

C. Mani Krishna, Member

Barbara Staudt Lerner, Member

J. Eliot B. Moss, Member

David W. Stemple, Member

David W. Stemple, Department Chair
Department of Computer Science

For Lisa, who kept me going.

ACKNOWLEDGEMENTS

First and foremost, I thank my advisor, Jack Wileden. Jack taught me a tremendous amount about interoperability, introducing me to a very interesting area and helping me to develop the research skills to explore it. At the same time, through example, he taught me his effective style of interoperability between mentor and student. I am deeply indebted to him. I also thank both Jack and his wife, Andrea, for many evenings of generous hospitality at their home.

I thank my committee, Professors C. Mani Krishna, Barbara Staudt Lerner, Eliot Moss, and David Stemple, for reading and critiquing this dissertation, and for the ideas they put forth at the proposal and defense.

I thank Alan Kaplan for beginning the work on PolySPIN that eventually led to this dissertation, and for the use of his prototype source code for the original PolySPINner V0. He (and Jack) also contributed significantly to Chapter 5, which we presented at the 1996 Foundations of Software Engineering conference.

I thank my officemate and sometime collaborator, John Ridgway, for many insightful conversations and particularly for his (and Jack's) contributions to Chapter 6. Although the initial paper didn't make it into FOSE, many good ideas were engendered during its writing. John also created the LaTeX2e `umthesis` class, a macro package that adheres to the dissertation typesetting requirements of the University of Massachusetts. As the result of his efforts, this dissertation was typeset with relative ease, and future generations of grad students will be similarly helped. This is no small matter.

I thank the 13 respondents to my interoperability survey (Chapter 3) for taking the time to fill out the survey form.

I thank Jim Purtilo of the University of Maryland and Steve Reiss of Brown University for some enlightening email conversations and for their critiques of my early research in software integration; and David Curtis of the Object Management Group for answering my overly technical questions about CORBA.

I thank Paula Pietromonaco and Howard Schultz and their daughters, Julia and Jennifer, for their generosity in allowing me to stay at their home numerous times while working on this dissertation.

I thank my parents, Judith and Stephen, for inspiring me that education is important; my sister Debbie for beating me to it; and my brother Ben for his encouragement and love.

I thank Robert Strandh, one of my closest friends, and Matt Dwyer, my former officemate, for their strong inspiration that a Ph.D. was within my reach.

I thank my email pals Steve Hahn, Cheryl Carnahan, and Jonathan Roberts for a steady stream of encouragement toward my degree.

Finally, I thank the authors of the many free software packages used to prepare this dissertation: the Free Software Foundation (GNU Emacs, GNU C++ compiler and class library), Donald Knuth (TEX), Leslie Lamport (LATEX), Vern Paxton (Flex), Robert Corbett (Bison), John C. Mallery (CL-HTTP), and Gertjan Kersten (Lucid port of CL-HTTP).

ABSTRACT

POLYLINGUAL SYSTEMS: AN APPROACH TO SEAMLESS INTEROPERABILITY

FEBRUARY 1998

DANIEL J. BARRETT

B.A., UNIVERSITY OF PENNSYLVANIA

M.S.E., THE JOHNS HOPKINS UNIVERSITY

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Jack C. Wileden

If software components are written in different programming languages, communication between the components can become problematic. The study of *multilanguage interoperability* attempts to bridge the communication barriers that arise between software components due to language differences. Many approaches to multilanguage interoperability have been proposed and implemented, such as remote procedure calls, message-passing, foreign type systems (e.g., CORBA), and databases with application programming interfaces for different languages.

A significant shortcoming of such approaches is their lack of *seamlessness*, or transparency of interoperability. Components may require heavy modification in order to interoperate via these approaches; and after integration, the components

may be riddled with interoperability-based code, adversely affecting their readability, maintenance, and cohesion.

This dissertation proposes a far more transparent approach to interoperability, called the *polylingual systems* approach, in which, under certain assumptions, software components need not be modified in order to be made interoperable. The approach is explained using a general framework, called PolySPIN, in which issues of multilanguage interoperability are addressed through the use of formal type theory, statistical analysis, experimentation, and comparison to other approaches. The advantages, disadvantages, and tradeoffs of PolySPIN are examined in detail. Finally, an implementation of PolySPIN, called PolySPINner, provides automated support for creating polylingual systems, and it is compared with CORBA, the current reigning approach to interoperability in industry.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

Computer software is written in many programming languages. This diversity occurs for numerous reasons. Advances in programming language theory and compiler technology have led to new, increasingly powerful languages that replace their predecessors. Political and managerial agendas have led to the spread and adoption of various languages in the workforce. Many special-purpose languages have been designed in the course of academic research. And there's always personal choice: software developers are individuals and simply prefer some languages over others.

Today's proliferation of programming languages was inevitable but is also desirable because different languages are best suited for different purposes. For example, a large application consisting of multiple threads of control might be best served by a language with a tasking construct, such as Ada. A smaller, lighter-weight application might be best implemented in C or even a UNIX shell script. In addition, certain problems are best solved by different programming paradigms, such as imperative, functional, logic, or object-oriented programming.

Because of this diversity of language, programs written in distinct languages are sometimes required to communicate with one another. This is a serious, practical issue. Consider that vast amounts of *legacy code* is found in industry, often written in languages that have become unpopular or obsolete, making modification or extension of the code difficult. The cost of rewriting this legacy code in a modern or more desirable language may be excessive compared to the benefits expected to be derived from the translation. So, software developers sometimes attempt to create extensions

to legacy code, written in a different language, and somehow bridge the gap between the languages.

The issue of "bridging the gap" is known as *multilanguage interoperability*, or just *interoperability*.[1] A familiar medium for interoperability is the World Wide Web, linking not only documents but also programs written in Java, Perl, Lisp, and other languages. A more intricate example is a large banking system: account management software written in the 1970s and running at the bank, PC banking software distributed to the account holders to run at home, investment software used by the bank to invest its money, loan-tracking software, and so on, all written at different times in different languages, and all needing to share data.

Interoperability is still an unsolved problem in general, though many approaches have been considered and implemented. Common file formats, from ASCII text to complicated binary records, allow applications to share data via the file system or via UNIX-style pipes [6]. Remote procedure calls [70] appear as ordinary procedure calls in source code but actually cross address spaces to communicate with other programs. Message-passing systems, such as FIELD [61] and its derivatives, broadcast informational messages to any applications that have signed up to listen. Large initiatives today push interoperability "solutions" such as the Object Management Group's CORBA [55] and Microsoft's OLE [15], adding political confusion to an already technically difficult area.

An important problem with most approaches to interoperability is that they lack transparency. Software developers are forced to perform additional, time-consuming steps to accommodate the interoperability mechanism. For example, programmers might need to translate their data structures into a foreign type system, possibly

[1] Interoperability problems arise also in systems written in a single language—for example, systems that span several hardware platforms or several dialects of the same language. This dissertation focuses on multilanguage interoperability, which we will henceforth refer to simply as "interoperability."

losing type expressiveness along the way. In addition, the application code might become laced with references to the interoperability mechanism, not only decreasing readability and increasing coupling in the code, but also making it difficult to replace the interoperability mechanism in the future, should a better one come along.

This dissertation is about a technique for increasing the transparency of interoperability, making it less visible, or even invisible, to application programmers.

1.1 Motivation

Let's draw an example from the banking system just discussed. Suppose the bank has a program called **Money** that tracks the whereabouts of all money handled by the bank. This program needs to know about consumer accounts, loans, investments, and even the bank employees' salaries. Let's consider just one of those issues, consumer accounts, and say that the bank has another program, called **Accounts**, for tracking them. Let us suppose that because of political differences at the bank, **Accounts** is written in C++ and **Money** is written in CLOS (the Common Lisp Object System) [26]. Somehow, these two programs must communicate across the language barrier. **Money** needs to know (from **Accounts**) how much money is held in consumer accounts, and **Accounts** needs to know (from **Money**) whether the bank has sufficient funds to cover the withdrawals that **Accounts** controls.

Both **Money** and **Accounts** have a notion of what an "account" is: that is, they represent an account as a data type. To **Money**, an account is a CLOS class, such as the one in Figure 1.1. To **Accounts**, an account could be given in C++ as in Figure 1.2. Each application manipulates accounts represented by instances of their respective classes.

Suppose that `big-total`, the CLOS function in Figure 1.3, is part of the **Money** program. It examines a list of bank accounts to calculate the total amount of money they hold. Since `big-total` is written in CLOS, it can access bank accounts only

```
(defclass consumer-account (...) (...))
(defmethod account-balance ((acc consumer-account)) ...)
```

Figure 1.1. A bank account represented in CLOS by **Money**.

```
class account {
  public:
    void Withdraw(double amount);
    void Deposit(double amount);
    double Balance();
    Boolean Overdrawn();
};
```

Figure 1.2. A bank account represented in C++ by **Accounts**.

if they are represented as CLOS `consumer-account` objects. This is unfortunate, because if `big-total` could directly access C++ `account` objects, it could provide interoperability between the two programs. This is a particularly tempting possibility because `account-balance`, the one method invoked by `big-total`, has an almost exact analog in the C++ `account` type: `Balance`. If only the CLOS `big-total` function could directly invoke the C++ `Balance` method, it could operate transparently on both kinds of account objects without regard for their language.

Interoperability is rarely accomplished so transparently, however. More commonly, one of several general techniques is used:

```
(defun big-total (accountlist)
  (cond ((empty accountlist) 0)
        (t (+ (account-balance (car accountlist))
              (big-total (cdr accountlist))))))
```

Figure 1.3. CLOS code fragment from **Money** to calculate the total money in a list of accounts.

- A central data repository, such as a file or database, could be used by the two applications for storing and retrieving shared data. Such an approach can be extremely inefficient, error-prone, and time-consuming to implement, and it does not provide transparent compatibility between the two similar types `account` and `consumer-account`.

- A low-level interlanguage communication mechanism, such as foreign function calls (FFC) or remote procedure calls (RPC), could shuttle the data directly between the applications. Bridging the gap between complex, abstract data types, such as bank accounts, and the primitive types supported by most FFC and RPC mechanisms, however, may require enormous work.

- Both applications could use message-passing within a framework such as FIELD. This involves the same difficulties as the FFC and RPC mechanisms, since types in both applications must be marshaled and unmarshaled as messages, and also requires the applications to be modified to pass messages.

- Using an approach such as CORBA, which has an object-oriented intermediate representation for types, one could translate the application types into CORBA's representation to make them "compatible" with each other. This requires application developers to learn yet another language (the CORBA interface definition language, IDL) and to restrict their programming language types to use only features that IDL is capable of representing.

Clearly a more transparent and easy-to-use technique is needed. The technique described by this dissertation, PolySPIN,[2] provides a level of transparency that allows programmers to work exclusively with abstract types defined in their chosen languages. PolySPIN provides a framework in which object-oriented method calls

[2]PolySPIN = "**POLY**lingual **S**upport for **P**ersistence, **I**nteroperability, and **N**aming."

```
account acc;
while (there are accounts left to process) {
  acc = GetNextAccount();
  if (acc.Balance() <= 0)
    PrintLetter(acc);
}
```

Figure 1.4. C++ pseudocode for iterating through a set of accounts.

can cross language boundaries seamlessly, and it does so without a need to modify the application programs. The `big-total` function of Figure 1.3 could be a point of interoperability between **Accounts** and **Money** as is, without modification, accessing both C++ and CLOS objects.

As another example, consider the C++ pseudocode in Figure 1.4 for iterating through a set of accounts. The PolySPIN approach would allow this code fragment to be applied as-is to a set of "account" objects implemented in different languages, such as our C++ `account` and CLOS `consumer-account` objects. If `acc` happens to be a C++ object, `acc.Balance()` will be invoked normally. If `acc` is a CLOS object, the `Balance()` method call will transparently cross the language boundary to call the CLOS object's `account-balance` method. Thus, both C++ and CLOS programs would transparently access objects of both languages, as depicted in Figure 1.5.

In our simple banking example, the C++ `account` and CLOS `consumer-account` types had some obvious similarities. The issue of compatibility between types of different languages, however, is deep and complex. Some kinds of compatibility are trivial or well-studied, such as converting C++ `int` values into CLOS `Integer` values. When comparing abstract data types, however, one faces the problem of *semantic heterogeneity*[3] [67], in which types with similar structure may have significant se-

[3]Also known as the *database merge* problem.

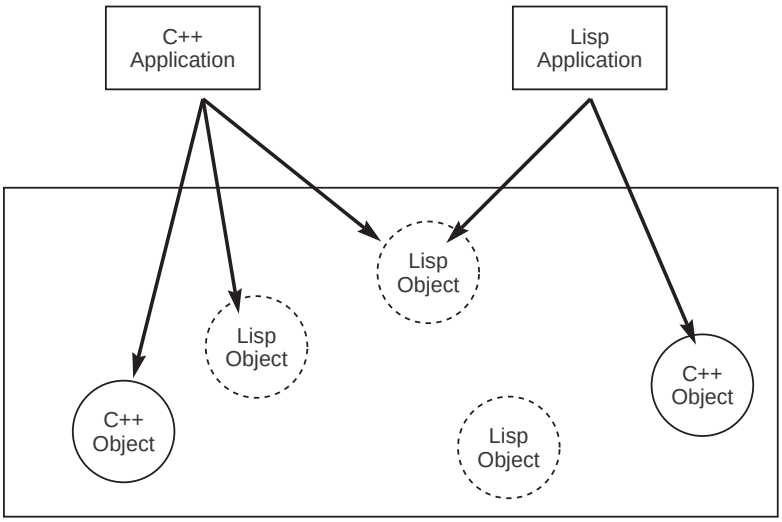


Figure 1.5. Polylingual interoperability.

mantic differences. Semantic heterogeneity is considered an intractable problem in general, as its solution requires tremendous domain knowledge.

1.2 Polylingual Systems

Informally, a *polylingual system* is a collection of software components, written in diverse languages, that communicate with one another transparently. More specifically, in a polylingual system it is not possible to tell, by examining the source code of a polylingual software component, that it is accessing or being accessed by components of other languages. All method calls, for example, appear to be within a component's language, even if some are interlanguage calls. In addition, no IDL or other intermediate, foreign type model is visible to developers, who may create types in their native languages and need not translate them into a foreign type system. Such transparency is highly advantageous, since developers needn't write their programs specifically to interoperate and may focus more closely on each program in isolation, reflecting the software engineering principle of *separation of concerns*.

This dissertation discusses interoperability in polylingual systems at three levels of detail:

- **Polylingual systems**: their definition and theory.
- **PolySPIN**: a particular approach to building polylingual systems.
- **PolySPINner**: an implementation of PolySPIN.

1.3 Research Goals

1.3.1 History

The study of polylingual systems represents a new area of research within the field of interoperability. It does not build on a single area but draws from many related areas, such as type theory, databases, object-oriented programming, and software integration.

Polylingual systems research began as an incidental outgrowth of Alan Kaplan's doctoral dissertation on naming [40]. Having created an interlanguage naming facility for the TI/ARPA Open Object-Oriented Database [72], Kaplan hypothesized that the naming facility could facilitate transparent interoperability among applications of distinct languages if they accessed their objects through the Open OODB. He created a minimal, proof-of-concept implementation of the idea, tried it with one or two small examples, and then moved on to other research.

This was the state of polylingual systems research before this dissertation. Although an intriguing idea, polylingual systems had no formal definition, no associated theory, no rigorous comparison to other approaches to interoperability, and just the beginnings of an implementation. It was also strongly tied to the Open OODB, or at least to object-oriented databases.

1.3.2 Scope of This Research

This dissertation creates a foundation for the study of polylingual systems. Rather than scrutinizing one or two specific questions—impractical in a largely undefined area—it explores the subject broadly, expanding its frontiers in multiple directions. The result is a solid body of knowledge that permits polylingual systems to be defined, reasoned about, compared to other approaches, and built. In the process, this research also raises many interesting questions, suggesting that polylingual systems research is a fruitful area for future research.

The directions of research explored by this dissertation include:

- *Terminology and basic concepts.* Polylingual systems are defined, and a vocabulary is established for discussing them.

- *Type theory.* Initial research on polylingual systems lacked a formal foundation. This dissertation provides a formal framework, based on object-oriented type theory and monolingual type matching, for describing and reasoning about polylingual systems. Particular attention is given to the issue of *relaxed match* between types of different languages.

- *Comparison.* The basis for polylingual systems is a technique different from any other approach to interoperability, but no rigorous comparison to other approaches had previously been attempted. This dissertation establishes a set of criteria for comparing interoperability approaches, provides statistical evidence of its utility and relevance, and uses the criteria to compare the approach to CORBA, the most popular interoperability approach in industry today.

- *Experiments.* The original, proof-of-concept implementation had many limitations and was tested only on toy applications. This dissertation research has led incrementally to several new implementations, each with greater capabilities and robustness, that have been tested on a large third-party application.

- *Independence.* Formerly, this research was tied to the study of naming, persistence, and object-oriented databases. Interoperability in general, however, need not be related to these other fields. This dissertation makes an initial exploration of polylingual interoperability in the absence of persistence, revealing previously hidden assumptions and raising many questions.

Interoperability is a challenging area of study. Any two large, independently-developed applications are likely to be written in entirely dissimilar styles, particularly if they are in different languages. Making two such applications cooperate may be extremely difficult since, in a sense, one must reconcile the differences between two software developers' imaginations. The challenge of interoperability research, then, is to see how much can be done without crossing the line into semantic heterogeneity, and yet not limit the research to trivial examples.

1.3.3 Research Hypotheses

This dissertation presents justification for the following research hypotheses:

Hypothesis 1 *Polylingual systems are a viable approach to interoperability.*

This will be demonstrated analytically, by showing that polylingual systems address many important aspects of interoperability, and empirically, by using PolySPINner to integrate independent software components gathered from several sources.

Hypothesis 2 *PolySPIN is superior to other approaches to interoperability, particularly CORBA, in transparency, ease of use, type expressiveness and safety, and megaprogramming capability.*

This will be demonstrated analytically, by comparing features of PolySPIN and CORBA, highlighting each approach's strengths and weaknesses.

Hypothesis 3 *PolySPINner provides interoperability with acceptable performance, that is, performance approximately equal to that of the low-level communication substrate on which PolySPINner is built (i.e., foreign function calls, RPC, etc.).*

This will be demonstrated empirically.

1.4 Assumptions

The following assumptions hold throughout this dissertation.

- This research is currently applicable only to programming languages that are object-oriented. This is for two reasons. First, PolySPIN requires the explicit separation of the specification and implementation of a software component (e.g., the interface and implementation of an object type), and the encapsulation of operations (methods) within types. While there exist non-object-oriented languages with these features, such as Ada83, object-oriented languages are more commonly used. Second, PolySPIN addresses inheritance as found in object-oriented languages.

- Software components have some means of identifying or locating objects of other languages, for example, through the use of an interlanguage name server. Any approach to interoperability requires this capability, whether it is visible to components or not. Otherwise, it would be impossible for components of different languages to contact one another and achieve interoperability at all.

- All source code examined by our tools is syntactically correct. Since the source code can simply be compiled to check this before our tools are applied, this is a reasonable assumption.

1.5 Road Map

This dissertation is organized roughly into four parts.

I. *Background and Motivation.* Chapters 2–4 introduce terminology, develop a means for comparing approaches to interoperability, and survey related work.

II. *Analytical Research.* Chapters 5–8 provide an overview of polylingual systems, introduce PolySPIN and PolySPINner, and develop formal frameworks for interlanguage type compatibility, and compare PolySPIN with CORBA.

III. *Empirical Results.* Chapters 9–11 describe the implementation of PolySPINner and demonstrate its application to software components obtained from diverse sources.

IV. *Technical Details.* Three appendices elaborate on more general information presented earlier in the text.

CHAPTER 2

TERMINOLOGY

Interoperability has no accepted, standard terminology.[1] Therefore, it is necessary to define terms to be used throughout this dissertation.

2.1 Software Components and Interoperability

A *software component* is a program, a subprogram, a set of subprograms considered as a unit (e.g., a module, class or library), an object, or a set of objects considered as a unit (e.g., a composite object). We abbreviate the term as *component* when it is unambiguous to do so.

Interoperability denotes interaction, or the potential for interaction, among software components written in different programming languages. The components are said to *interoperate* with one another, or to be *interoperable* with one another.

An *interoperability approach* is a general technique by which software components may interoperate. Examples are remote procedure call (RPC), use of a shared database, CORBA [55], and PolySPIN. An *interoperability mechanism* is an implementation of an interoperability approach. For example, Matchmaker [38] is an RPC implementation, the TI/DARPA Open Object-Oriented Database [72] is a database implementation, Orbix [7] is a CORBA implementation, and PolySPINner is a PolySPIN implementation.

Associated with an interoperability mechanism are two kinds of software developers: those who build the interoperability mechanism, and those who use it to integrate

[1] The author has made some progress on terminology in the field of software integration [8].

software components. In this dissertation, an *interoperability designer* is the builder or maintainer of an interoperability mechanism. An *interoperability engineer* uses an interoperability mechanism to make a set of software components interoperate.

2.2 Programming Languages

The source code of a software component is written in one or more languages. Given a software component c, we define $L(c)$ to be the set of languages present in the source code of component c. This is always a non-empty set. The number of elements of $L(c)$ is denoted $|L(c)|$ and is termed the *size* of $L(c)$.

Let $C = \{c_1, c_2, \ldots, c_n\}$ be a set of software components. The languages of C, denoted as $L(C)$, is defined as the union of the languages of $c_1, c_2, \ldots, c_n$,

$$L(C) = \bigcup_{i=1}^{n} L(c_i)$$

If $|L(C)| = 1$, set C is called *monolingual*. If $|L(C)| > 1$, C is called *multilingual* or *multilanguage*. The definition of "polylingual" is deferred to Chapter 5; informally, it implies multilingual plus transparency of interoperability.

When discussing the languages of software components, statements can easily be ambiguous. Consider the sentence, "Programs $p_1, p_2, \ldots, p_n$ are written in three different languages." This sentence is unclear because it does not specify whether any individual program, p_i, is written in exactly one language or may be written in multiple languages. More precisely, we do not know whether $|L(p_i)| = 1$ or $|L(p_i)| \geq 1$, for $1 \leq i \leq n$. The following phrases are defined to avoid such imprecision.

- We say that the n components of C are written in *distinct* languages if:

$$|L(c_i)| = 1 \qquad i = 1 \ldots n$$
$$L(c_i) \neq L(c_j) \qquad i, j = 1 \ldots n,\ i \neq j$$

That is, each component is written in precisely one language, and no two languages are the same. This implies $|L(C)| = n$.

- We say that the n components of C are written in *different* languages, or in *a variety of* languages, if:

$$|L(c_i)| = 1 \qquad i = 1 \ldots n$$
$$|L(C)| > 1$$

That is, each component is written in precisely one language, but the languages need not be distinct. There are, however, at least two languages represented.

- We say that the n components of C are written in *multiple* languages if:

$$|L(c_i)| \geq 1 \qquad i = 1 \ldots n$$
$$|L(C)| > 1$$

In this case, the languages are not necessarily distinct, nor is each component necessarily written in a single language; however, at least two languages are represented.

2.3 Types

Type theorists use several different formalisms for describing types, as will be discussed in Section 4.4. In this dissertation, we use the following terminology drawn

from Bruce [16] and Abadi and Cardelli [1]. A *type* is a set of attributes (fields) and a set of method specifications (i.e., signatures with return types).[2] The set of public method specifications is sometimes called the *interface* of the type. A *class* is a type accompanied by implementations of its methods. An *object* is an instance of a class.

For example, in C++, the following is the declaration of a type:

```
class Person {
  private:
    String *name;
  public:
    String Name();
};
```

the following is the declaration of a class:

```
class Person {
  private:
    String *name;
  public:
    String Name();
};
String Person::Name() {
  return name;
}
```

and the following is the declaration of an object:

---

[2]Types in some object-oriented languages also include the declaration of exceptions. The object-oriented type theory models that are the formal basis of this dissertation do not address exceptions. Exceptions are discussed in Section 12.2.2 on future work. Visibility specifications, such as public and private, are also not part of the definition of a type in this dissertation, but our toolset (Chapter 9) can take them into account when comparing types.

```
Person p;
```

Type t' is a *subtype* of type t if an instance of type t' can be used in any context that expects an instance of type t [16]. Subtyping is represented by the <: infix operator. The notation $A<:B$ indicates that type A is a subtype of type B. The *subsumption rule* of subtyping states: if a is of type t', and $t'<:t$, then a is also of type t.

Types, like components, are represented using programming languages. Given a type t, we define $L(t)$ to be the single programming language in which type t is represented.[3] Unlike components, which may be written in multiple languages, a type is always written in a single language: $|L(t)| = 1$.

Recall that in Section 2.2 we defined the phrases "distinct languages," "different languages," and "multiple languages" as applied to software components. For types, we make analogous definitions. Let T be a set of types $\{t_1, t_2, \ldots, t_n\}$.

- We say that the n types in T are written in *distinct* languages if:

$$L(t_i) \neq L(t_j) \qquad i, j = 1 \ldots n,\ i \neq j$$

  That is, no two languages are the same, implying $|L(T)| = n$.

- We say that the n types in T are written in *different* languages, or in *a variety of* languages, if

[3]Here, $L(t)$ is defined as a *single* language, whereas in the corresponding definition for components, $L(c)$ is a *set* of languages. If consistency between these definitions is required, consider $L(t)$ to be a set containing a single programming language.

$$|L(T)| > 1$$

That is, the languages need not be distinct, but at least two languages are represented.

- There is no analogous definition for "types written in multiple languages," since a type is always written in exactly one language.

CHAPTER 3

AN INTEROPERABILITY MANIFESTO

One goal of this dissertation is to compare our new interoperability approach, PolySPIN, to other approaches, notably CORBA. Before we can discuss the merits of one approach to interoperability, however, we must have standards by which such approaches are rated. Diverse interoperability approaches have been proposed and implemented with different features and limitations. But what are criteria for a "good" approach? Phrases like "bug-free" and "easy to use" come to mind but they lack precision. This chapter aims to provide structure for comparing and contrasting approaches to interoperability, for the purpose of evaluating PolySPIN in subsequent chapters.

A basic premise of this dissertation is that a good interoperability approach should impose minimal effort, and few or no constraints, on software developers beyond those imposed by the independent development of the individual software components. In order for components to be made interoperable, interoperability engineers should *not* have to:

- Manually translate components' code or data structures into other languages.
- Modify legacy code significantly.
- Write low-level code for interlanguage communication (marshaling and unmarshaling parameters, accessing sockets, etc.).
- Treat every interoperating application as unique, without reuse of previous interoperability infrastructure.

With this in mind, I propose the following list of features—an Interoperability Manifesto—that should be addressed by any interoperability approach. These features are the criteria by which interoperability approaches will be evaluated and compared in this dissertation. Informally, they are:

1. Type Issues

   (a) *Expressiveness*: Support for a wide range of data types that can be communicated among interoperating components.

   (b) *Safety*: A guarantee that data (objects) from one language will be sensible when interpreted by components of other languages.

   (c) *Compatibility*: Strictness (or not) of compatibility required between data types of interoperating software components, in order for meaningful communication of data to take place.

2. Ease of Use for the Interoperability Engineer

   (a) *Diversity*: Support for a wide range of programming languages.

   (b) *Transparency*: Invisibility of the interoperability mechanism to the developer.

   (c) *Nonintrusiveness*: The amount of modification necessary to existing components—ideally, none at all—in order for the components to be made interoperable.

   (d) *Automation*: The extent to which components can be made interoperable without human intervention.

   (e) *Performance*: Time and space efficiency of interlanguage communication.

3. Ease of Use for the Interoperability Designer

   (a) *Extensibility*: Ease of adding interoperability support for a new programming language.

I do not claim that the above features are comprehensive, just that they are important. In the following sections, I will present an argument to this effect: first by exploring each feature in detail, and then by presenting the results of a survey of software developers.

3.1 Type Issues

3.1.1 Type Expressiveness

Early attempts at interoperability focused on primitive data types, such as integers, allowing them to be interpreted compatibly by components of distinct languages. The interoperability issues were at the representation level [74], in that the proper ordering and interpretation of bits and bytes were the primary concerns. An example approach is remote procedure call (RPC).

Support for only primitive data types is not sufficient for modern software, which commonly uses more complex types. CORBA [55], for example, supports structured types such as arrays and records. Few approaches support interoperability of pointer-based data structures or abstract data types.

In this dissertation, the type expressiveness of an interoperability approach will be rated according to its support for four kinds of types, from simple to complex:

1. Primitive types
2. Structured types
3. Pointer types
4. Abstract data types

A good interoperability approach should support all of these; if it doesn't, the more the better.

3.1.2 Type Safety

Type safety in a software system is the property that every operation invoked on an instance of type t is defined on type t, for all types t in the system. Type safety is generally recognized as a necessary property for correct software.

The definition of type safety in multilanguage interoperability is an open question. Type safety in polylingual systems is the subject of Chapters 6 and 7.

3.1.3 Type Compatibility

Suppose a software component, c, defines an object of type t and wishes to communicate the object to another software component, c'. However, the languages of components c and c' are different: $L(c') \neq L(c)$. In order for the object to be received and interpreted, the component c' must have some type, t', in its own language $L(c')$ that can represent the object.

Some interoperability approaches require strict compatibility between types t and t'; that is, the two type definitions must be exactly the same, modulo differences in language syntax. Such strictness is not always desirable, however. Independently developed software components are unlikely to have identically defined types. It is advantageous if similar but non-identical types may be considered "compatible enough" so the type definitions need not be modified in order to permit interoperability. Such compatibility is called *relaxed* [75]. For example, one could consider two objects "compatible" if their types are in a subtype/supertype relationship, or if the types are structurally equivalent but not name equivalent. The bank account types given in Chapter 1 are another example, as they have some but not all methods in common.

In this dissertation an interoperability approach will be favored if it supports both strict compatibility and some form(s) of relaxed compatibility. Examples of relaxed compatibility are renaming of methods and reordering of method parameters.

Note that type compatibility and type safety are inherently in conflict. If the difference between two types is great, it may be difficult or impossible to ensure type safety of shared instances of those types. Relaxed matching of types will be covered in great detail in Chapter 6.

3.2 Ease of Use for the Interoperability Engineer

3.2.1 Diversity

All interoperability approaches support at least two languages: that is, they allow software components written in two distinct languages to communicate. Some, such as ILU [36], support more than two.

Support of a third language tends to introduce new complexities. If the approach uses an intermediate representation for types, the features of a third language may reveal limitations in the intermediate representation that worked properly for the first two languages but fails for the third. If on the other hand, the approach maintains language-to-language mappings between all pairs of languages, the addition of a third language continues the quadratic climb in complexity of maintaining these mappings.

In this dissertation, the more languages that are supported by an interoperability approach, the more favorably that approach will be rated. In addition, if an interoperability approach supports languages from differing programming paradigms (e.g., functional, procedural, object-oriented, logic programming, etc.), this will also count in its favor.

3.2.2 Transparency

It is advantageous for the source code of a software system to be free of references to interoperability: that is, for interoperability to be transparent.[1] If a software

[1]It is important to emphasize that this statement refers to source code written by a software developer, not source code generated by an automated tool, and not object code. At some lower level in any multilingual software system, the bridges between languages become apparent.

component becomes laced with interoperability-related code, the component becomes difficult to create, test, and maintain independently. On the other hand, if interlanguage method calls appear identical to intralanguage method calls, then at least in theory, a component created for use within a monolingual software system could be used later, without modifications, in a multilingual system.

In this dissertation, the transparency of an interoperability approach is given one of three ratings: non-transparent, uniform, or seamless. A *non-transparent* approach places explicit, identifiable, interoperability-related statements into the source code of the interoperating components.

A *uniform* approach implies transparency of programming language, from the perspective of an interacting software component. Intuitively, interoperability in a system is uniform if one cannot discern the language of any component in the system except by examining that component's source code. That is, the language of a component is not visible from outside the component. Uniformity promotes separation of concerns, allowing interoperability engineers to ignore the languages of the components they are integrating, thereby reducing the need to plan for special cases of language-to-language interoperability.

Definition 3.1 (Uniform) *Interoperability in a software system s is* **uniform** *if the following are true for every software component c in s:*

1. *For all components $c_1, c_2, \ldots, c_k$ in system s that are accessed within the source code of component c, it is not possible, by examining only the source code of c, to determine whether components $c_1, \ldots, c_k$ are written in the same or distinct programming languages. Note that the language of c is not relevant, i.e., $c \notin \{c_1, \ldots, c_k\}$.*

2. *It is not possible, by examining all of the source code in s except c, to determine the language of c.*

Seamlessness refers to transparency of interoperability, from the perspective of an interacting software component. Intuitively, interoperability is *seamless* if one cannot tell the difference between intralanguage and interlanguage accesses within the source code of the system. Seamlessness, like uniformity, promotes separation of concerns.

Definition 3.2 (Seamless) *Interoperability is* **seamless** *in a system s if for every software component c in s, and every other software component c' in s that is accessed by c, it is not possible, by examining the source code of component c, to determine whether the languages of c and c' are the same or distinct.*

A seamless approach is trivially uniform: if interoperability is transparent, so are language differences. A uniform approach is not necessarily seamless, however, since interlanguage calls could hide the language of the callee (hence, uniformity) but still look different from intralanguage calls (violating seamlessness). Thus, seamlessness is a stronger condition than uniformity.

3.2.3 Nonintrusiveness

Given a set of software components that are required to interoperate with one another, it is advantageous to minimize the work required to bring about this interoperability. In particular, we'd like to avoid modifying the components at all.

Interoperability approaches can be rated according to their level of intrusiveness, that is, how much modification is necessary to a component in order to make it interoperable. If no modifications are necessary, the approach is called *nonintrusive*.

Definition 3.3 (Nonintrusive) *An interoperability approach is* **nonintrusive** *with respect to a software system s if no modifications need to be made by the developer to the source code of the components of s in order for them to be made interoperable.*

It is important that nonintrusiveness is defined in context, with respect to a particular software system. SoftBench [24], for example, is nonintrusive only when applied

to software components that interact with the outside world through standard input and output (stdin and stdout). SoftBench could not bring interoperability to a graphical paint program that accepts only mouse events as input without modification of the program's source code.

An example of an intrusive approach is FIELD [61], which requires components to convert their data into ASCII strings before transmission. Thus, all communicating components must be modified.

3.2.4 Automation

Automation is the replacement of human effort by machine computation. It relieves the interoperability engineer from repetitive tasks that are not only tedious but also error-prone. An example of automation is an RPC stub generator, such as CORBA's IDL compiler. An approach that required an engineer to modify procedure calls by hand, converting them into remote procedure calls, would not be considered automated.

In this dissertation, the more automated an interoperability approach, the more favorably it will be rated. The ideal approach is completely automated—no human intervention required—but this is beyond current programming technology and perhaps impossible due to tremendous semantic requirements.

Definition 3.4 (Partially Automated) *An interoperability approach is* **partially automated** *if any interoperability-related code is generated by a software tool, other than a compiler, for any of the component's languages.*

Definition 3.5 (Fully Automated) *An interoperability approach is* **fully automated** *if, given only a set of software components and a high-level specification*[2] *of how those components should interact, the approach makes the components interoper-*

[2]I recognize that "high-level" is in the eye of the beholder.

able in accordance with the specification without any additional programming required by humans.

3.2.5 Performance

Time efficiency and space efficiency of an interoperability approach can be highly important in practice. This dissertation, at times, will touch upon performance issues, but primarily it is concerned with more model-level differences between interoperability approaches. Chapter 11 will evaluate the performance of PolySPINner in particular.

3.3 Ease of Use for the Interoperability Designer

3.3.1 Extensibility

As old programming languages die out and new ones come into vogue, it is highly desirable for an interoperability mechanism to support new languages so software written in those new languages can be integrated successfully with legacy code. Hence an interoperability mechanism should be as extensible as possible, allowing new languages to be supported by the mechanism with minimal effort on the part of future interoperability designers. As it is impossible to anticipate all interoperability requirements of future languages, a mechanism should at least provide a clear specification of how to add support for a new language.

In this dissertation, an interoperability approach will be viewed favorably if it specifies a process for adding support for a new programming language, particularly if that approach is well documented. Some approaches, such as CORBA [55], specify standards to aid interoperability designers in creating CORBA language bindings for new languages. Most interoperability approaches, however, do not specify such standards, and many approaches are not readily extensible.

3.4 Other Features Not Considered

The preceding list of interoperability features is not comprehensive. It focuses on model-level features. Applied features like the following are not considered:

- Computing requirements, such as physical memory, disk space, and processor speed required to use the approach.
- Political requirements, such as the industry reputation of the approach or its inventor.
- Other practical requirements, such as availability of the source code of the interoperability mechanism so it can be modified in house, or effectiveness of the technical support staff.

3.5 Justification

In June 1997, I conducted a survey of software engineering practitioners to verify that the interoperability features listed in the Interoperability Manifesto are generally acknowledged as important. The survey form, presented in Appendix A, asked practitioners for their opinions of the importance of 25 interoperability features. These included not only the features of the Manifesto but also some implementation-level features, such as disk space requirements and vendor reputation, that are not of direct concern here.

Features were presented as questions. Survey respondents were asked to imagine that they were considering the use of a new interoperability mechanism, called "XYZ." If an XYZ expert were in the room with them, what questions would the respondents most want to ask the expert, in order to decide whether XYZ were an appropriate interoperability solution for their project? Each question was rated on a Likert scale [33] from 1 ("Completely irrelevant—I do not need this information to make a valid

decision") to 5 ("Extremely important—I could not make a valid decision without this information").

The survey was posted in eight Usenet newsgroups devoted to relevant topics, and distributed to several known individuals in the field. In all, thirteen respondents completed the survey. While this number is too small to generalize the results to the software engineering community, it is enough to generate descriptive information about the practitioners in the sample. In certain cases, findings were statistically different from zero, suggesting that they might be useful for making inferences (with caution) to the software engineering community.

The survey results, described in detail in Appendix A, suggest the following:

- The six most highly-rated features of an interoperability approach are:

    1. Which programming languages are supported.
    2. Which platforms are supported.
    3. Transparency.
    4. Nonintrusiveness.
    5. Support for interoperability of primitive types.
    6. Support for interoperability of structured types.

- Every feature was rated very important (4 or 5) by at least one respondent, suggesting that all features have importance to at least some software developers.

- Although general comments were solicited from respondents, no respondent suggested that any interoperability feature was missing from the survey.

These results suggest that the Interoperability Manifesto contains a representative set of important features of interoperability, and therefore may be used as a means for comparison among interoperability approaches.

Other interesting results examined in Appendix A include:

- A practitioner's experience with particular interoperability mechanisms influenced his/her ratings of the importance of certain features.
- Practitioners who had experience with similar mechanisms (e.g., SoftBench and ToolTalk, which are based on the same abstraction) made similar ratings of importance.

3.6 Summary

The Interoperability Manifesto provides a means for comparing and contrasting interoperability approaches. It will be used as a basis of comparison throughout this dissertation, and in particular in Chapter 8 (PolySPIN vs. CORBA).

CHAPTER 4

RELATED WORK

Polylingual systems research builds primarily on previous work in four areas of computer science: interoperability, databases, object-oriented programming, and type theory. As an interoperability approach, the polylingual system concept can be compared to other interoperability approaches such as remote procedure calls, common type systems, and language extensions. Databases are often used for interoperability, both as a medium for communication (as in some implementations of PolySPINner), and as interoperating entities in their own right (e.g., federated database systems). Polylingual systems model their data as objects in a traditional object-oriented sense, so previous research in object-oriented programming is relevant. Finally, several aspects of polylingual systems, such as the type-matching features of PolySPIN, draw from type theory. This section provides an overview of related work in these four fields.

4.1 Interoperability

4.1.1 Low-Level Approaches

Some approaches to interoperability make little use of abstraction, requiring the application developer to attend to fine-grained details of interoperability such as data translation and parameter marshaling. We informally call such approaches "low-level."

Common file formats: A low-level approach popular in PC software is the use of common file formats, in which a disk file is used to store data for access by several

software components. The components agree on the data format (e.g., a file of records, each containing three integers and a float) and access the data using their language's respective input/output operations. Since each component uses its language's type system for interpreting the data in the file, crude interoperability is achieved. A major disadvantage is that each application is responsible for converting its data into a stream of bytes to be stored in the file, an inefficient process that is particularly wasteful and difficult when converting pointer-based data structures. This approach is most applicable when components operate in sequence: one component writes the data, and another component reads it afterward. A common example is a UNIX pipeline [6].

Procedure call: Another low-level approach is the use of procedure calls, in which a software component known as the *caller* sends procedure parameters to another known as the *callee*, optionally returning values from the callee to the caller.

Procedure call began as a software integration technique for communication among subprograms within a single program, written in a single language, executing in a single address space. The model has since been extended to work with distributed programming, in the form of *remote procedure calls* (RPCs). An RPC transmits procedure parameters and return values between software components executing in different address spaces. *Heterogeneous* RPC is RPC between components written in distinct languages or running on computers with differing data representations.

RPC has been implemented in numerous systems. Perhaps the best known RPC implementation is Sun RPC [70], included in virtually every UNIX implementation. A Sun RPC is accomplished by invoking a special "call RPC" function, so RPCs are easily distinguishable from ordinary procedure calls. A more transparent approach is that of Cedar [12], which adds stub generation for both the caller and the callee so that RPCs look like ordinary procedure calls. This is advantageous because it allows the choice between ordinary and remote procedure call to be delayed until link time,

facilitating loose software integration. Cedar stubs are generated automatically from higher-level specifications, called interface modules, via a compiler. For calls that fail, an exception handling mechanism is provided. Matchmaker [38] is similar to Cedar. It consists of a Pascal-like language for creating RPC declarations, and a compiler that converts the declarations into stubs.

The problem of heterogeneous RPC has been addressed by HRPC [10] and Mercury [47], among others. HRPC focuses on the dynamic, automatic location and activation of callees as they are needed. Mercury focuses on *asynchronous* RPC, using a mechanism called a call-stream to guarantee that procedure calls and their return values reach their recipients in the same order that they were sent. Q [52] is an RPC mechanism that provides point-to-point communication between C and Ada programs.

The *promise* approach [48] is similar to asynchronous remote procedure calls but uses the principle of lazy evaluation. In this approach, if a remote procedure call is made that should return a value of type t, the call immediately returns a placeholder, called a promise, to stand in for the true return value. This allows the caller to continue executing. When the remote procedure completes its execution, the promise in the caller is automatically replaced by the return value. Call-streams are used, as in the Mercury RPC approach, to insure that ordering is preserved. Promises were preceded by *futures* in MultiLisp [28], but futures lack strong typing and exceptions, which promises have. Promises have also been extended to be first-class objects, called *responsibilities* [23], so they can be passed as procedure parameters, not just as return values.

Procedure call is conceptually simple and remains one of the most popular models for the integration of software components. Unfortunately, it has numerous problems. If components are supplied without source code (black boxes), it may be impossible to link them together using procedure calls, remote or otherwise. Remote procedure

calls also have great difficulty when passing pointers as parameters, since a pointer in the caller's address space may be meaningless in the callee's address space. Finally, RPC does not support many of the useful abstractions (e.g., multicast) found in more modern mechanisms. Higher-level integration mechanisms, however, are commonly built on top of RPC. For example, Sun's ToolTalk [39] builds on Sun RPC to provide multicasting.

Message passing: *Message passing* (e.g., [68]) is a method of communication in which units of information, called *messages*, are transmitted between "sender" and "receiver" software components. In various message-passing approaches, messages may be transmitted via point-to-point, multicast, or broadcast semantics, and either synchronously or asynchronously.

In order for software components to communicate via message passing, either the components must agree on a common message format that all can send and/or receive, or each component must be supplied with a front-end (e.g., a wrapper, a server process) that translates messages into a format the component can understand. A common means for getting components of distinct languages to exchange messages is to provide a library of message-passing functions for each programming language. For example, Polylith [60] provides functions for sending and receiving synchronous and asynchronous messages. Other well-known message-passing systems include FIELD [61], SoftBench [24], and Isis [11].

4.1.2 Common Type System

A popular means for allowing software components to interoperate is to require all of them to create and access data objects via a common type system, rather than (or in addition to) the type systems of their respective programming languages. A primitive example is that of FIELD [61], in which all shared data is represented as

ASCII strings. Applications are responsible for converting their data into strings, though FIELD provides some assistance.

A somewhat more structured approach is that of CORBA [55], which provides a type system for shared objects. Types are declared in a language called Interface Definition Language (IDL) which can encapsulate primitive types, arrays, records, and other types. If a participating software component has data to share, it provides an IDL specification of the data type, including public methods for other components to invoke to access the data. CORBA's IDL compiler then translates the IDL interface into linkable object code. IDL is not rich enough to represent types as complex as those found in most programming languages, such a pointer-based data structures, and objects with multiply-inherited methods. So the software component must also provide code that translates between its internal data and IDL's representation of that data. Once IDL types are defined and compiled, CORBA uses an intermediary component, called an Object Request Broker (ORB), that provides name service and location service for objects. When an application (called a "client") invokes an IDL method, an ORB transparently locates the intended object, forwards the invocation (called a "request") to the object, and forwards any return value back to the client. In this sense, CORBA is much like a remote procedure call approach. CORBA will be discussed in great detail in Chapter 8.

A similar approach is that of data definition languages, as found in SDL and CDDL of DEC's VMS operating system [13]. A single data specification is written in a common language, and a translator converts this specification into "equivalent" type definitions in various programming languages. In theory, any data created using one of these generated type definitions is accessible using any of the others. Some languages, however, might not support all constructs of the data definition language; in this case, the translator represents the data as "untyped." Untyped data can be

read but cannot be used in computations. A related problem is that not all data types in a given language are necessarily representable in the data definition language.

A more high-level approach is taken by CHAIMS [59]. Although the research is in a very early stage, CHAIMS is intended to permit interoperability among components that already use other interoperability mechanisms, such as CORBA, OLE, JavaBeans and DCE. An interface language is used to describe the components, and these descriptions are compiled into interoperability code.

Some systems use the type model of a database as a common type model, using the database for storing and retrieving shared objects, and requiring software components to use the database's type system for these objects. The database approach is discussed further in Section 4.3.

4.1.3 Languages and Language Extensions

Rather than require interoperating software components to use a type system external to their programming languages, some approaches extend the programming languages to add features helpful for interoperability. For example, Concert [5] adds type annotations to formal parameters in the C programming language, creating a new language, Concert/C. The Concert/C compiler reads the annotated type declarations of a component and generates a language-independent interface for that component to support interoperability. These interfaces are similar to CORBA's IDL specifications, but they are meant to be used as black boxes, not read by humans. Thus, Concert/C trades the complexity of writing IDL wrappers for the inconvenience of using an extended language. Concert works only with C and C++, but a followup project, Mockingbird [4], provides Concert-inspired features to several other languages. Its focus is still on primitive and structured types, not abstract data types.

Another approach is subject-oriented programming [30], an extension of object-oriented programming. Rather than giving each object a single interface (that of its

class), subject-oriented programming permits an object to have multiple interfaces, with choice of interface being a function of the object's accessor. That is, depending on who is accessing the object, the object presents a different interface. Although subject-oriented programming has thus far been presented as a technique within a single, object-oriented language, it holds promise as the basis for a method of interoperability. Specifically, an object could present a different interface for each programming language. Polylingual systems could be viewed as an extension of subject-oriented programming, permitting the same physical object to be accessed from multiple languages, as if the object were an instance of a different type in each language.

4.1.4 Polylingual Systems

Polylingual systems were proposed by Kaplan and Wileden [41, 42] as an unexpected outgrowth from Kaplan's research on naming [40]. While creating a name management facility for the Open Object-Oriented Database (OpenOODB) [72], Kaplan realized that further transparency could be obtained from the OpenOODB using the technique he came to call PolySPIN. This dissertation builds on the original PolySPIN work, specifying it more precisely and extending its features.

4.2 Object-Oriented Programming

Object-oriented programming had its inception with Simula [19], a programming language that was rooted in Algol but introduced an important new concept: the object. Objects were typed entities that contained *methods* (i.e., operations) and *variables* (attributes, fields). The type of an object was called a *class*, and in order to define a class, one defined a set of methods that pertained to objects of that class. Thus, the emphasis of programming was shifted from procedural, in which operations were invoked on data not associated with the operations in any formal way, to object-oriented, in which data "contained" the operations relevant to them.

Smalltalk [25] extended the object metaphor by making everything in the language an object, including primitive types and even classes themselves. Many other object-oriented languages have emerged since then, such as C++ and Eiffel.

A key concept in object-oriented programming is *inheritance*, also introduced by Simula. If a class A inherits from a class B, then A shares the the attributes and methods of B. Class A is called a *subclass* of B and may also contain additional attributes and methods. The graph in which the nodes are classes and the edges represent inheritance relationships is called the *inheritance hierarchy* of a set of classes. If a class may inherit from no more than one other class—i.e., the inheritance hierarchy is a tree—this is called *single inheritance*. If a class may inherit from more than one other class, then the inheritance hierarchy is a directed acyclic graph and the model is known as *multiple inheritance*. In neither case may inheritance be circular.

The object-oriented metaphor is used not only for programming languages but also for other aspects of building systems; in particular, it is found in interoperability. One popular object-oriented approach to interoperability is that of *compound document* systems, such as Microsoft's OLE [15] and Apple's OpenDoc [50]. In these approaches, objects called compound documents 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 spreadsheet is accessed within the document, method calls are automatically routed to Excel. CORBA [55], discussed in Section 4.1.2, also uses the object-oriented metaphor, though its inheritance model is severely restricted, prohibiting the overloading of method names.[1]

[1] In addition, approaches such as OLE and CORBA tend to support only inheritance of interfaces, i.e., subtyping, not inheritance of method implementations.

4.3 Databases

A *database* (e.g., [71]) is a repository for data of various kinds. Data can typically be added to, deleted from, changed in, and retrieved from a database. Data access is accomplished by *queries* on the database, in which a subset of the data is selected using a language called a *query language*. A *database management system* (DBMS) consists of the database itself and whatever software is necessary to enable the database to be used, such as a query engine and a graphical user interface.

The most frequently encountered database model is the *relational database* (e.g., [21]), in which data are modeled as relations (i.e, tables). For example, a set of personnel records could be modeled as a relation on a name (a string), an age (an integer), and a salary (a real number), and viewed as a table containing these values.

Data can conceivably be organized in many ways in a database. For example, the above personnel database is organized as records containing a string, an integer, and a real. Such an organization is called a *schema*. A schema is analogous to a type in a programming language, in that it describes allowable organizations of data.

Databases serve two roles in interoperability: they may be used as a medium for achieving interoperability (e.g., a shared repository), or multiple databases may themselves interoperate as in a federated database system.

4.3.1 Federated Database Systems

A *federated database system* [31] is a set of database systems, called component database systems, that operate independently but also cooperate and share data. For example, the databases of all major airlines en masse may be considered a federated database system, since they often permit queries on one another. The component database systems may be loosely or tightly integrated. Federated database systems also may be composed, allowing one federated database system to be a component in another federated database system.

Much attention has been focused on the problem of heterogeneity in federated database systems: that is, achieving cooperation between DBMSs from a variety of hardware and software environments, using a variety of schemas, query languages, and constraints [67]. If components of a federated database system have differing query languages, for example, it may be desirable for queries in one language to be automatically converted into queries in another. This issue, termed "command transformation," is analogous to the property of transparency in interoperability, since both issues are concerned with homogeneous data access to heterogeneous entities.

Another issue of heterogeneity is schema translation, in which data represented by one schema must be converted into another schema so that component databases with differing schemas can interact meaningfully. This issue is analogous to type compatibility in interoperability, in that both address compatibility between data of differing representations. Of note is the issue of semantic heterogeneity [67] in federated databases, in which data represented by syntactically similar schemas actually have different semantic meanings, and thus should not be considered equivalent. A similar problem arises in polylingual systems when attempting to match methods with the same names and signatures but different execution semantics.

The problem of constructing a federated database has been addressed (e.g., [67]). The "top-down" approach describes the situation when a federation already exists and new component databases must be added, and the "bottom-up" approach describes the situation when the component databases exist prior to the federation.[2]

4.3.2 Object-Oriented Databases

An *object-oriented database system* (OODB) permits access to data via an object-oriented abstraction [3]. In contrast to relational database systems, which treat data

[2]These two cases correspond to the "easiest case" and "megaprogramming" interoperability scenarios, respectively, to be described in Section 5.2.1.

as a set of relations, OODBs encapsulate data as complex, typed, persistent objects. The traditional notion of schema is replaced by a type hierarchy that typically supports some form of inheritance.

OODBs theoretically combine the convenient persistence of databases with the rich type model of programming languages. As such, they are useful as a medium for facilitating interoperability between software components. Whereas a traditional relational database might require software components to convert their data into relational tuples before the data can be stored in the database (and later retrieved by other components), an OODB may require little or no data conversion. For example, the Open Object-Oriented Database (OpenOODB) [72], a software package used by PolySPINner, permits C++ and CLOS programs to store and retrieve objects created within those languages. OpenOODB does not, however, provide support for interoperability of those objects; i.e., a C++ program might not be able to make sense of a CLOS object.

4.3.3 Views

A *view* [21] is an abstract representation of data that is derived from, but differs from, the physical or logical representation of that data.[3] Suppose a relational database contained two tables: one listing people's names, ages, and telephone numbers, and a second table listing people's names and marital status. A view of this database might contain the names, ages, and marital status of all people named John appearing in both tables. Thus, the view represents a subset of the data from the join of the two tables. The classic problem with views is that some of them cannot be updated; that is, modifications made to the data in a view cannot always be translated into semantically equivalent modifications of the underlying data.

[3]Views have also been considered as partial representations of programs, e.g., [35].

Multiple software components that access the same data may have different views of the data. Thus, views are related to interoperability. An example of view-inspired interoperability is found in ViewPoints [54], a framework for integration of independently-developed components. A viewpoint is an encapsulation of a problem domain that includes component information. By integrating the viewpoints, component integration is achieved as well. Another example is the MultiView project [63], which incorporates views into object-oriented databases. The views encapsulate schemas and provide an additional level of abstraction, thereby facilitating integration.

4.4 Type Theory

The definition of "type" given in Chapter 2 is not the only definition adopted by type theorists. Some common definitions are:

- Informal: A type is a description of properties of a set.
- Constraint-based: A type is a set of constraints on the permissible values of an expression.
- Denotational semantics: A type is a property associated with a logical term.
- Algebraic: A type is a set of elements upon which the operations of some algebra are defined.
- Object-oriented: A type is a set of operations. This is similar to the algebraic definition but views a type primarily with respect to the operations.

The informal definition is most akin to philosophy: that similar entities have common properties that are encapsulated as an abstract archetype. The constraint-based definition has been adopted by some as a convenient vehicle for proving assertions about type correctness (e.g., [58]). The denotational semantics definition is based on

the lambda calculus [66]. The algebraic definition comes from mathematics and finds a place in group theory, software specification languages (e.g., Larch [27]), and other areas. The operation-based definition (e.g., [1, 16]) is closely related to the software engineering notion of an abstract data type (ADT) and is adopted in this dissertation as explained in Chapter 2.

4.4.1 Object-Oriented Type Theory

Most type theories for procedural and functional programming are based on the lambda calculus [18], a language of formal logic that is utilized to model functional computation. The lambda calculus is an inconvenient model for object-oriented programming, however, since it cannot conveniently model assignment, inheritance, and late binding [58]. As a result, object-oriented type theories have been developed. They are particularly relevant to polylingual systems research because the data shared by software components are modeled as objects. Numerous approaches to object-oriented type theory have been proposed, and Danforth and Tomlinson [20] classify them according to their modeling of classes and inheritance. More recent work, notably that of Abadi and Cardelli [1], Bruce [16], and Palsberg and Schwartzbach [58], endeavors to model object-oriented notions explicitly without dependence on the lambda calculus. Abadi and Cardelli specify several levels of object calculi to model subtyping and inheritance, focusing on difficult issues of covariance and contravariance in recursive types. The Abadi/Cardelli model plays a crucial role in the development of a formal framework for PolySPIN in Chapter 6.

4.4.2 Type Compatibility

An important concern in polylingual systems is the notion of compatibility between types, since (as shall be seen) operation invocations on an instance of type t in a language l may need to be converted into invocations on an instance of another type t' in another language l'. Zaremski and Wing [75] devised a taxonomy of functions for

non-exact, or *relaxed*, type matching, both for types (which they call module types) and for operations (which they call function types). Their domain is software reuse: given a library of types, plus a single type called the query, the goal is to locate types in the library that match the query according to specified criteria. Zaremski and Wing's criteria span exact matching, generalized and specialized matching through type substitution, reorderings of parameters within a signature, reorderings of operations within a type, and so on. Zaremski and Wing's taxonomy applies to types within a single language only. In this dissertation, their model is extended and applied to interlanguage type matching (Chapter 6), and this functionality is implemented in PolySPINner.

Relaxed matching has also been applied to interoperability by Glinert and Sánchez-Ruíz [65]. Given a type t defined in a language l_1, Glinert and Sánchez-Ruíz focus on the representation of type t in a foreign language l_2 when l_2 lacks the features to represent t directly. For example, t could be a pointer-based data structure, and l_2 a language without pointers. Using an interconnection language similar to that of Polylith [60], and a table of language-to-language data structure equivalences, their technique generates stubs that encapsulate legacy types and permit them to interoperate. Abstract data types are not supported, just primitive and structured types. Sánchez-Ruíz also presents a formal approach to type compatibility [64] by considering a type model to be a set of building blocks (primitive types) and construction rules (e.g., how to define a record). The goal is to present a unified theory for type models so that type models of different languages may potentially be reconciled.

Konstantas [43] has also addressed relaxed matching in object-oriented interoperability. His language H-TMSL is used to describe relationships among method signatures, including not only transformations similar to Zaremski and Wing's (parameter reordering, etc.), but also arbitrary functions to transform the parameter list of one method into that of another.

4.4.3 Type Evolution

An important application of type matching is type evolution, also called schema evolution by the database community. If the type definitions of an application program are modified, persistent data of that type might also need to be modified to remain consistent with the new type definition. Certain kinds of modifications are not difficult to deal with, such as adding a field to a record type, but others can be more complex, such as reordering the fields of a record and simultaneously changing their names.

When types change, it is important to be able to infer matchings between old types and their modified versions [45]. Sometimes inference can be fully automated, but other times it is not possible without human intervention. For example, if a record type `person` contains one integer field, `age`, and type `person` is modified so that it now contains two integer fields, `years` and `days`, it is not clear whether `age` was transformed into one of the new fields (and which one?), or whether `age` was deleted.

Type evolution and interoperability are related because both require data types to be compared. When a type changes, it is common to compare the old and new versions in order to determine how the old data must be changed to be consistent with the new type. Similarly, when two software components interoperate, their data types might not be equivalent, so transmitted data may need to be transformed from one type (or type representation) to another.

Type evolution and the PolySPIN approach are related but have different emphases and perspectives. Primarily, type evolution is mostly concerned with the structure of a type while PolySPIN is most concerned with compatibility of methods. Type evolution is often applied to different versions of the same type: that is, type definitions that are historically related. PolySPIN is intended to be applied to independently-developed types with no historical relationship. Type evolution

emphasizes the issues of whether types are equivalent, and what is the set of transformations for converting instances of one type into another. PolySPIN emphasizes the issues of whether two types are compatible with respect to a user-supplied compatibility criterion. Type evolution takes place over time, in that old types are replaced by new ones, and in many cases the old types will no longer be used to create new instances. Interoperability with PolySPIN is geared toward pairs of types, both of which may still be in active use for creating new instances. Finally, PolySPIN's emphasis is on interlanguage type compatibility, whereas much type/schema evolution research has tended to focus on the single-language case; however, Tess [45] is capable of multilanguage comparisons of types.

4.5 Summary

Polylingual systems research is a fairly recent development in the field of interoperability. It draws upon research on software engineering, programming languages, databases, and type theory to create a novel approach to high-level interlanguage communication.

CHAPTER 5

POLYLINGUAL SYSTEMS

Now that Chapters 1–4 have set the stage for discussing interoperability in general, it is time to introduce polylingual systems, the subject of this research. This chapter[1] provides a compact overview of polylingual systems and the concepts behind PolySPIN (the framework) and PolySPINner (the implementation). The versions of PolySPIN and PolySPINner discussed in this chapter—version 1, or "V1"—were the first ones created as a part of this dissertation. The description will culminate in a brief comparison with ILU [36], a CORBA-style interoperability mechanism.

The general discussion found in this chapter will be expanded in greater detail in later chapters. The present goal is to make the reader familiar with the big picture of polylingual systems and PolySPIN—their features, strengths, and weaknesses—in preparation for more detailed chapters to come:

- PolySPIN: Chapters 6 and 7 will examine issues of type safety and compatibility through the use of formalisms for polylingual interoperability. (Type safety and compatibility will be discussed only briefly in this chapter.)

- PolySPINner: Chapters 9 and 10 will discuss the architecture and design decisions of the PolySPINner toolset, and Appendices B and C will cover its internals in great detail. Note that the version of PolySPINner discussed in this chapter, PolySPINner V1, lacks some important features discussed in later chapters.

---

[1] This chapter is an extended version of [9],

- CORBA: Chapter 8 will present an in-depth comparison of CORBA and PolySPIN, using the Interoperability Manifesto of Chapter 3 as a guide.

A running example of two architecture firms will be used to illustrate key points. The chapter will then define polylingual interoperability, describe the PolySPIN framework and the PolySPINner toolset, and apply both ILU and PolySPIN to the architectural example.

5.1 A Motivating Example

Suppose two architecture companies, the Frank Firm and Lloyd Ltd., are contemplating a merger. Each company has important software assets, such as personnel information and computer-aided design 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 in CLOS (the Common Lisp Object System). If the merger is to be successful, Frank Lloyd must solve the multilanguage 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, Inc., expects its new assets to provide.

Suppose the newly merged Frank Lloyd company wants to develop a new CLOS application that assigns office space to its employees. Office space is assigned based on information in personnel records, such as the employee's salary or the number of years the employee has been with the company. The old Frank Firm records are implemented as C++ objects, and the old Lloyd Ltd. objects are in CLOS. The newly merged company does not want to translate the objects of one language into another; rather, the office space application should access both the C++ and CLOS objects uniformly, using functions like that of Figure 5.1, which computes a simple office-

```
(defun office-rank (employee)
  (/  (*  (YearsOfService employee)
          (Salary employee))
      10000))
```

Figure 5.1. Sample function used in the CLOS office space application.

ranking score based on an employee's salary and years of service. Frank Lloyd, Inc., wants such functions to operate, unchanged, on both C++ and CLOS objects.

5.2 Polylingual Interoperability

In Chapter 2, the term *multilingual interoperability* was defined as interoperability among software components of at least two languages. A simple example would be a remote procedure call from a C++ component to a CLOS component. The sort of interoperability envisioned by Frank Lloyd, however, is of a different form: a single component interacting seamlessly with a set of other components implemented in different languages. It is defined as follows:

Definition 5.1 (Polylingual Interoperability) *Given a software component, c, and a set of other software components, $C = \{c_1, c_2, \ldots, c_n\}$. Interoperability between component c and the components of set C is* **polylingual** *if both of the following conditions hold:*

1. *The components of C are written in different languages, i.e, $|L(C)| > 1$. The language of c is irrelevant.*

2. *The interoperability is seamless (Section 3.2.2), i.e., the fact that the components of C are written in different languages is not detectable from the source code of component c.*

Definition 5.2 (Polylingual System) *A* **polylingual system** *is a set of interoperating software components, S, such that for every component $c \in S$, and every other*

component $c' \in S - \{c\}$ *with* $L(c) \neq L(c')$*, interoperability between components* c *and* c' *is polylingual.*

In the Frank Lloyd example, a personnel application written in C++ that invoked a subprogram written in CLOS by remote procedure call would be a case of multilingual interoperability. A CLOS program that seamlessly accessed personnel records, some stored as C++ objects and others stored as CLOS objects, to assign employee office space would be a case of polylingual interoperability. The term *polylingual* was coined by analogy to the software term "polymorphic," which means "uniform processing, independent of data type." "Polylingual" means "uniform processing, independent of language."

Suppose that two new tools under development at Frank Lloyd, Inc., are being written in distinct object-oriented programming languages, C++ and CLOS, and must share objects. In a typical multilingual system, the developers of both applications would write special code for accessing objects across the language boundary, either directly (using low-level foreign function calls, for example) or through an intermediary (e.g., a CORBA object request broker). In a polylingual approach, however, both applications would access C++ and CLOS objects seamlessly, invoking their methods as if no language barrier existed.

Such seamlessness has long been a highly desirable property of interoperability, but it has rarely been achieved. As shall be described, the PolySPIN approach achieves seamlessness by transparently modifying the bodies of methods to become "language aware," but not modifying their names or signatures. The approach allows the developers of software components and objects to write code without concern for language differences.

Definition 5.1 has a particularly interesting implication: in a polylingual system, it is possible to construct data structures in which the individual elements (objects) of a data structure can be of different languages. For example, an array or tree of

"employee" objects may contain both C++ and CLOS objects representing employees. Several experiments in Chapter 11 seamlessly utilize arrays of objects of different languages.

Also notable is that in a polylingual system, the interoperating software components access objects of other languages without the need for a foreign type system, such as CORBA's Interface Definition Language (IDL). Instead, each component manipulates only objects defined with its local type system, even if those objects may actually be instances of types of other languages. Existing components need not be modified to interoperate with these objects. The remainder of this section provides an overview of polylingual interoperability concepts.

5.2.1 Three Interoperability Scenarios

The decision to cause two software components A and B to interoperate can be made at three different times in the software lifecycle, as illustrated in Figure 5.2: 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 design both interoperating components from scratch, however. The second scenario is a more *common case*, in which a new software component B must be designed to interoperate with legacy system A. In this case, it is desirable that legacy component A need not be modified; but unless A was designed with future interoperability in mind, this is unlikely.[2]

PolySPIN's approach addresses the third (and therefore the first and second as well) and most difficult of the three scenarios, sometimes called *megaprogramming* [14, 73], in which both components A and B already exist. We want to modify

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

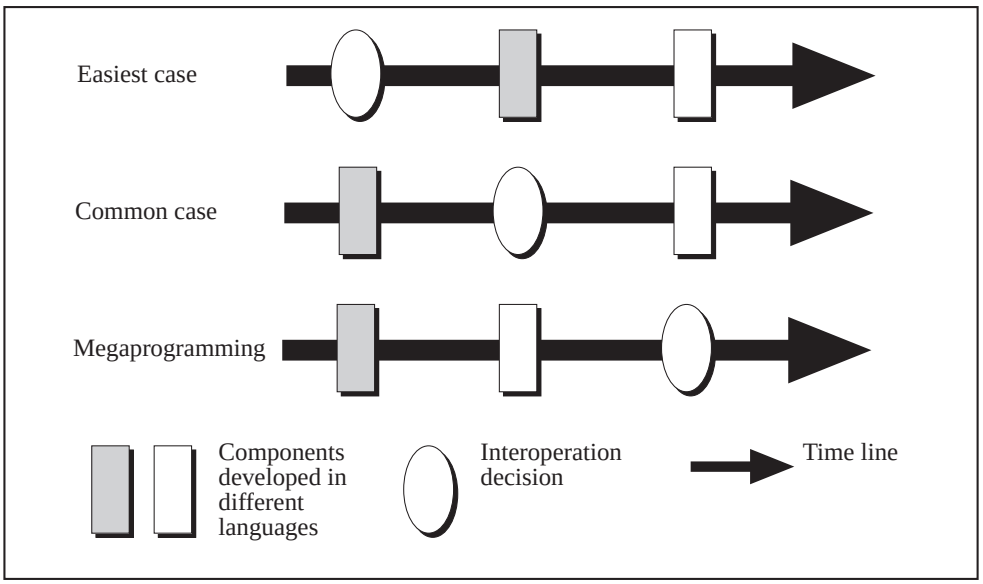


Figure 5.2. Three interoperability scenarios.

them as little as possible to make them interoperate. PolySPIN allows A and B to interoperate with no modifications visible to the developer. Other approaches such as CORBA are problematic in this scenario because they are intrusive, requiring both A and B to be modified significantly. Since type definitions in components A and B are translated into another language (IDL), all uses of those types may need to be changed into uses of IDL types.

5.2.2 Dimensions of Polylingual Interoperability

The PolySPIN approach is pictured in Figure 5.3.[3] One or more software components, called *accessors*, access other software components, called *objects*. (A given software component can be both an accessor and an object.) The accessors may be of different languages and, since we are modeling polylingual interoperability, the objects are implemented in at least two distinct languages.

[3]We stress that this is a conceptual framework, not to be interpreted as enforcing an implementation strategy.

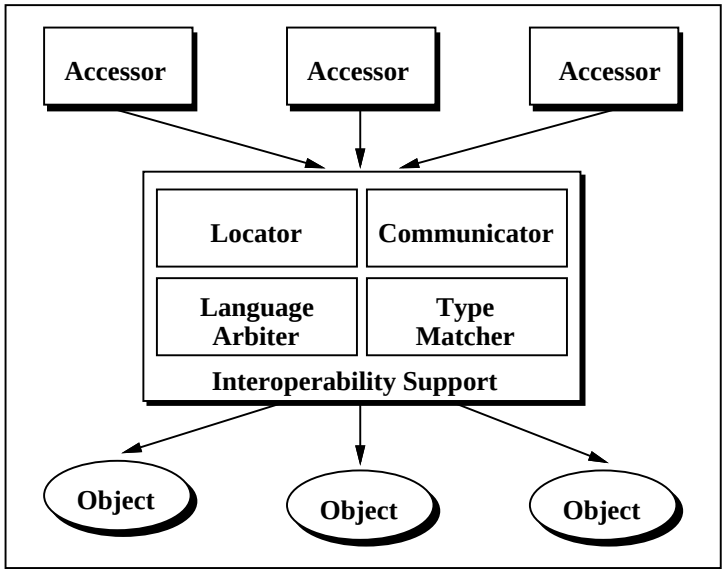


Figure 5.3. Conceptual framework for polylingual interoperability.

The four central subcomponents in Figure 5.3 represent dimensions of interoperability:

Interlanguage naming (Locator): Interoperability requires a 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 framework, 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 framework includes a mechanism for transparently associating language-specific information with a given object. This mechanism is termed a Language Arbiter.

Interlanguage invocation (Communicator): If the accessor and object are implemented in distinct languages, an approach to interoperability must provide a mechanism for invoking operations across the language boundary, including

marshaling and unmarshaling of arguments. In the framework, this mechanism is termed a Communicator.

Type compatibility (Matcher): In order to have uniform access to objects of different types, an accessor must be able to represent those objects in its own language. Thus, those types must be sufficiently compatible so that communication between accessor and object is sensible and meaningful. In the framework, satisfying this requirement is the role of the Matcher.

5.3 PolySPIN

Existing approaches to interoperability are generally not seamless. 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. PolySPIN, on the other hand, imposes neither different invocation mechanisms nor different type models and hence supports transparent interoperability. In this section we outline PolySPIN, and in the next, the toolset that automates its use.

PolySPIN is an approach to **S**upport for **P**ersistence, **I**nteroperability and **N**aming in **POLY**lingual systems [41]. Its origins lie in the development of a language-neutral name management mechanism that allows for uniform name-based access to objects [40]. The persistence and naming aspects of PolySPIN are discussed briefly in Chapter 10.

The interoperability aspect of PolySPIN focuses on transparency, allowing existing objects to be made interoperable without any changes to their interfaces. Instead, the *method bodies* are automatically instrumented to become aware of language differences and to convert same-language method calls into interlanguage method calls. When an object and its accessor have been implemented in the same language, and the accessor invokes a method of the object, the object responds normally, as it would in

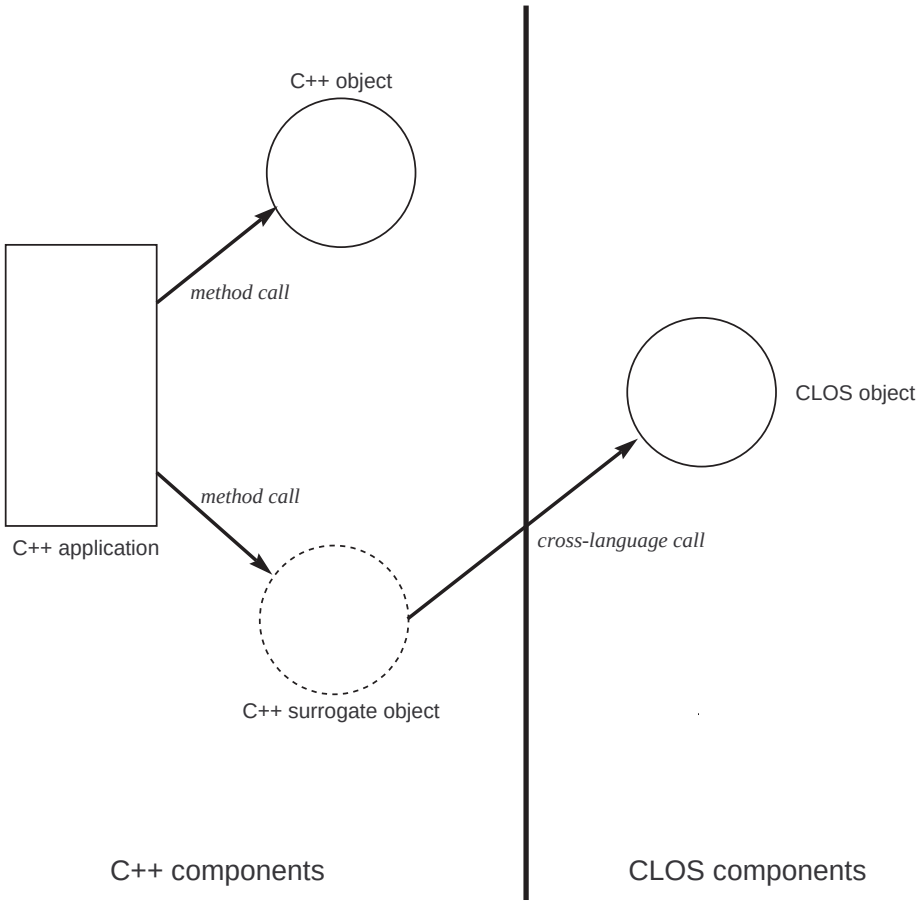


Figure 5.4. PolySPIN at work.

a monolingual system. If the object and accessor are of different languages, however, the method invocation is transparently routed through a *surrogate object* with the same interface, automatically generated by PolySPIN in the language of the accessor. Figure 5.4 illustrates both cases. The surrogate object then turns the accessor's method call into an interlanguage call to the object. Surrogate objects are not a new concept (e.g., see the literature on proxy patterns [17]), but their application to transparent, multilingual interoperability in the PolySPIN technique represents a new approach.

5.3.1 Type Matching

In order for PolySPIN to achieve transparency, a foreign object and its associated surrogate object must have interfaces that "match." That is, any method defined on the foreign object, in order to be invoked from an accessor's language, must be associated with a corresponding method defined on the surrogate. In other words, the types of the foreign object and the surrogate object must be, in some sense, "compatible."[4] From this notion comes one of the most important aspects of PolySPIN.

PolySPIN requires interoperating applications to have "compatible" types in order for the surrogate objects to be created automatically. More specifically, suppose we have a set of types $t_1, t_2, \ldots, t_n$ implemented in distinct languages. These types must be compatible enough that an instance of type t_i can "stand in" as a surrogate for a foreign object of any other type t_j. When an accessor of language i wants to access a foreign object of type t_j, an automatically-generated surrogate of type t_i transparently forwards the method call to the foreign object. This may seem a fairly strong requirement; however,

- *Any* approach to interoperability must wrestle with this problem. If two components have wildly dissimilar types, interoperability will be difficult to achieve regardless of the technique used.

- As will be seen in Chapter 6, "compatibility" between types may be relaxed, providing more flexibility at the possible expense of type safety.

In addition, some applications may contain similar types, such as the C++ and CLOS "employee" types of the Frank Lloyd example, that are similar enough to stand in for one another as surrogates.

---

[4] Never fear, the informal term "compatible" will be formally defined in Chapters 6 and 7.

5.3.2 The Components of PolySPIN

PolySPIN contains the functionality of the Locator, Communicator, Type Matcher and Language Arbiter of Figure 5.3. PolySPIN's interlanguage name-management functionality [40] plays the role of the Locator. The Language Arbiter transparently encapsulates language-specific information within objects by the PolySPIN technique. Interlanguage invocation (Communicator) functionality is achieved by automatically modifying the implementation of object methods so that they consult the Language Arbiter at each invocation (i.e., determine the language of the callee) and transparently select between making a local method call or an automatically-generated interlanguage call. Finally, the Type Matcher functionality is achieved via type compatibility checking, described extensively in Chapter 6, whose results influence the modifications made by PolySPIN to the object method implementations.

5.3.3 PolySPIN Tradeoffs

PolySPIN's major advantages are transparency and nonintrusiveness. Since class interfaces do not change, existing classes can be made interoperable by simply recompiling their method bodies and relinking. This is particularly advantageous for the megaprogramming interoperability scenario, in which one does not want to modify the types in either legacy system.

PolySPIN has two major tradeoffs as an interoperability approach. The first is the requirement of "compatible" types. This makes PolySPIN better suited for integrating applications like Frank Lloyd's, in which two notions of "employee" are found in distinct languages, than for applications with very dissimilar types.[5] The second tradeoff is that the surrogate objects require extra space. In the worst case, each accessor requires a surrogate for each object in a polylingual system. Thus, the

[5]Then again, applications with very dissimilar types are difficult to integrate no matter what technique is used.

storage overhead is $O(mn)$ for a system with m accessors and n objects. Since the capacity of storage media has increased by a factor of 400 in the past 10 years with no increase in price,[6] PolySPIN's increased storage requirement is a reasonable tradeoff.

A less significant tradeoff is that same-language method calls may incur a slight performance penalty in a polylingual system. Before a method call is made on a class that has been modified by PolySPIN, the Language Arbiter checks whether the call is intralanguage or interlanguage. In practice, this test is done by checking the value of an object attribute, and it adds a small, constant amount of overhead to the call. (Chapter 11 will demonstrate PolySPINner's performance.)

5.4 PolySPINner

PolySPINner is a toolset automating the application of the PolySPIN technique. The version discussed in this chapter, PolySPINner V1, 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 toolset, including its architecture, foundations, and implementation.

PolySPINner operates in the following manner. Given a set of accessors and objects written in programming languages $l_1, l_2, \ldots, l_m$, the interoperability engineer 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 the methods become callable from all languages $l_1, l_2, \ldots, l_m$. Details of the instrumentation will be shown briefly in the example of Section 5.6, and in great detail in Chapter 10 and Appendix C.

---

[6] In 1987, a 50 megabyte hard drive cost $1300. In 1997, the same $1300 buys nearly 20,000 megabytes [34].

In order for PolySPINner to accomplish this instrumentation, each object type t_1 written in language l_1 (without loss of generality) must have corresponding object types $t_2, \ldots, t_m$ in languages $l_2, l_3, \ldots, l_m$ that "match" t_1. Currently, these corresponding types either already exist or must be created by the interoperability engineer, but they could conceivably be generated automatically or semi-automatically with user guidance.[7] This requirement can be fairly heavy for interoperability applications in which few of these matching types already exist, so certain kinds of applications are more appropriate than others for application of the polylingual systems approach. As previously noted, legacy components that address similar semantic domains, such as the banking applications discussed in Chapter 1, are prime candidates as they are more likely to define similar types, such as people or bank accounts.

After PolySPINner has done its work, any accessor of language l_i can invoke methods on an instance of any type t_j, $L(t_j) \neq l_i$, believing it to be an instance of type t_i, $L(t_i) = l_i$. This is accomplished transparently as PolySPINner automatically generates a local, surrogate object of type t_i to stand in for the foreign object of type t_j. (Issues of type safety will be addressed in Chapters 6 and 7.) Since none of the type names or interfaces are modified, the method invocations in application programs (accessors) that previously used these types need no modification in order to interoperate via these types. If the Frank Firm and Lloyd Ltd. implemented their respective concepts of "employee" in distinct languages and with different method calls to access the person's name, age, occupation, and so on, PolySPINner allows an application developed by the Frank Firm for processing its personnel information to be applied to Frank Lloyd's merged database of C++ and CLOS objects with no

[7] Generation of types is conceivable, but generation of classes—types together with *implementations* of their methods—would require language-to-language translation, which is an unsolved problem in general.

visible modifications to the objects and no modifications at all to the application's existing method calls.[8]

5.4.1 PolySPINner Architecture

The architecture of PolySPINner, illustrated in Figure 5.5, demonstrates how each set of type definitions is fed through a Parser component to convert it into a language-independent intermediate representation invisible to the interoperability engineer. Unlike an IDL, PolySPINner's intermediate representation is not exposed[9] and permits language-specific extensions. 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 another language. Finally, the type definitions, Matcher output, and other information are fed to a Generator that creates types that are accessible from various languages.[10] These types have interfaces that are identical to the originals.

PolySPINner is extensible because the Parser, Matcher, and Generator components are generic. By instantiating these components with particular parsing requirements, match criteria, or code-generation information, an interoperability designer can tailor PolySPINner for a given set of languages without affecting PolySPINner's overall structure.

The Parser: PolySPINner's generic Parser component is responsible for managing the parsing of type definitions into PolySPINner's language-independent intermediate

---

[8] As noted in Chapter 1, complete treatment of all database merging issues, such as semantic heterogeneity [67], is beyond the scope of this dissertation.

[9] The representation can be exposed if the interoperability engineer wants to extend PolySPINner or add new type-matching criteria, as in Appendix C.

[10] The components of the PolySPINner implementation (Parser, Matcher, Generator) should not be confused with dimensions of the PolySPIN framework (Locator, Language Arbiter, Communicator, Matcher). The former components are an abstraction of the *pre-compilation* process; the latter dimensions represent the *runtime* behavior of a polylingual system.

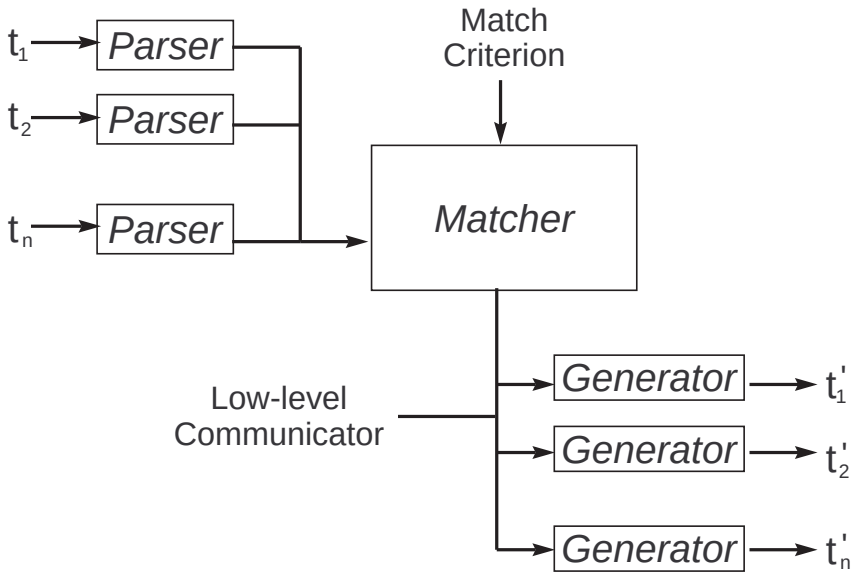


Figure 5.5. The PolySPINner architecture.

representation, which represents object types, methods, and parameters as abstract data types (ADTs). (The intermediate representation is described in detail in Appendix B.) More specific language-dependent information is captured through subtyping of these abstract data types; for example, the "C++ parameter" ADT includes representation for pointers but the "CLOS parameter" ADT does not.

The Matcher: PolySPINner's generic Matcher component is responsible for matching "compatible" types from different languages. What does it mean for two types in different languages to be compatible? 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 components whose types are nearly but not exactly equivalent. More lenient, or *relaxed*, matching criteria are more realistic and useful. Zaremski and Wing [75] have constructed a taxonomy of relaxed type matching criteria within the language ML, as described in Section 4.4.2. Using polylingual system concepts, we have created an

extension of their taxonomy (Chapter 6) that models matching across languages, and PolySPINner V1 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, interoperability engineers 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. Examples of such modifications as they are generated by PolySPINner V1 appear in Section 5.6.

A Generator also generates code for the automatic creation and maintenance of surrogate objects by the Locator. As will be shown in Chapter 10, PolySPINner currently implements the Locator functionality with a language-independent nameserver.

5.5 Example Application of a CORBA-Style Approach

We now describe an application of a representative approach to the "Frank Lloyd" interoperability problem given in Section 5.1. 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 [36, 37].

CORBA-style approaches emphasize the type compatibility dimension of interoperability. They require interoperating components to adhere to a single type model, separate from that of the components' languages. This approach is typified by CORBA [55] and ILU. Software components are considered to be of two kinds: server objects, which provide public interfaces, and clients, which invoke the methods of server objects. These server objects may be written in numerous languages, but a wrapper must be created for each server 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 server object. This interface can then be translated automatically into a client's language, allowing the client to interoperate with the wrapped server objects.

In addition to addressing type compatibility, object wrappers generally encapsulate interlanguage invocation, interlanguage naming, and language information. Thus, these approaches offer some support for all dimensions of polylingual interoperability. They fall short with respect to seamlessness, however, because clients must use two different type systems: their language's type system for accessing local data, and the IDL for accessing server objects.

5.5.1 An Overview of ILU

ILU is an approach to creating interoperating, client-server applications, 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.

```
INTERFACE Employee;

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

Figure 5.6. ISL for `Employee` type.

In ILU, interoperable types are specified using an interface definition language called ISL (Interface Specification Language), its rendition of IDL. Types are described by declaring a type identifier and associating a set of operations with the type. Figure 5.6 shows the ISL for an `Employee` type based on the example scenario.

Given the ISL for a type, an ILU-based polylingual application is created by the following steps:

1. *Create class interfaces for clients.* This is accomplished by applying language-specific translators (provided by the ILU development environment) to the ISL type 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. These generated classes are called *client classes* (classes for use by clients). The ILU translators also produce additional client and server 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 subcomponents of the framework.

2. *Construct the servers:*

   (a) A *server class*, which is a subclass of the client class generated in step 1, must be supplied by the programmer. This class contains the implementations of the 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 creates instances of the server class implementation, publishes names for them in a shared location, and then waits for requests from clients.

3. *Construct the clients.* Clients access server objects by issuing requests, in the form of names or identifiers, to servers. The CLOS office ranking application of Figure 5.1 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 if it is implemented in a different one.

4. *Invoke the servers and the clients.* In our example, each server would manage instances of personnel data, and the client, i.e., the office ranking application, would access the instances via these servers.

5.5.2 An Assessment of ILU

ILU provides support for each of the dimensions in the interoperability framework. A name service, albeit a simple one, allows applications to locate server objects. In addition, the ILU ISL translators generate the required Language Arbiters and Communicators. As noted previously, CORBA-style approaches such as ILU emphasize the type compatibility dimension. ILU imposes a language-external type model on interoperability engineers who must use ISL to describe types instead of using the native (CLOS, C++, etc.) type model. This approach is best suited to the easiest case scenario, and perhaps the common case scenario, described in Section 5.2.1. For example, if it is known *a priori* that two or more components may need to interoperate, then an application can be designed while keeping an ISL description of its types in mind. More problematic is the megaprogramming scenario, as in the Frank

Lloyd example. Using ILU in this case would require the interoperability engineer to translate existing types into 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 components cannot create and access shared objects by using their own type systems. They must use an external, common type model, which typically offers only a subset of the capabilities of the native language's type model, so some types cannot be expressed. As we demonstrate in the forthcoming sections, a major feature of PolySPIN is that no common type model is imposed on interoperability engineers, and therefore no interface language need be used by them. Objects of any type representable by two languages can be communicated polylingually between components of those languages. A related problem is that the common type model is not transparent to the software components; thus, legacy systems must be modified or retrofitted to use the interface language, and programmers must learn and reason about a separate type model. In a polylingual system, each software component can access shared objects via its language's native type system.

5.6 Applying PolySPINner

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 PolySPINner V1 by applying it to the example scenario outlined in Section 5.5.

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, an interoperability engineer would apply PolySPINner to each of the original type definitions for the personnel

data. Figure 5.7 shows portions of the original CLOS and C++ `Employee` types. Note that the method interfaces are slightly different for each of the types. In addition, each of the types is now a subtype of `NameableObject`, a type defined by PolySPIN, which allows objects to participate in its unified name management mechanism.[11]

```
// 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 5.7. Original C++ and CLOS `Employee` classes.

Next, the tool determines whether the types are "compatible" with one another. The rules of compatibility are specified by the interoperability engineer, as was discussed briefly in Section 5.3. The two `Employee` types specify two methods with sig-

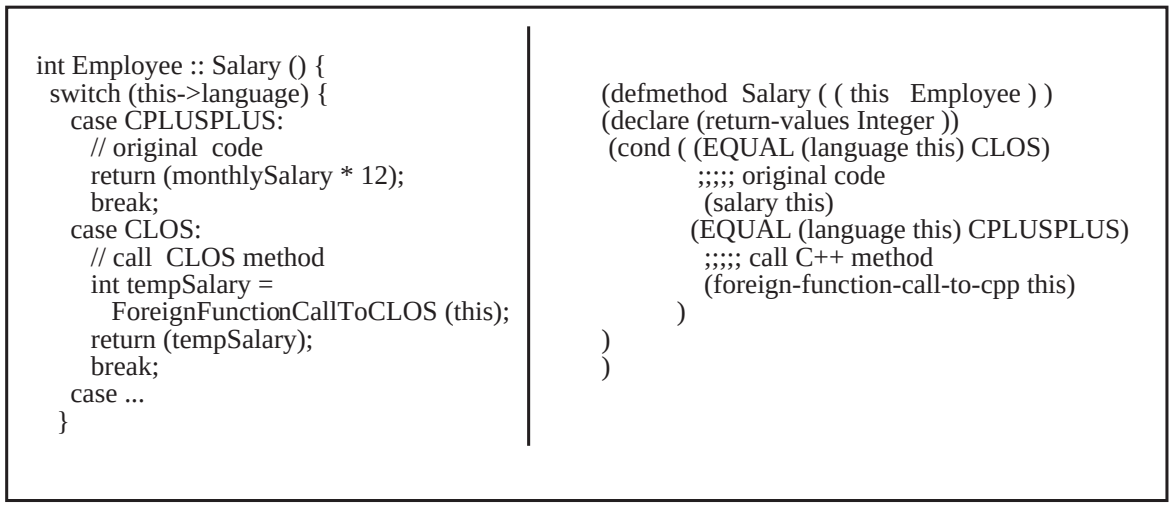
[11] All languages currently supported by PolySPINner have multiple inheritance. To support a language with single inheritance, such as Java, a different implementation would be needed, since one could not simply add a subtype to the original type. Another implementation achieving seamlessness might be to modify the root of the type hierarchy (i.e., `Object`) to contain PolySPINner's name management methods.

natures highly similar to one another—the `YearsOfService` methods, and the `Salary` methods—but each type also has other methods that are not in obvious correspondence. Thus, a plausible compatibility specification for this example is *intersection match*, which is supported by PolySPINner.

As described in Section 5.4, PolySPINner then modifies the implementation for each of the matching methods[12] and generates the necessary communication code enabling the appropriate interlanguage references. PolySPINner V1 produces code based on constructs provided by PolySPIN's naming mechanism and the foreign function interface mechanisms of C++ and CLOS [41]. Figure 5.8 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 to determine whether the object is real or a surrogate. 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 real object's corresponding C++ version of the `Salary` method.

Finally, the generated code (i.e., the modified method implementations) must be compiled and linked with the existing office ranking application. Note that PolySPINner leaves the type name, method names, and method signatures intact, so therefore legacy accessors need not be modified. The only modification that might be necessary to the CLOS main program is the addition of calls to fetch remote objects by name, but this would be required of any interoperability approach and is a basic assumption of Section 1.4.

[12] Unmatched methods will be considered in Chapter 7 with the introduction of the Missing Method Manager.



```
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 5.8. Modified implementation of salary methods.

5.6.1 PolySPINner Tradeoffs

The major tradeoff of PolySPINner, when compared to interoperability approaches that use an IDL, concerns its language-to-language mappings. By using "compatible" pairs of types of different languages, PolySPIN achieves a richness of type compatibility that IDL approaches do not; their "homogenized" intermediate representations represent a least-common-denominator approach to type compatibility. PolySPIN's richness, however, comes at a cost. In an IDL-based approach with support for n languages, n language mappings are required: one between each language and the IDL. PolySPIN requires $n(n-1)/2 = O(n^2)$ language-to-language mappings: one between each pair of languages, as shown in Figure 5.9.

However, the number of languages supported, n, is likely to be small. PolySPINner V1 supports only two languages, but even if it supported five, this would mean $(5)(4)/2 = 10$ language mappings instead of five, which is arguably a reasonable tradeoff for the increased richness of typing and transparency of interoperability provided by PolySPIN. In addition, these language-to-language mappings are created only once for each language pair, and they will typically be done by the interoper-

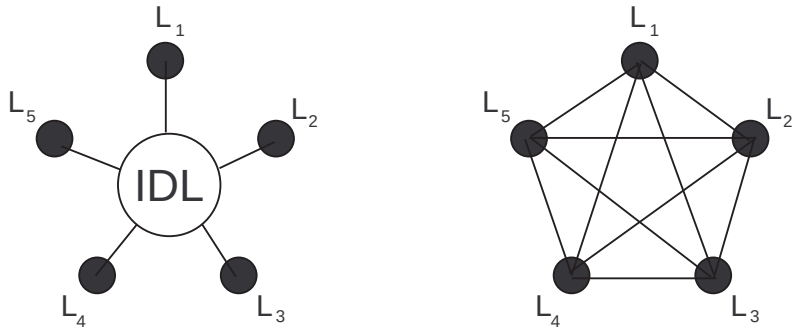


Figure 5.9. IDL-based approach (left) requires $n = 5$ language mappings, whereas PolySPIN-based approach (right) requires $n(n-1)/2 = 10$ mappings.

ability designer who maintains PolySPINner, not the interoperability engineer who builds systems with it. Once these language mappings are made, they are reused indefinitely. These mappings impact both the Matcher and Generator components, as shown in Section C.7.3 on creating a language binding.

5.6.2 A Comparison with ILU

Both PolySPIN and ILU provide support for each of the dimensions in the framework for polylingual interoperability. PolySPIN's name management mechanism [40] provides a much richer naming service than does ILU's. In addition, PolySPIN does not impose a language-external type model on interoperability engineers, as ILU does. Instead of ISL, PolySPIN permits developers to use their native type models (those of CLOS, C++, etc.) 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. Relaxed matching may compromise type safety, however, as discussed in Chapter 6.

5.7 Summary

Existing approaches to interoperability are not sufficiently seamless. If software components are required to use invocation mechanisms or type models 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 components, 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 interoperability among legacy systems.

This chapter has described the PolySPIN approach and introduced the PolySPINner toolkit for automating seamless interoperability in polylingual systems. Since this approach evolved from work on name management in persistent object systems, PolySPINner V1 relies upon features of the Open Object-Oriented Database (OpenOODB) [72] and exploits inheritance of name management-associated capabilities in implementing seamless interoperability. The approach, however, does not depend on OpenOODB, persistence in general, nor inheritance.

In the chapters to come, PolySPIN and PolySPINner will be explained in increasing detail. In addition, more sophisticated issues and features, such as interlanguage type matching and support for transient objects, will be presented. This chapter serves as an overview of the fundamental capabilities of PolySPIN.

CHAPTER 6

TYPE MATCHING IN POLYLINGUAL SYSTEMS

PolySPIN relies fundamentally on the ability to determine if two types, defined using two distinct programming languages, are *compatible*. In a polylingual system, two types t and t' of distinct languages are compatible if, to an application program, instances of t and t' appear to be of the same type and can be manipulated seamlessly as if they were of the same type. The term "compatible" will be used informally until Section 6.2, where it will be formally defined.

PolySPIN's style of determining type compatibility, as used by the initial versions of PolySPIN and PolySPINner described in Chapter 5, lacked generalizable, formal foundations. In particular, it did not address the polymorphism found in object-oriented programming languages, an unfortunate omission as PolySPINner was originally developed to support interoperability between object-oriented languages, namely C++ and CLOS. PolySPINner V1 was successfully applied to some examples of C++/CLOS interoperability in Chapter 5, but without a satisfying level of generality.

Ad hoc approaches to type compatibility impose unacceptable limitations not only on PolySPINner but also on interoperability in general. They also fail to provide a solid basis for supporting polylingual interoperability among components written in a wider variety of languages.

In this chapter, we discuss how concepts from signature matching [75] and object-oriented type theory [1] can be applied to provide a suitable formal foundation for interlanguage type compatibility in polylingual systems. We also describe how this

```
class House {
  public:
    int SquareFeet();
    int Bedrooms();
};
```

Figure 6.1. C++ `House` class.

```
(defclass House ... );
(defmethod SquareFeet ((this House))
   (declare (return-values Integer)) ...);
(defmethod Bedrooms ((this House))
   (declare (return-values Integer))...);
```

Figure 6.2. CLOS `House` class.

foundation serves to guide enhancements to PolySPIN, notably by greatly improving its treatment of subtype polymorphism. Finally we discuss the role of this foundation as a basis for extending PolySPINner.

The version of PolySPIN discussed in Chapter 5 will be referred to as PolySPIN V1. The extended PolySPIN described in this chapter will be called PolySPIN V2. Kaplan's original PolySPIN is designated PolySPIN V0.

6.1 Motivation and Background

Suppose that a rudimentary real estate application, written in C++, manipulates descriptions of houses that are defined by the `House` class shown in Figure 6.1. A user wishes to apply this program to a set of house descriptions implemented in CLOS, as in Figure 6.2. The user also wishes to apply the program to another set of CLOS house descriptions, implemented as in Figure 6.3.

```
(defclass Building ...);
(defmethod SquareFeet ((this Building))
   (declare (return-values Integer)) ...);

(defclass House (Building) ... );
(defmethod Bedrooms ((this House))
   (declare (return-values Integer))...);
```

Figure 6.3. Another CLOS `House` class and its superclass, Building.

The `House` type definitions shown in the three figures are obviously similar. Each is defined by an interface having two methods, `SquareFeet` and `Bedrooms`, and the signatures of the methods are, modulo language differences, identical. The two CLOS definitions differ only in that the second (Figure 6.3) defines some of the methods associated with `House` objects in a more general supertype, called `Building`, whose methods are inherited by type `House`. Thus, all three `House` definitions could reasonably be considered "compatible" with one another, in the sense that the signatures of their applicable methods are highly similar.

PolySPIN V1 can easily determine that the type definitions shown in Figure 6.1 and Figure 6.2 are "compatible." PolySPINner V1, the toolset implementing PolySPIN V1, when presented with these two definitions, can confirm that the two types match closely and can produce modified versions of the code implementing these two types, allowing the unmodified real estate program to access C++ and CLOS instances of `House` transparently [40].

PolySPINner V1 can determine type compatibility and generate modified type implementations in more complex cases, such as when one type's methods match only a subset of the other's methods, as illustrated in Chapter 5. Diverse criteria for type compatibility, and associated code-generation modules, may be plugged into PolySPINner V1's generic architecture. A library of useful type-compatibility criteria

is available, and users may also create their own (using C++), thus obtaining some control over this aspect of PolySPINner's functionality.

This flexibility, however, is both limited and undisciplined. It is limited in that inter-type relationships, such as the inheritance relationship of Figure 6.3, go unnoticed. When presented with the type definitions of Figures 6.1 and 6.3, PolySPINner V1 will confirm that a *subset* compatibility exists between the two definitions of type `House`, since both provide compatible `Bedrooms` methods, and will produce suitably modified versions of their implementations to support polylingual usage of the `Bedrooms` method. PolySPINner V1 fails to determine that the two definitions of type `House` have a stronger compatibility via inheritance and cannot produce modified versions of the types to support polylingual usage of `SquareFeet`.

The flexibility of PolySPINner V1 is also undisciplined in that it is not based on a formal notion of polylingual type compatibility. Instead, compatibility may be defined as any criterion an interoperability engineer chooses to code in C++. While we argued in Chapter 5 that such flexibility is desirable, the potential is great for type safety to be violated. For example, the engineer could naively decree compatibility between the C++ `House` type of Figure 6.1 and the CLOS `House` type of Figure 6.4, ignoring the possibility that an application program might attempt to invoke the CLOS `LockDoors` method on an instance of the C++ type, which would be a type error.

The remainder of this chapter describes a formal foundation for type compatibility aimed at overcoming such difficulties in polylingual systems. We begin in Section 6.2 by defining "compatible" and related terms. Next, in Section 6.3 we introduce two type-related models and extend them to work with multiple languages. Section 6.4 explains how these two models will be used to formalize polylingual systems. Section 6.5 formalizes the notion of *polylingual method matching* that is necessary for the subsequent formalizations of *polylingual type matching* in Section 6.6.

```
(defclass House ... );
(defmethod SquareFeet ((this House))
   (declare (return-values Integer)) ...);
(defmethod Bedrooms ((this House))
   (declare (return-values Integer))...);
(defmethod LockDoors ((this House))
   (declare (return-values Void))...);
```

Figure 6.4. CLOS class with an additional method, `LockDoors`.

6.2 Definitions

Thus far, the term "compatible" has been used without a formal definition. Here we define compatibility with respect to *conformance* between types, that is, the ability of one type to stand in for another.

Definition 6.1 (Conforms To) *Type t'* **conforms to** *type t if an instance of t' can be used in place of an instance of t in any context in which an instance of type t is valid.*

As shall be seen, the `House` type of Figure 6.1 conforms to the `House` type of Figure 6.2. Conformance is not just subtyping: it crosses programming languages, and it covers relationships between types other than subtyping.

Definition 6.2 (Conformant) *Two types t and t' are* **conformant** *if either t conforms to t', or t' conforms to t.*

The term "conformant" can introduce ambiguity because it does not specify which of two types conforms to which; however, the term proves to be useful on occasion.

Conformance between types is not necessarily symmetric. If type t conforms to type t', the reverse is not necessarily true. If each of two types conforms to the other, we call the types *compatible*.

Definition 6.3 (Compatible) *Two types t and t' are* **compatible** *if type t conforms to type t', and type t' conforms to type t.*

As shall be seen, the two `House` types of Figures 6.1 and 6.2 are compatible, as are those of Figures 6.1 and 6.3. The types `Building` and `House` of Figure 6.3 are not compatible: an instance of `Building` cannot stand in for an instance of `House` because `Building` lacks the method `Bedrooms`. In this figure, however, type `House` conforms to type `Building`.

6.3 Choosing a Framework for Polylingual Type Matching

Type theory is important for a thorough understanding of type-related issues in programming languages. It addresses the structure of types, semantic relationships among types, type safety, and related issues in a formal manner without getting bogged down in syntax details and compiler vagaries.

In a polylingual system, a given object of language l may be accessed by software components implemented in languages other than l. Conversely, a given component may access objects implemented in languages other than its own. Since different languages may have different type models and data representations, a notion of interlanguage type compatibility is necessary in order for polylingual systems to share objects of distinct languages. Interlanguage type compatibility is different from other well-studied relationships among types, such as subtyping, inheritance, and genericity, though it may be related to them. It is therefore highly beneficial to have a theory of polylingual types in order to ensure type safety in polylingual systems.

In this section, we provide an overview of two prominent type-related theories:

- Abadi and Cardelli's theory of objects [1].
- Zaremski and Wing's relaxed matching taxonomy [75].

and combine and extend them for use in modeling type compatibility in polylingual systems.

6.3.1 Abadi and Cardelli's Theory of Objects

The Abadi/Cardelli (A/C) model is a calculus for modeling types, classes, and objects found in object-oriented languages. It models both untyped (not useful for our present purposes) and typed objects, the latter with and without subtyping. First-order models address traditional object-oriented issues of subtyping, inheritance, and recursive types, while second-order models focus on quantified types (e.g., parametric polymorphism, generics) and self types.

The A/C model specifies its features in terms of *judgments*: expressions of the form $E \vdash X$, in which E is an environment and X is a fact deducible from that environment. For example,

$$E \vdash A <: B \tag{6.1}$$

denotes that in environment E, it is deducible ($\vdash$) that type A is a subtype ($<:$) of type B. Alternatively one can say that the statement $A<:B$ is "well-formed in environment E." A *type property* is modeled as a set of premise judgments, written above a horizontal bar, followed by a conclusion judgment below the bar. For example, a rule for subsumption is

$$\frac{E \vdash A \quad E \vdash B \quad E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B} \tag{6.2}$$

denoting that if A and B are types, and a is of type A, and A is a subtype of B, then a is also of type B.

The A/C model is designed for object-oriented languages, making it attractive for describing types in polylingual systems. It has no facility, however, for representing objects or types of distinct languages. We make a small extension to the A/C model so

it can include language information in judgments. We extend the judgment notation $E \vdash T$, which means "in environment E, type T is well-formed," to be $E \vdash_L T$, which means "in environment E, type T is well-formed in language L." If language is irrelevant, we omit the L subscript and get back the original A/C notation and semantics. The use of this extended notation will be demonstrated in Section 6.6.

The A/C model also does not have any way to represent relaxed matches. For this, we turn to the work of Zaremski and Wing.

6.3.2 Zaremski and Wing's Relaxed Matching Taxonomy

The Zaremski/Wing (Z/W) model is not a type theory, but a taxonomy of matching strategies for similar but not necessarily identical types. The model is patterned after the typing issues of the language ML [53] and does not use an object-oriented model, so there are no notions of inheritance and subtyping. ML modules are analogous to classes, and ML functions to methods, so an application to the object-oriented domain is reasonable.

The model is inspired by a problem in the domain of software reuse: how to locate a type, within a large type library, that fits the needs of a software developer. Zaremski and Wing first considered the problem of signature matching [75]: given the signature of a function f, and given a library of functions, locate a function f' in the library whose signature "matches" that of f. The hope is that the semantics of function f' will be the same as, or at least related to, those of function f, so that f' may be reused to implement f. Zaremski and Wing later considered specification matching [76], in which algebraic specifications, rather than signatures, are the criteria used to match pairs of functions.

The heart of the Z/W model is its notion of relaxed matching. The authors asked the question: how dissimilar can function signatures be, while still being similar enough to be considered a "match?" In other words, what are some reasonable,

natural differences that may arise in practice between signatures of similar functions, and how might they be formalized? For example, one programmer might implement a file input/output function with the signature <file descriptor, data pointer>, whereas another might use <data pointer, file descriptor>, and a third might use <data pointer, data length, file descriptor>.

Zaremski and Wing produced a taxonomy of such differences between signatures, which they termed *relaxed matches*, along with a formal model for representing and combining matches. Some matches were *transformations*, such as parameter reordering; some were *relationships*, such as syntactic equality; and some were combinations of transformations and relationships. Noticeably absent is subtyping, most likely because Zaremski and Wing's language of choice, ML, does not support subtypes. Table 6.1 lists their matching criteria.

Generic matching is defined as a function on two types τ_l and τ_q:[1]

$$M(\tau_l, \tau_q) = T_l(\tau_l)\ \mathcal{R}\ T_q(\tau_q) \tag{6.3}$$

T_l and T_q are transformations on types τ_l and τ_q, respectively. Transformations include reordering of methods, currying, etc. $\mathcal{R}$ is a relationship between types, such as matching a subset of functions. Using generic matching as a starting point, Zaremski and Wing define more specific relaxed matching predicates such as those in Table 6.1.

Like the A/C model, the Z/W taxonomy is not designed for multiple languages. Consider two natural ways in which Z/W matching could be extended to multilanguage matching without affecting the rest of the model. We will argue that one means of extension is preferable to the other. The first is to extend the set of transformations to include transformations from one language into another. This could be

[1]The subscripts l and q come from software reuse: τ_l is a type in a type *library*, and τ_q is the software developer's *query* that may or may not match type τ_l. For our purposes in the domain of polylingual systems, τ_l and τ_q are simply two types.

Table 6.1. Zaremski and Wing's matching criteria for functions (analogous to methods) and modules (analogous to types).

Function Match Relationships	
Exact Match	Signatures match except for parameter renaming.
Generalized Match	Signatures match if there exists a sequence of generalizing type substitutions (e.g., replace "sequence of integer" by "sequence of any type") on the parameters to make the signatures identical.
Specialized Match	Signatures match if there exists a sequence of restrictive type substitutions (e.g., replace "sequence of any type" by "sequence of integer") on the parameters to make the signatures identical.
Function Match Transformations	
Reorder	The parameter types of one function, when permuted, match those of another function.
Uncurry	The signature of one function, when curried, matches the signature of another.
Module (Type) Match Relationships	
Exact Match	There exists a mapping between the functions defined within one module and those defined within another, in which each function in the first module exactly matches a function in the second, and the mapping is a bijection.
Generalized Match	Same as exact match, but the mapping is not onto.
Specialized Match	Same as exact match, but the mapping is not one-to-one.

represented by introducing a new pair of transformations, I_l and I_q (where I stands for interoperability), to the definition of generic matching, like so:

$$M(\tau_l, \tau_q) = T_l(I_l(\tau_l)) \ \mathcal{R} \ T_q(I_l(\tau_q)) \tag{6.4}$$

Given that types τ_l and τ_q are in distinct languages, transformations I_l and I_q are applied to τ_l and τ_q, respectively, to handle interlanguage matching issues. We apply the interoperability transformations before the original transformations in order to shield relationship $\mathcal{R}$ from knowledge of interoperability issues, in keeping with the principle of separation of concerns.

This extension has an undesirable property, however. The original transformations T_l and T_q expect their input types to be of the same language. In order to preserve this property transparently, transformations I_l and I_q must produce types of the same language. This is effectively modeling the IDL approach of using an intermediate language for types, which we argued against in Chapter 5.

An alternative way to extend the Z/W model to multilanguage matching is to extend the relationship, $\mathcal{R}$, to include relationships between languages. We define $\mathcal{R}'$ to be a composition of three functions: a relationship $\mathcal{R}$ as before, and two new interoperability functions I_1 and I_2, applied before and after $\mathcal{R}$, respectively.

$$\mathcal{R}'(x, y) = I_2(\mathcal{R}(I_1(x, y))) \tag{6.5}$$

Rather than choose whether such multilanguage relationships should be applied before or after $\mathcal{R}$, we allow for both possibilities through relationships I_1 and I_2, applied before and after $\mathcal{R}$, respectively. (Either I_1 or I_2 or both could be eliminated by defining them to be the identity relationship.) Thus, our definition of extended generic matching would be:

$$\begin{aligned} M(\tau_l, \tau_q) &= T_l(\tau_l)\ \mathcal{R}'\ T_q(\tau_q) \\ \mathcal{R}'(x, y) &= I_2(\mathcal{R}(I_1(x, y))) \end{aligned} \tag{6.6}$$

This extension via relationships does not require types τ_l and τ_q to be translated into the same language (although it does not preclude the possibility either). Thus, extension via relationships is preferable to the previous extension via transformations.[2]

Another issue concerns renaming of parameters and methods. In the Z/W model, an exact match permits renaming of parameters, but in polylingual systems, we define exact match not to permit renaming. Instead, we consider parameter renaming to be a form of relaxed match (a transformation).

Finally, the Z/W generic match function returns a Boolean value indicating the presence (True) or absence (False) of a match. It would be useful to expand this return value into a more complex object, such as a numeric compatibility rating, or a listing of the components that did and didn't match (and reasons why). Thus, we define the function $M(\tau_l, \tau_q)$ to have an abstract return type `MatchResult` rather than Boolean. `MatchResult` is defined in Chapter 9.

6.4 Perspectives on Polylingual Type Compatibility

Having described the foundations needed for our model of polylingual type matching, we provide an overview of our plans for using the A/C model and the Z/W taxonomy.

6.4.1 The Matching Perspective

From one perspective, the determination of polylingual type compatibility is a question of *matching* two type definitions written in two distinct languages. This

[2] An equivalent way to view this choice is that interoperability transformations should be applied outside the T_i transformations, not inside: $M(\tau_l, \tau_q) = I_l(T_l(\tau_l))\ \mathcal{R}'\ I_q(T_q(\tau_q))$.

might well be the view taken by a software developer attempting to determine whether some existing piece of code, such as the C++ and CLOS types of the real estate application described in Section 6.1, can be made to interoperate. It is therefore not surprising that our development of foundations for polylingual type compatibility is heavily influenced by the work of Zaremski and Wing.

PolySPIN V0 [40] required types to match exactly: identical names, and identical methods with identical signatures, modulo differences in language syntax. An example is the C++ and CLOS definitions of `House` in Figures 6.1 and 6.2. Still, this "exact" match has subtleties. For example, the self parameter in C++ methods ("this") is implicit, whereas it is declared explicitly in CLOS. An interlanguage method matching model must take such subtleties into account.

PolySPINner V1 had a library of relaxed matching criteria inspired by the Z/W taxonomy and also permitted arbitrary relaxed matches in a type-unsafe manner. The goal of PolySPIN (and PolySPINner) V2 is increased type safety while still supporting useful relaxed matches.

Following the lead of Zaremski and Wing, we will first formalize *method matching* (which they term *function matching*), then consider various kinds of *type matching* (which they term *module matching*) to be defined in terms of particular kinds of method matching. Also in the spirit of their work, we will differentiate *exact* matches from *relaxed* matches. Zaremski and Wing have also extended their approach to encompass specification matching [76], which relies upon formal specifications of program semantics. This area has also been explored in the object-oriented domain by Liskov and Wing's work on behavioral subtyping [49]. This dissertation considers only signature matching, leaving polylingual specification matching for future work.

While similar in outline, our work differs from that of Zaremski and Wing in some important respects. First, where their definitions of matching all assumed that the entities being matched were defined in a single language, our definitions assume the

existence of multiple languages. Second, because their examples and implementation were all developed in the language ML, the kinds of polymorphism they addressed were quite different from those that are of most immediate concern to us. ML has a well-developed notion of *parametric* polymorphism, but, not being an object-oriented language, does not admit *subtype* polymorphism. Our primary concern, on the other hand, as illustrated by the `House` example in Section 6.1, is the correct handling of subtype polymorphism. Although the C++ template facility provides a limited form of parametric polymorphism, we choose to defer treating it in our formalization until we extend PolySPIN to incorporate another language, such as Ada95 or Java, that also provides parametric polymorphism (or plans to provide it, in the case of Java [2]).

6.4.2 The Type Checking Perspective

From another perspective, the determination of polylingual type compatibility is an extension of *type checking* to a setting where type definitions are written in different languages. In this view, the objective is to ensure the absence of type errors in a polylingual program. This might well be the view taken by a software developer assembling a polylingual system from fragments that, taken together, meet the system requirements. In this situation, the developer should be primarily interested in the kinds of type safety assurances that can (or cannot) be provided about the resulting polylingual program, and the level of confidence one can have in those assurances.

Toward this end, some concepts from the Abadi and Cardelli model will be used. We, like these theorists, are attempting to establish a language independent framework in which to understand the extent and limitations of type safety assurances that we can obtain under various assumptions. Similarly, we also focus particularly on the implications of subtype polymorphism, unlike the work of Zaremski and Wing. Our work is primarily distinguished and motivated by a desire to apply this framework to

(pairwise) specific, existing languages. We therefore defer, for the moment, some of the more esoteric issues addressed in some areas of object-oriented type theory, and devote the majority of our attention to issues relevant to interoperability between components written in C++ and CLOS. Other object-oriented languages are in the process of being incorporated into PolySPIN, notably Java, and the framework must be sufficiently general to handle such additional languages.

6.5 Polylingual Method Matching

We now develop a formal foundation for the determination of type compatibility in polylingual systems. This section covers method matching, and Section 6.6 covers type matching.

Following Zaremski and Wing, we initially define a generic match predicate for methods, then define exact matching, and finally examine several notions of relaxed matching. Also like Zaremski and Wing, we define type matching in terms of method matching. An important difference between our work and that of Zaremski and Wing, however, is that some of our method matches will also be defined in terms of type matches, in order to treat the subtype polymorphism that is characteristic of object-oriented languages. Since their approach was monolingual and lacked subtyping, no such mutual dependencies between method matching and type matching arose. Adapting Zaremski and Wing's notation, we define a generic form of method match, M, between m_X, a method written in language X, and m_Y, a method written in language Y, as a predicate:

Definition 6.4 (Generic Method Match)

$M(m_X, m_Y) \equiv \exists T_X \text{ and } T_Y \text{ such that } T_X(m_X)\ \mathcal{R}\ T_Y(m_Y)$.

T_X and T_Y are transformations (e.g., parameter reordering) applied to the method signatures written in language X and language Y, respectively, and $\mathcal{R}$ is some relationship between method signatures. By defining a generic kind of method match,

we establish a common ground for more specific kinds of method match. By considering various values of the transformations T_i and the relationship $\mathcal{R}$, we can define different, more specific kinds of matches in terms of Generic Method Match.

A given relationship $\mathcal{R}$ is not necessarily symmetric, that is, the statement $T_X(m_X)\ \mathcal{R}\ T_Y(m_Y)$ does not necessarily imply that $T_Y(m_Y)\ \mathcal{R}\ T_X(m_X)$. We will examine the symmetry or asymmetry of the relationships in each of the matching predicates that follow. Asymmetric relationships are not necessarily type-safe, so additional steps are required in order to ensure that type errors do not occur, as we shall see in Chapter 7.

6.5.1 Exact Match

Zaremski and Wing define an *exact* match as one in which the only transformations permitted are parameter renamings, and the relationship $\mathcal{R}$ is type equality. For example, suppose our C++ `House` class had a method `InstallAlarm` that would install a burglar alarm in a house at a specified location:

```
void InstallAlarm(Alarm, Location)
```

The following C++ methods would be considered an exact match:

```
void InstallAlarm(Alarm a, Location x)
void InstallAlarm(Alarm b, Location y)
```

This definition of matching is symmetric and type-safe because type equality is trivially type-safe, and parameter renaming cannot affect type safety. However, exact match is not expressive enough to define matching between methods of different languages, since type equality between languages is a nontrivial issue. Another problem is that we desire a kind of match that forbids parameter renaming, in order to make "exact" match be as exact as possible. To address this, we model renaming as a

transformation, rather than a part of exact match,[3] and propose two, more realistic, definitions of exact method match.

Recall that many approaches to interoperability, such as CORBA [55] and ILU [37], attempt to simplify type matching by introducing an additional type definition notation: a so-called "interface definition language" (IDL) [55] or "unifying type model" (UTM) [74]. Such approaches transform both method signatures into this common notation, and consider types to be equal if their IDL representations match exactly. This is notated as a match predicate, $M_{exactIDL}$:

Definition 6.5 (IDL Exact Method Match)

$M_{exactIDL}(m_X, m_Y) \equiv \exists T_{X \to IDL}$ *and* $T_{Y \to IDL}$ *such that*

$T_{X \to IDL}(m_X) =_{IDL} T_{Y \to IDL}(m_Y)$.

This definition models the conversion from a type's native language (X or Y) into IDL as a transformation ($T_{X \to IDL}$ and $T_{Y \to IDL}$), and the relationship $=_{IDL}$ is type equality between IDL types, i.e., identity.

For example, the C++ `House` type of Figure 6.1 and the CLOS `House` type of Figure 6.2 could each be represented, minimally, by the following CORBA IDL interface:

```
// CORBA IDL
interface House {
  integer Bedrooms();
}
```

in which the CORBA `Bedrooms` method gets mapped to the C++ and CLOS `Bedrooms` methods.[4] Thus, the C++ and CLOS `Bedrooms` methods would be considered to

[3] A possibility that Zaremski and Wing also considered [75].

[4] A CORBA IDL interface need not provide IDL mappings for every method of a type.

match exactly with respect to $M_{exactIDL}$. This method match is symmetric and type-safe because all invocations of methods occur through their CORBA IDL interfaces, and type-safety is a prerequisite between an IDL interface and its underlying, native language method. (That is, one cannot compile a CORBA application unless all its IDL interfaces have type-safe mappings to their underlying methods in each language.)

Although this approach may be appealing, it is limiting since it restricts matching to a least common denominator, that of the IDL. (It corresponds to the first, less desirable approach given in Section 6.3.2.) Thus, such an approach either demands an implausibly general and powerful IDL that encompasses every feature of every language, or it rules out perfectly reasonable matches in cases where both languages X and Y support some feature that the IDL does not. For example, C and C++ both have a pointer type, and both languages have mappings to CORBA IDL, but CORBA IDL does not have a pointer type. Thus, C and C++ programs have no direct way to communicate pointers via CORBA, even if C and C++ pointers are represented identically.

The approach we have adopted in PolySPIN is to create transformations on method signatures for each pair of languages, so that the signatures have a common "shape," and then use a definition of equality specific to that language pair. This approach is represented by a match predicate $M_{exactX:Y}$:

Definition 6.6 (PolySPIN Exact Method Match)

$M_{exactX:Y}(m_X, m_Y) \equiv \exists T_{X:Y}$ *and* $T_{Y:X}$ *such that* $T_{X:Y}(m_X) =_{X:Y} T_{Y:X}(m_Y)$.

In this definition, there is no central, foreign type system such as that of an IDL. As a result, the given transformations and type equality relationship may be complex, as we shall see. Note that this definition corresponds to the second approach given in Equation 6.5, Section 6.3.2, in which the interoperability function I_1 is the application of $T_{X:Y}$ to m_X and $T_{Y:X}$ to m_Y, respectively, I_2 is the identity function, and $\mathcal{R}$ is $=_{X:Y}$, a type equality relationship that is aware of language differences. As with

Exact Match, PolySPIN Exact Method Match is symmetric and type-safe because the relationship $=_{X:Y}$ is type equality modulo language syntax differences.

In the case of C++ and CLOS, for example, we could define a transformation $T_{C++:CLOS}$ as follows:[5]

1. We begin with a C++ method, m.

2. An additional formal parameter c *this* is inserted first in the parameter list of m, where c is the name of the C++ class. This accounts for the self parameter, which is implicit in C++ but explicit in CLOS.

3. Every C++ formal parameter declaration t x, where t is the parameter type and x is the parameter name, is transformed into a CLOS formal parameter declaration $(x\ t)$. If the C++ formal parameters have their names omitted (which is legal in function prototypes), random names are generated.[6]

4. Transform the C++ return type t into a CLOS return type declaration, `(declare (return-values` t`))`, and place it immediately after the formal parameter list.

For example, consider the C++ signature of `SquareFeet` from Figure 6.1:

```
int SquareFeet()
```

It is transformed via $T_{C++:CLOS}$ by the following steps, corresponding to those above:

1. `int SquareFeet ()`

2. `int SquareFeet (House this)`

[5]Such a transformation could be done in many other ways. This is just one way that works. The PolySPINner implementation transforms type definitions into abstract syntax trees as shown in Chapter 9.

[6]If this occurs, exact match will likely fail because parameter renamings are not tolerated.

3. `int SquareFeet ((this House))`

4. `SquareFeet ((this House)) (declare (return-values int))`

Likewise, we define $T_{CLOS:C++}$ to transform the CLOS declaration, stripping away the word `defmethod`, the outermost parentheses, and the method body except for the return type declaration. Thus, the CLOS `SquareFeet` method of Figure 6.2 is transformed into:

```
SquareFeet ((this House)) (declare (return-values Integer))
```

Finally, we define a language-pair-specific type equality function ($=_{C++:CLOS}$) that requires the two methods to have the same names, parameters, and return types. This also includes definitions of type equality between the primitive types of the two languages, such as the C++ `int` and CLOS `Integer` types. In our example, we have:

`SquareFeet ((this House)) (declare (return-values int))`

$=_{C++:CLOS}$

`SquareFeet ((this House)) (declare (return-values Integer))`

Thus, our example has shown that the C++ and CLOS `SquareFeet` methods match exactly, so we may write:

$M_{exact C++:CLOS}$(`int SquareFeet(),`

`(defmethod SquareFeet ((this House)) (declare (return-values Integer))`

`...)`)

Note that in the PolySPIN framework, a type equality relationship must be defined between each pair of languages, such as C++ and CLOS. In the above example, we called this relationship $=_{C++:CLOS}$. A strength of PolySPIN is that such relationships are defined only once per language pair, not once per application of PolySPIN. Thus,

Table 6.2. Some relaxed matching transformations.

Transformation	Meaning
$T_{\pi}(m)$	Apply permutation π to m's parameters
$T_{[\alpha/\beta]}(m)$	Replace name β with name α in m

all subsequent definitions of C++/CLOS type equality in this chapter will use the same $=_{C++:CLOS}$ relationship.

Note also that the division between the functionality of transformations and relationships is not set in stone. Zaremski and Wing provide both transformations and relationships, rather than just one or the other, because both are useful tools.

6.5.2 Relaxed Match

In practice, exact matches rarely occur between types from different applications, let alone types of different languages. A significant limitation of Kaplan's original PolySPINner V0 [40] is that it allowed only exact method matches in its type compatibility determinations. PolySPINner V1 began to address *relaxed* matching, and in this section, we consider the subject more thoroughly by examining useful relaxed matches. Such matches can be expressed by choosing appropriate values for the transformations T_i and the relationship $\mathcal{R}$ in Definition 6.4.

6.5.2.1 Renaming and Reordering

Similar methods may be named differently, and the parameters in their signatures may have different names and/or occur in different orders. Each of these possibilities represents an opportunity for a relaxed match, and can be expressed as transformations, as given in Table 6.2, or as a composition of such transformations.

Under the standard assumption that renamings are not permitted to introduce ambiguities by assigning additional meanings to names already in use (i.e., $T_{[\alpha/\beta]}(m)$ is not defined if the name α already appears in method m), compositions of these

transformations can allow for arbitrary renamings and reorderings in method signatures. Moreover, composing these transformations with those defined for exact matching (Section 6.5.1) will yield relaxed polylingual method matches. For example, a predicate for method matching with reordered parameters can be defined using the transformation T_π from Table 6.2:

Definition 6.7 (PolySPIN Method Match with Reordered Parameters) $M_{reorder X:Y}(m_X, m_Y) \equiv \exists T_{X:Y}$ *and* $T_{Y:X}$ *and* T_π *such that* $T_\pi(T_{X:Y}(m_X)) =_{X:Y} T_{Y:X}(m_Y)$.

For example, the `House` methods

```
void InstallAlarm(Alarm a, Location x)
(defmethod InstallAlarm ((this House) (x Location) (a Alarm)) ...)
```

match exactly except for a permutation of the parameters, given the same transformations $T_{C{+}{+}:CLOS}$ and $T_{CLOS:C{+}{+}}$ used in Section 6.5.1, composed with a permutation transformation T_π that interchanges the `Alarm` and `Location` parameters. Similar definitions can be made for method match with renamed method parameters, and for method match with reordered and renamed parameters, a composition of the previous two match predicates.

It is important to note that parameter reorderings do not produce unique results if two parameters of a method have the same type. For example, given a method with signature (t_1, t_2, t_1), there are two different permutations of the parameters that produce the signature (t_1, t_1, t_2).[7] A tool for polylingual method matching, based completely on signatures and not semantics, would require human assistance to determine which of two reorderings is superior if they have the same parameter types

[7]Disambiguation might be possible by examining the parameter names.

in the same order. PolySPINner has a user interface to allow such human guidance, described in Chapter 9.

Method matches involving only method renamings and parameter reorderings are symmetric and type-safe. No type information is lost, and the mappings between the two method names and the two parameter lists are bijections.

6.5.2.2 Parameter Subset and Superset

Another category of relaxed method match permits the signatures of methods m_X and m_Y to have different numbers of parameters. For example, a parameter of method m_X could encode the same information as some set of parameters of m_Y, as considered by Konstantas [43]. Consider the following methods that both assign a telephone number to a house:

```
void SetPhone(int number[9])
void SetPhone(int areacode[3], int exchange[3], int rest[4])
```

The first represents the telephone number as a single array of digits. The second represents the same number as a three-digit area code, a three-digit exchange, and a four-digit suffix. Konstantas defined a language of transformations from one set of parameters into another to handle cases like these.

In our multilanguage setting, the same concepts could be applied, and we could potentially formulate definitions of signature matching for such situations. They are of limited utility, however, since semantic analysis is necessary to determine whether two sets of different parameter types encode the same information. We therefore omit such definitions here and defer treatment of these situations to future work on polylingual specification matching (Chapter 12). In addition, Section 7.3 addresses the use of actual parameters in general for determining type compatibility.

Parameter superset and parameter subset method matches are not symmetric and therefore not necessarily type-safe. If method m is matched to a method m'

in another language, and the parameters of m' are a superset of those of m, then a call to m might not be convertible into an appropriate call to m'. In this case, PolySPIN permits default values to be specified for the additional parameters in m' and thus preserve type safety. PolySPINner implements this functionality via its Missing Method Manager, described in Section 7.4.1.

6.5.2.3 Parameter Subtype and Supertype

For object-oriented method matching, the types of various parameters of method m_X might be subtypes or supertypes of corresponding[8] parameters of method m_Y. Once again, we can describe such situations in terms of transformations. In particular, we define:

$T_{<:[\alpha:\gamma/\delta]}(m)$	Transform the type of parameter α in method m from δ to γ, where γ is a subtype of δ.
$T_{:>[\alpha:\gamma/\delta]}(m)$	Transform the type of parameter α in method m from δ to γ, where γ is a supertype of δ.

Note that these definitions depend upon relationships between types, which are defined in Section 6.6, whose definitions in turn depend recursively upon method match definitions. This mutual dependency is characteristic of typing in object-oriented languages, and its correct treatment is of central importance to type safety in object-oriented systems. As noted previously, Zaremski and Wing do not address it.

Although arbitrary compositions of the transformations $T_{<:[\alpha:\gamma/\delta]}$ and $T_{:>[\alpha':\gamma'/\delta']}$ could represent matches in which parameters are arbitrary combinations of subtypes and supertypes, here we limit consideration to three particular composition patterns that are of primary importance in practice: covariance, contravariance, and subtyping. We begin with covariance.

[8] Possibly renamed, reordered, etc.

Definition 6.8 (PolySPIN Covariant Method Match)

Given method m with parameters $\alpha_1, \alpha_2, \ldots, \alpha_n$, let $T_{<:}$ denote the composition of transformations $T_{<:[\alpha_1:\gamma_1/\delta_1]} \circ \ldots \circ T_{<:[\alpha_n:\gamma_n/\delta_n]}$. We define $M_{covarX:Y}(m_X, m_Y) \equiv \exists T_{X:Y}$ and $T_{Y:X}$ and $T_{<:}$ such that $T_{X:Y}(m_X) =_{X:Y} T_{<:}(T_{Y:X}(m_Y))$.

As an example, consider the `SquareFeet` methods of Figures 6.1 and 6.3, the relevant parts of which[9] are:

```
// C++
class House {
  int SquareFeet();
};

; CLOS
(defclass Building () ())
(defmethod SquareFeet ((this Building)) ...)
(defclass House (Building) ())
```

Notice that the CLOS type `House` inherits the method `SquareFeet` from a supertype, `Building`.

Now consider a new type, `Architect`, representing a person who designs buildings (including houses), with a method `AcceptJob` permitting the architect to accept the project of designing a given building. This type might be represented in C++ and CLOS as follows:

```
// C++
class Architect : Person {
```

[9]Here, and later in the chapter, we omit the C++ `public` keyword for brevity. Assume in this chapter that all C++ methods are public.

```
    void AcceptJob(House);
  };

  ; CLOS
  (defclass Architect (Person) ...)
  (defmethod AcceptJob ((this Architect) (b Building)) ...)
```

The two `AcceptJob` methods would be related by PolySPIN Exact Method Match if both had a final parameter of type `House`, by an analysis much like that of the `SquareFeet` example illustrating Definition 6.6, assuming for the moment that the two `House` types are considered to match.[10] In this case, however, one method has the final parameter type `House` and the other has `Building`. Because the CLOS `House` type is a subtype of the CLOS `Building` type, the two `AcceptJob` methods are related by PolySPIN Covariant Method Match:

$$\begin{aligned} X &= \text{C++} \\ Y &= \text{CLOS} \\ T_{C++:CLOS} &= \text{same as in the example following Definition 6.6} \\ T_{CLOS:C++} &= \text{same as in the example following Definition 6.6} \end{aligned}$$

and transformation $T_{<:}$ is the composition of transformations:

$$\begin{aligned} T_{<:} &= T_{<:[1:\texttt{Architect/Architect}]} \circ T_{<:[2:\texttt{House/Building}]} \\ &= \mathit{Identity} \circ T_{<:[2:\texttt{House/Building}]} \end{aligned}$$

where *Identity* represents the identity transformation (i.e., do nothing). Thus we may conclude:

[10] Type matching will be covered in Section 6.6. The mutual dependency of method matching and type matching is evident here.

$$M_{covarC{+}{+}:CLOS}(m_{C{+}{+}}, m_{CLOS})$$

where:

m_{C++} = `void AcceptJob(House)`

m_{CLOS} = `(defmethod AcceptJob ((this Architect) (b Building)) ...)`

PolySPIN Covariant Method Match is asymmetric. It is type-safe only in one direction; in the previous example, a C++ parameter of type `House` can stand in for the associated CLOS parameter of type `Building`, but not vice-versa.

We model contravariant matching in an analogous manner.

Definition 6.9 (PolySPIN Contravariant Method Match)
Given method m with parameters $\alpha_1, \alpha_2, \ldots, \alpha_n$, let $T_{:>}$ denote the composition of transformations $T_{:>[\alpha_1:\gamma_1/\delta_1]} \circ \ldots \circ T_{:>[\alpha_n:\gamma_n/\delta_n]}$. We define $M_{contravarX:Y}(m_X, m_Y) \equiv \exists T_{X:Y}$ and $T_{Y:X}$ and $T_{:>}$ such that $T_{X:Y}(m_X) =_{X:Y} T_{:>}(T_{Y:X}(m_Y))$.

In our `Architect` example, the two `AcceptJob` methods are related also by PolySPIN Contravariant Method Match when m_X is the CLOS method and m_Y is the C++ method, the reverse of the covariant example, due to the inverse relationship between supertype and subtype. Thus this match is also asymmetric, and type-safe in only one direction.

Finally, we examine polylingual subtype match, in which the parameters of the two methods are in a contravariant relationship and the return types are in a covariant relationship:

Definition 6.10 (PolySPIN Subtype Method Match)
Given method m with parameters $\alpha_1, \alpha_2, \ldots, \alpha_{n+1}$, where α_{n+1} is the return parameter,

let T_{sub} *denote the composition of transformations* $T_{<:[\alpha_1:\gamma_1/\delta_1]} \circ \ldots \circ T_{<:[\alpha_n:\gamma_n/\delta_n]} \circ T_{:>[\alpha_{n+1}:\gamma_{n+1}/\delta_{n+1}]}$. *We define* $M_{subX:Y}(m_X, m_Y) \equiv \exists T_{X:Y}$ *and* $T_{Y:X}$ *and* T_{sub} *such that* $T_{X:Y}(m_X) =_{X:Y} T_{sub}(T_{Y:X}(m_Y))$.

For example, suppose each `Architect` type has a method `CurrentProject` that returns the project on which the architect is currently working:

```
House CurrentProject();
```

```
(defmethod CurrentProject ((this Architect))
  (declare (return-values Building)) ...)
```

We now construct transformation T_{sub} by defining transformations on the parameters and the return types of these methods. The self parameters of the two `CurrentProject` methods are of the two (presumed to be matching) `Architect` types, so the parameter transformation $T_{<:[1:Architect/Architect]}$ may be defined. `Building` is a supertype of `House` in CLOS, and the CLOS and C++ `House` types are presumed to match, so the transformation $T_{:>[2:\texttt{Building/House}]}$ may be defined. Thus,

$$T_{sub} \quad = \quad T_{<:[1:\texttt{Architect/Architect}]} \circ T_{:>[2:\texttt{Building/House}]}$$

indicating that the two `CurrentProject` methods are related by PolySPIN Subtype Method Match.

PolySPIN Subtype Method Match is asymmetric, which is not surprising as subtyping is asymmetric in general. ($A<:B$ implies $B \not<: A$, unless A and B happen to be equal.) The match is type-safe in one direction, permitting the subtype to stand in for the type but not vice-versa.

Each of the examples in this section used methods whose corresponding parameters had the same names and positions in the parameter lists; that is, they were

built upon PolySPIN Exact Method Match. Covariant, contravariant, and subtype method match can be made applicable to other situations in which parameter names or ordering differ between two methods, through the use of parameter renaming and reordering transformations, as given in Section 6.5.2.1.

6.6 Polylingual Type Matching

Now that we have defined matching between methods, we turn our attention to matching between types, using judgments and type properties in the style of Abadi and Cardelli [1]. The method-matching definitions of Section 6.5 will play an important role in the determination of compatibility between types.

Some of notation will be necessary, taken or adapted from Abadi and Cardelli:

m	A method.
$[m]$	Type containing a method.
m_X^i	The i^{th} method of a type written in language X.
$[m_X^{i\ i\in 1..n}]$	Type containing n methods $m_X^1, \ldots, m_X^n$.
$[m_X^i], i \in 1..n$	Type containing n methods $m_X^1, \ldots, m_X^n$.

The last two pieces of notation have identical meaning but are used in different contexts. The first is used when the index i appears only within the type, and the second is used when index i is also used elsewhere in a larger expression. Also recall $E \vdash_L B$ means that B is well-formed with respect to language L (Section 6.3.1).

6.6.1 Exact Type Matching

A first type property establishes the simplest form of type equivalence: that of objects with identical suites of n methods (same set of method names, equivalent parameter names and types, and equivalent return types). This is essentially the form of type equivalence used in Kaplan's PolySPIN V0.

(Exact Type Match)

$$\frac{E \vdash_X [m_X^i] \quad E \vdash_Y [m_Y^i] \quad E \vdash M(m_X^i, m_Y^i) \quad \forall i \in 1..n}{E \vdash [m_X^i] \; \mathcal{R} \; [m_Y^i]}$$

In other words: given a type in language X and another in language Y, and given that the i^{th} methods of each type match each other by some predicate M for all i from $1..n$, this type property concludes that the two types match according to a relationship $\mathcal{R}$. The choice of relationship $\mathcal{R}$ depends on the definition of M that is used. In this case, $\mathcal{R}$ is the type equivalence relationship if and only if M is an exact method match relationship.

For example, the following two types match exactly according to the above type property, when M is type equivalence and $\mathcal{R}$ is PolySPIN Exact Method Match.

```
class House {
  int SquareFeet();
};
```

```
(defclass House () ())
(defmethod SquareFeet ((this House))
  (declare (return-values Integer)) ...)
```

To see this, let language X be C++ and language Y be CLOS. The judgment $E \vdash_Y [m_X^i]$ indicates that the C++ `House` class is well-formed, and the judgment $E \vdash_Y [m_Y^i]$ indicates the same about the CLOS `House` class. We showed in the example following Definition 6.6 that the two `SquareFeet` methods match exactly according to PolySPIN Exact Method Match. Thus, all the required judgments are met for our type property, and we may conclude $E \vdash [m_X^i] \; \mathcal{R} \; [m_Y^i]$, which means that the two `House` types match according to Exact Type Match.

6.6.2 Relaxed Type Matching

Here we examine a few possible relaxations of the type matching rule. We focus on relaxations that are already implemented in PolySPINner, or that we foresee might be important; this is by no means an exhaustive list. Although a variety of relaxed matching techniques might seem convenient to developers of polylingual systems, and are possible to code as compatibility-checking and code-generating modules for PolySPINner, not all of them are suitable in practice for reasons of type safety.

Some relaxed matches are symmetric, meaning that each type may substitute for the other, and some are asymmetric. Subtyping is a prime example of an asymmetric relationship: any object of the subtype may be used as though it were a member of the supertype, but not vice-versa. This is in line with the normal rules of subsumption.

6.6.2.1 Renaming

The simplest relaxation is the renaming of a type. This is accomplished analogously to the renaming of a method, by defining a transformation $T_{[\alpha/\beta]}(t)$ to mean the replacement of name β with name α in type t.

The renaming of a type—that is, compatibility of two types of distinct languages when the types differ only in name—is type-safe, assuming that name equivalence is not required. The methods of the two types will still map one-to-one and onto.

6.6.2.2 Subset Matching

We specify that one type may have a superset of the methods of another type, and yet the two types may match. This asymmetric relationship is not considered by Zaremski and Wing.

(Subset Match)

$$\frac{E \vdash_X [m_X^{i\ i\in 1..n+m}] \qquad E \vdash_Y [m_Y^j] \quad E \vdash M(m_X^j, m_Y^j) \quad \forall j \in 1..n}{E \vdash [m_Y^{j\ j\in 1..n}]\ \mathcal{R}\ [m_X^{i\ i\in 1..n+m}]}$$

Specifically, type $[m_X^i]$ has $n+m$ methods, type $[m_Y^j]$ has only n methods, and because the two types have n methods that match, the latter type conforms to the former. For example, the following types have a subset match:

```
class House {
  int SquareFeet();
  int Bedrooms();
};
```

```
(defclass House () ())
(defmethod SquareFeet ((this House))
  (declare (return-values Integer)) ...)
```

The types have matching `SquareFeet` methods only, so the CLOS `House` type conforms to the C++ `House` type.

Subset match is asymmetric ($A \subset B$ implies $B \not\subset A$) and type-safe in one direction. In the previous example, an instance of the CLOS `House` type can safely stand in for an instance of the C++ `House` type because all of the CLOS methods have analogs in C++, but not vice-versa. This assumes (recursively) that the associated method mappings are type-safe.

The Subset Match type property may be used as a basis for other type properties depending on the values of $\mathcal{R}$ and M. For instance, if M is taken to be the subtype method match relationship, then $\mathcal{R}$ would be the standard subtype relationship, <:.

Some choices of M, however, may yield type matching rules that seem convenient but violate type safety. Casting the rules in our formal framework makes it possible to compare them with known results from object-oriented type theory [1] and hence to assess their type safety and other properties.

6.6.2.3 Intersection Matching

Sometimes two classes may have an intersecting set of methods but no subset relation. Consider the following C++ class:

```
class House {
  int SquareFeet();
  int Bedrooms();
};
```

and the related CLOS class:

```
(defclass House ...)
(defmethod Bedrooms ((this House))
   (declare (return-values Integer)) ...)
(defmethod AnnualHeatingCost ((this House))
   (declare (return-values Dollars)) ...)
```

If an application program is interested only in the number of bedrooms found in a house, then it ought to be possible to consider a match between these types because both have a `Bedrooms` method. The following type property represents a relationship between two types with a nonempty intersection of methods:

(Intersection Match)

$$\frac{\begin{array}{c} E \vdash_X [m_X^{i\ i\in 1..m}] \\ E \vdash_Y [m_Y^{i\ i\in 1..n}] \\ E \vdash M(m_X^i, m_Y^i)^{i\in 1..k} \quad 1 \leq k \leq min(n,m) \end{array}}{\begin{array}{c} E \vdash_X [m_X^{i\ i\in 1..k}] \\ E \vdash_Y [m_Y^{i\ i\in 1..k}] \\ E \vdash [m_X^{i\ i\in 1..k}]\ \mathcal{R}\ [m_Y^{i\ i\in 1..k}] \end{array}}$$

Here we are given two types—one in language X with m methods, and one in language Y with n methods—having k methods that match. The type property specifies that we can create synthetic base classes in each of the languages, containing the k matching methods, that are related to one another by relationship $\mathcal{R}$. In our example, the synthetic base classes contain only the `Bedrooms` methods, which the types have in common:

```
class HouseBase {
  int Bedrooms();
};
```

```
(defclass HouseBase ...)
(defmethod Bedrooms ((this House))
   (declare (return-values Integer)) ...)
```

Although the synthetic classes are called "HouseBase" for reference here, they are kept invisible from the interoperability engineer by PolySPIN. They are also abstract,

in that one cannot create an instance of such a class directly.[11] Thus, formalization helps us to understand the implications for type safety of a potentially convenient type matching rule and to determine what measures (e.g., introduction of synthetic base classes) we might need to take as a result.

Intersection match is a practical concept, since similar types are likely to have at least some similar methods, and it provides a type relationship absent from traditional object-oriented languages. Nevertheless, it is not type-safe in either direction.[12] Neither type conforms to the other because each has methods that the other does not, and if any of these methods are called in a polylingual system, a type error can occur. PolySPIN's Missing Method Manager (Section 7.4.1) was invented to prevent type-unsafe calls from occurring, allowing such types to be considered conformant or compatible.

6.6.2.4 Matching with Inheritance

Abadi and Cardelli identify two views of inheritance: embedding, in which inherited methods are viewed as being inside the inheritor, and delegation, in which inherited methods are viewed as external to the inheritor [1]. PolySPIN subscribes to the embedding view, as it simplifies type comparison by "flattening" the inheritance hierarchy, so that a type contains all its applicable methods. If all methods are viewed as being "inside" the types that define or inherit them, the modeling of method comparison is localized to the two types involved. In PolySPINner, the Parser components are responsible for flattening inheritance in each modified class.

---

[11] In our implementation, PolySPINner creates these synthetic base classes implicitly: the original type interfaces are used, but the Missing Method Manager (Chapter 7) guards against the execution of all methods outside the intersection.

[12] It is not clear whether to call intersection match "symmetric" or not, because *both* directions are type *un*safe.

```
// C++
class HouseC {
public:
  int SquareFeet();
  int Bedrooms();
  void InstallAlarm(Alarm, Location);
};

;; CLOS
(defclass HouseCLOS (Building) ())
(defmethod Protect ((self HouseCLOS) (loc Location) (a Alarm)) ...)

(defclass Building () ())
(defmethod Area ((self Building))(declare(return-values Integer))...)
```

Figure 6.5. Example types.

6.7 A Larger Example

We now demonstrate the application of relaxed matching theory to a more substantial example, shown in Figure 6.5. Types `HouseC` and `HouseCLOS` are intended to represent similar concepts (houses), and we wish to match the types for use in a polylingual system. Suppose the methods `SquareFeet` and `Area` both return the size of their respective buildings in square feet; and the methods `InstallAlarm` and `Protect` install an alarm system, an instance of type `Alarm`, into the building at a specified location, an instance of the type `Location`. Assume that the C++ and CLOS `Alarm` types have already been matched under PolySPIN, as have the C++ and CLOS `Location` types. This is to keep the size of the example manageable while still illustrating the pertinent formalisms.

The key points of this example are:

- The two class names are different: `HouseC` vs. `HouseCLOS`.
- The methods that we wish to correspond to one another have different names: `SquareFeet` vs. `Area`, and `InstallAlarm` vs. `Protect`.

- The C++ type `HouseC` has an additional method, `Bedrooms`, that `HouseCLOS` does not.
- The C++ method `InstallAlarm` has its formal parameters in an order different from those of the corresponding CLOS method, `Protect`.
- The CLOS type `HouseCLOS` inherits one of its methods, `Protect`, whereas the corresponding C++ method, `InstallAlarm`, is not inherited.

We now undertake the application of PolySPIN V0, V1, and V2 to the example, in turn.

6.7.1 Application of PolySPIN V0

The original PolySPIN V0, created prior to this dissertation, supports only the strictest kind of matching: type equality, modulo differences in language syntax. Thus, none of the key points listed in Section 6.7 are recognized by PolySPIN V0, which determines only that types `HouseC` and `HouseCLOS` are not compatible. (In fact, PolySPINner V0 simply examines the two type names, finds them different, and fails.)

6.7.2 Application of PolySPIN V1

PolySPIN V1, the subject of Chapter 5, supports various kinds of relaxed match, permitting the types `HouseC` and `HouseCLOS` to be considered conformant or compatible, and hence made interoperable by PolySPINner. We now apply PolySPIN V1 to the example and demonstrate how to:

1. Match the methods `InstallAlarm` and `Protect` by converting them to a canonical form and using renaming and permutation match.
2. Match the complete types using renaming and intersection match.

The methods `SquareFeet` and `Area` cannot be matched because PolySPIN V1 does not model inherited methods. Inheritance match will be demonstrated when we apply PolySPIN V2.

6.7.2.1 Methods `InstallAlarm` and `Protect`

The C++ method `InstallAlarm` and the CLOS method `Protect` are to be matched. For shorthand, we will call them m_I (for `InstallAlarm`) and m_P (for `Protect`).

In the example following Definition 6.6 in Section 6.5.1, we defined two language-pair-specific transformations, $T_{C++:CLOS}$ and $T_{CLOS:C++}$, that model PolySPIN's language mappings from C++ to CLOS and vice versa, respectively. These transformations will be used again here to transform the methods `InstallAlarm` and `Protect` into a canonical form in which they can be compared more easily. We begin by transforming `InstallAlarm` via $T_{C++:CLOS}$ by the following steps:

1. `void InstallAlarm (Alarm, Location)`

2. `void InstallAlarm (HouseC this, Alarm x1, Location x2)`

3. `void InstallAlarm (this HouseC, x1 Alarm, x2 Location)`

4. `void InstallAlarm ((this HouseC) (x1 Alarm) (x2 Location))`

5. `InstallAlarm ((this HouseC) (x1 Alarm) (x2 Location))`
   `(declare (return-values void))`

Step 5 is the transformed method $T_{C++:CLOS}(m_I)$ in canonical form. Likewise, we define $T_{CLOS:C++}$ to transform the CLOS declaration of `Protect`, stripping away the word `defmethod`, the outermost parentheses, and the method body except for the return type declaration, just as in Section 6.5.1. Thus, $T_{CLOS:C++}(m_P)$, transforming `Protect`, is:

```
Protect ((this HouseC) (loc Location) (a Alarm))
  (declare (return-values nil))
```

Unlike the example of Section 6.5.1, however, we cannot yet declare the two methods to be matched, because $T_{C{+}{+}:CLOS}(m_I) \neq T_{CLOS:C{+}{+}}(m_P)$. The method names are different, and the method parameters are in different orders. The next step is to reconcile the difference between their names, using the transformation for method renaming, $T_{[\alpha/\beta]}(m)$, in Table 6.2. The transformation

$$T_{[\texttt{InstallAlarm/Protect}]}(m) \tag{6.7}$$

represents the replacement of "Protect" by "InstallAlarm" in method m. Applying this to the already-transformed `Protect` method,

```
Protect ((this HouseC) (loc Location) (a Alarm))
  (declare (return-values nil))
```

yields:

```
InstallAlarm ((this HouseC) (loc Location) (a Alarm))
  (declare (return-values nil))
```

which may be written as $T_{[\texttt{InstallAlarm/Protect}]}(T_{CLOS:C{+}{+}}(m_P))$.

Next, we reconcile the differing parameter orders. Table 6.2 and Definition 6.7 define transformation $T_\pi(m)$ to permute the parameters of method m via permutation π. Let

$$\pi = (1, 3, 2) \tag{6.8}$$

be the permutation that swaps the second and third parameters of a method. Thus, applying $T_{(1,3,2)}$ to the method

```
InstallAlarm ((this HouseC) (loc Location) (a Alarm))
  (declare (return-values nil))
```

yields:

```
InstallAlarm ((this HouseC) (a Alarm) (loc Location))
  (declare (return-values nil))
```

which may be written as $T_{(1,3,2)}(T_{[\texttt{InstallAlarm/Protect}]}(T_{CLOS:C++}(m_P)))$. This is nearly identical to $T_{C++:CLOS}(m_I)$ except for the primitive types `nil` and `void`, which are the differing return types of the two methods. We reconcile this final difference using the language-pair-specific definition of equality, $=_{C++:CLOS}$, defined in in Section 6.5.1, that encapsulates a mapping of the primitive types in the two languages. Thus, we have:

$$T_{C++:CLOS}(m_I) \quad =_{C++:CLOS} \quad T_{(1,3,2)}(T_{[\texttt{InstallAlarm/Protect}]}(T_{CLOS:C++}(m_P))) \tag{6.9}$$

As a shorthand, we define a method matching relationship, $M^\star$, to represent this matching:

$$M^\star(m_I, m_P) \tag{6.10}$$

Thus, our example has shown that the C++ `InstallAlarm` method and CLOS `Protect` method match with respect to $M^\star$.

6.7.2.2 Type Renaming

Now that the methods `InstallAlarm` and `Protect` have been shown to match, we turn our attention to matching the two types, `HouseC` and `HouseCLOS`. First, we address the fact that the two types have different names. This is modeled by

$T_{[\alpha/\beta]}(t)$, the type-renaming transformation given in Section 6.6.2.1, that models the substitution of one name for another. The transformation

$$T_{[\texttt{HouseC/HouseCLOS}]} \tag{6.11}$$

replaces "HouseCLOS" by "HouseC" and will be applied to the type `HouseCLOS`.

6.7.2.3 Intersection Match

PolySPIN V1 is not able to recognize the inheritance relationship between the methods `SquareFeet` and `Area`, so the best match that PolySPIN V1 can manage between the types is an intersection match, since they have one method in common. As explained in Section 6.6.2.3, an intersection match is modeled in Abadi and Cardelli's notation as:

$$\text{(Intersection Match)}\quad \frac{\begin{array}{c} E \vdash_X [m_X^{i\ i\in 1..m}] \\ E \vdash_Y [m_Y^{i\ i\in 1..n}] \\ E \vdash M(m_X^i, m_Y^i)^{i\in 1..k} \quad k \leq min(n,m) \end{array}}{\begin{array}{c} E \vdash_X [m_X^{i\ i\in 1..k}] \\ E \vdash_Y [m_Y^{i\ i\in 1..k}] \\ E \vdash [m_X^{i\ i\in 1..k}]\ \mathcal{R}\ [m_Y^{i\ i\in 1..k}] \end{array}}$$

Let language X be C++ and language Y be CLOS. We begin with judgments describing the two types. Just as we defined m_I and m_P as a shorthand for the methods `InstallAlarm` and `Protect`, define the additional shorthand:

$$m_S \quad \equiv \quad \texttt{int SquareFeet()}$$

$$m_B \equiv \texttt{int Bedrooms}()$$

$$m_A \equiv (\texttt{defmethod Area } ((\texttt{self Building})) \ ...)$$

Type `HouseC` is notated as:

$$E \vdash_{C++} [m_S, m_B, m_I] \tag{6.12}$$

corresponding to the judgment $E \vdash_X [{m_X^i}^{i \in 1..m}]$ in the definition of Intersection Match. Likewise, type `HouseCLOS` is notated as:

$$E \vdash_{CLOS} [m_P] \tag{6.13}$$

corresponding to the judgment $E \vdash_Y [{m_Y^i}^{i \in 1..n}]$. We know from Equation 6.10 that methods m_I and m_P match by $M^\star(m_I, m_P)$. This corresponds to the judgment $E \vdash M(m_X^i, m_Y^i)^{i \in 1..k}$. Thus, all premise judgments are satisfied in the definition of Intersection Match, so we may instantiate the three conclusion judgments:

$$E \vdash_X [{m_X^i}^{i \in 1..k}] \equiv E \vdash_{C++} [m_I] \tag{6.14}$$

$$E \vdash_Y [{m_Y^i}^{i \in 1..k}] \equiv E \vdash_{CLOS} [m_P] \tag{6.15}$$

$$E \vdash [{m_X^i}^{i \in 1..k}] \ \mathcal{R} \ [{m_Y^i}^{i \in 1..k}] \equiv E \vdash [m_I] \ \mathcal{R} \ [m_B] \tag{6.16}$$

where $\mathcal{R}$ is the Intersection Match relationship. Note that the types $[m_I]$ and $[m_P]$ given in judgments 6.14 and 6.15 are PolySPIN-generated, synthetic base types as explained in Section 6.6.2.3. Thus, the full, relaxed matching of the types `HouseC` and `HouseCLOS` under PolySPIN V1 is written:

(PolySPIN V1 Example Intersection Match)

$$\frac{\begin{array}{c} E \vdash_{C\text{++}} [m_S, m_B, m_I] \\ E \vdash_{CLOS} T_{[\texttt{HouseC/HouseCLOS}]}([m_P]) \\ E \vdash M^{\star}(m_I, m_P) \end{array}}{\begin{array}{c} E \vdash_{C\text{++}} [m_I] \\ E \vdash_{CLOS} [m_P] \\ E \vdash [m_I] \ \mathcal{R} \ [m_B] \end{array}}$$

As a result of this analysis, PolySPINner V1 is capable of matching the types `HouseC` and `HouseCLOS` so that they are polylingually interoperable with respect to the methods `InstallAlarm` and `Protect`. This was previously not possible with PolySPINner V0. In addition, intersection match is not possible in traditional object-oriented languages that support only subtyping as a form of relaxed match. PolySPIN's additional flexibility is counterbalanced by a potential loss of type safety, however. The unmatched methods outside the intersection will need to be handled by PolySPINner's Missing Method Manager, discussed in Chapter 7, in order for this match to be type-safe.

6.7.3 Application of PolySPIN V2

Although PolySPIN V1 could not model the inheritance relationship between the C++ method `SquareFeet` and the inherited CLOS method `Area`, PolySPIN V2 is designed to do so. Applying PolySPIN V2 to the example will demonstrate how to:

1. Match the methods `InstallAlarm` and `Protect` using renaming and permutation match. This step is identical to Section 6.7.2.1 and will not be repeated.

```
// C++
class HouseC {
public:
  int SquareFeet();
  int Bedrooms();
  void InstallAlarm(Alarm, Location);
};

;; CLOS
(defclass HouseCLOS () ())
(defmethod Protect ((self HouseCLOS) (a Alarm)) ...)
(defmethod Area ((self HouseCLOS)) (declare (return-values Integer))
   ...)

(defclass Building () ())
(defmethod Area ((self Building))(declare(return-values Integer))...)
```

Figure 6.6. Example types after PolySPIN flattening.

2. Match the methods `SquareFeet` and `Area` using inheritance and renaming match.

3. Match the types using renaming and subset match, rather than intersection match.

6.7.3.1 Methods `SquareFeet` and `Area`

As described in Section 6.6.2.4, PolySPIN V2 flattens the inheritance hierarchy, so the types actually compared are those in Figure 6.6. Notice that the type of the self parameter of `Area` becomes `HouseCLOS`, rather than `Building`, when applied to a `HouseCLOS` object, modeling `Area` as a virtual method. PolySPIN V2 treats all inherited methods as virtual when comparing types.

Recall that we abbreviate method `SquareFeet` as m_S and method `Area` as m_A. As with methods `InstallAlarm` and `Protect`, we first apply the transformations $T_{C++:CLOS}$ and $T_{CLOS:C++}$ to convert methods m_S and m_A, respectively, to a canonical form in which they can be compared. The results are:

$$
\begin{aligned}
T_{C\texttt{++}:CLOS}(m_S) &= \texttt{SquareFeet ((this HouseC))} \\
&\quad\ \texttt{(declare (return-values int))} \\
T_{CLOS:C\texttt{++}}(m_A) &= \texttt{Area ((this HouseCLOS))} \\
&\quad\ \texttt{(declare (return-values Integer))}
\end{aligned}
$$

Again, these methods have different names, so a renaming transformation of the form $T_{[\alpha/\beta]}(m)$:

$$T_{[\texttt{SquareFeet/Area}]}(m) \tag{6.17}$$

may be used to model the replacement of "Area" by "SquareFeet" in method m. Applying this to the method $T_{CLOS:C\texttt{++}}(m_A)$ yields:

```
SquareFeet ((this HouseC)) (declare (return-values Integer))
```

This transformed method $T_{[\texttt{SquareFeet/Area}]}(T_{CLOS:C\texttt{++}}(m_A))$ matches the canonical form of method m_S, except for the difference between the primitive return types `int` and `Integer`. So once again, we use our language-pair-specific definition of equality:

$$T_{C\texttt{++}:CLOS}(m_S) =_{C\texttt{++}:CLOS} T_{[\texttt{SquareFeet/Area}]}(T_{CLOS:C\texttt{++}}(m_A)) \tag{6.18}$$

to illustrate the relaxed matching of methods `SquareFeet` and `Area`. Just as we denoted a similar matching of methods `InstallAlarm` and `Protect` as $M^\star$, we denote this matching as M°:

$$M^\circ(m_S, m_A) \tag{6.19}$$

6.7.3.2 Subset Match

Recall the subset match, defined in Section 6.6.2.2, which is denoted as:

$$\text{(Subset Match)}\qquad \frac{E \vdash_X [m_X^{i\ i\in 1..n+m}] \qquad E \vdash_Y [m_Y^j] \quad E \vdash M(m_X^j, m_Y^j) \quad \forall j \in 1..n}{E \vdash [m_X^{i\ i\in 1..n+m}]\ \mathcal{R}\ [m_Y^{j\ j\in 1..n}]}$$

As in the PolySPIN V1 example, we take language X to be C++ and language Y to be CLOS. Type `HouseC` is again denoted as:

$$E \vdash_{C++} [m_S, m_B, m_I] \tag{6.20}$$

corresponding to the judgment $E \vdash_X [m_X^{i\ i\in 1..n+m}]$ in the definition of Subset Match. Type `HouseCLOS`, however, now has an additional method, m_A, since it inherits the `Area` method, and is represented as:

$$E \vdash_{CLOS} [m_P, m_A] \tag{6.21}$$

corresponding to the judgment $E \vdash_Y [m_Y^j]$. We have seen that methods `InstallAlarm` and `Protect` match according to $M^\star(m_I, m_P)$, and that methods `SquareFeet` and `Area` match with respect to $M^\circ(m_S, m_A)$. Thus, we can define a relaxed match predicate to encapsulate both:

$$M(m_1, m_2) = \begin{cases} M^\star(m_1, m_2) & \text{if } m_1 = m_I \\ M^\circ(m_1, m_2) & \text{if } m_1 = m_S \end{cases} \tag{6.22}$$

to fit the premise judgment $E \vdash M(m_X^j, m_Y^j)$.[13] Having satisfied all premise judgments of the Subset Match type property, we may write the conclusion judgment:

[13] Having established that such an M exists, we will use the individual relationships $M^\star$ and M° instead, for clarity.

$$E \vdash [m_S, m_B, m_I] \; \mathcal{R} \; [m_P, m_A] \tag{6.23}$$

where $\mathcal{R}$ is the subset match relationship. Thus, the full, relaxed matching between types `HouseC` and `HouseCLOS` under PolySPIN V2 is:

$$\frac{\begin{array}{c}\text{(PolySPIN V2 Example Subset Match)} \\ E \vdash_{C++} [m_S, m_B, m_I] \\ E \vdash_{CLOS} T_{[\texttt{HouseC/HouseCLOS}]}([m_P, m_A]) \\ E \vdash M^{\star}(m_I, m_P) \qquad E \vdash M^{\circ}(m_S, m_A)\end{array}}{E \vdash [m_S, m_B, m_I] \; \mathcal{R} \; [m_P, m_A]}$$

As a result of this analysis, PolySPINner V2 is capable of matching the types `HouseC` and `HouseCLOS` so that they are interoperable by subset match, a polylingual analog of subtyping in a monolingual system.[14] This was previously not possible with PolySPINner V1 due to the inheritance relationship. This match is asymmetric, in that an instance of `HouseC` may be considered, polylingually, as an instance of `HouseCLOS`, but not vice versa. In order to make the match type safe, PolySPINner's Missing Method Manager, discussed in Chapter 7, must prevent the additional method `Bedrooms` from ever being invoked polylingually on an instance of `HouseCLOS`.

6.8 Implementation

PolySPINner V2 detects inheritance relationships, and its Matcher includes inherited methods when analyzing types for matching methods. This is accomplished by traversing the type hierarchy. If a method is found in type t_1 but not in type t_2, the supertypes of t_2 are checked[15] for a match for that method.

[14] A subset relationship among the method sets of two types is not always a subtype relationship, due to issues concerning self types and binary methods; see [1, 16] for details.

[15] Supertypes are checked in whatever order is appropriate for the given programming language.

More complex matches are supported as well. Guided by the formalisms in the previous sections, PolySPINner V2 has been provided with a more sophisticated matching engine and a library of useful, ready-made matching criteria. These include exact match and various combinations of permuted parameters and methods, subset and superset relationships among parameters and methods, and intersection match of parameters and methods, all while detecting subtype relationships and inheritance of methods. It may be possible to match two types in multiple ways, and it is the responsibility of the interoperability engineer to select (or create) appropriate matching criteria that produce an unambiguous match, with or without user guidance.

C++ and CLOS have very different inheritance mechanisms, so PolySPINner lets inheritance relationships be computed by each language's run-time system rather than attempting to precompute them itself. PolySPINner need not, for example, compute the most applicable CLOS method when matching C++ and CLOS methods. Instead, PolySPINner matches C++ methods with CLOS generic functions. When an interlanguage call is made from C++ to CLOS, the CLOS run-time system computes the most applicable method as usual, transparently to PolySPINner. Likewise, PolySPINner need not track CLOS's `:before`, `:after` and `:around` specifiers [26], as the CLOS run-time system takes care of invoking the proper methods. PolySPINner will be covered in much greater detail in Chapter 9.

6.9 Summary

This chapter has presented a formalism for interlanguage matching of types in multilanguage settings. Building on formalisms for monolingual type matching and object-oriented type theory, we have developed a basis for reasoning about and comparing types written in different programming languages, using both strict and relaxed matching. Previously, such notions have been informally specified or left implicit in multilanguage settings. Although designed for application to polylingual systems, it

may have wider applicability for multilanguage interoperability. The PolySPINner toolset implements significant parts of this formalism in support of polylingual interoperability, currently supporting C++ and CLOS. The next chapter delves more deeply into specific issues of type safety in polylingual systems.

CHAPTER 7

TYPE SAFETY IN POLYLINGUAL SYSTEMS

7.1 Overview

In Chapter 6, we took a formal approach to the matching of methods and types in polylingual systems. But what happens to methods that do not get matched? For instance, suppose the methods of some type t are a subset of those of type t'. Thus the two types are related by Subset Match, as in Section 6.6.2.2, and type t conforms to t'. This cannot be the end of the story, because an application program might try to invoke a method of t' on an instance of t, when t does not contain that method. This would be a type error.

Defining relaxed relationships between types is not enough: PolySPIN must also ensure that type errors cannot occur. That is, once a pair of types is matched, or declared "compatible," all method invocations on an instance of either type must be type-safe.

In this chapter, we present a functional definition of type compatibility that is more general than that of Chapter 6 and does not assume a multilanguage setting. There is little difference between the type errors that result from a monolingual or polylingual method call; in either case, an invocation of a nonexistent method is attempted. We then introduce the *message-not-understood problem*, which states that unmatched methods can cause type errors. Finally, we explain how PolySPIN uses the functional definition of type compatibility as the basis for its *Missing Method Manager* to prevent the message-not-understood problem.

7.2 Notions of Compatibility: A Taxonomy

In software, there are many notions of "compatibility" between types, defined at various levels of abstraction [74]. On the more primitive or concrete end of the spectrum, two types may be "compatible" with respect to the binary representations of their instances. For example, two arrays could store the same values in the same order and therefore be considered to hold "the same data," but one array might be packed while the other is not. On the more abstract end of the spectrum, two types may be "compatible" if, in some sense, they represent "the same thing." For example, two types could both represent the concept "automobile," even if they have entirely different implementations and method suites. Somewhere in between these endpoints is the notion that two types are "compatible" if they have the same methods, or if one is a subtype of the other, or if some other relaxed match applies between them (Chapter 6). All of these notions of "compatibility" are in service of the same goal: to allow instances of different types to be manipulated, transparently, as if they were of the same type.

In an attempt to classify diverse kinds of type compatibility, we identify several levels of abstraction:

- *Representation level*: As in [74], compatibility with respect to the bitwise representation of type instances.

- *Declaration level*: Compatibility with respect to the source code declaration of types. This is similar to specification level compatibility [74] but specifically identifies source code declarations as the means for determining compatibility.

- *Behavioral level*: Compatibility with respect to the semantics of a type. Examples are pre- and post-conditions on types, behavioral subtyping [49], and specification matching [76].

If we consider "level of abstraction" to be one dimension of type compatibility, another is the strictness of compatibility: how similar two types must be in order to be considered compatible:

- *Exact*: Literally, no differences with respect to the given level of abstraction.
- *Structural*: Identical structure, though the names of component parts may differ.
- *Relaxed*: Structural differences are permitted.

Figure 7.1 illustrates pairings of abstraction and strictness, including representative examples from the literature. Of particular note is relaxed, declaration-level compatibility, which has several important subcategories:

- *Formal mappings*: In two types, compatibility between methods may be described by a mapping of formal parameters of one method to those of another. Examples are Z/W matching (e.g., generalized, specialized, reordering, currying), subtyping, and Bruce's matching [16].
- *Actual mappings*: In two types, compatibility between methods cannot be described simply by mappings between formal parameters of methods. Actual parameters are required in the mapping. An example is Konstantas's H-TMSL [43].

Actual mappings are more general because they can simulate formal mappings by using a universal quantifier over all possible actual parameters. That is, reasoning about "all objects of type t" is equivalent to reasoning about "type t."

The present study of polylingual interoperability is most concerned with declaration level interoperability. Relaxed representation level interoperability has been addressed by others [4], as has relaxed behavioral level type compatibility [76] in a monolingual context.

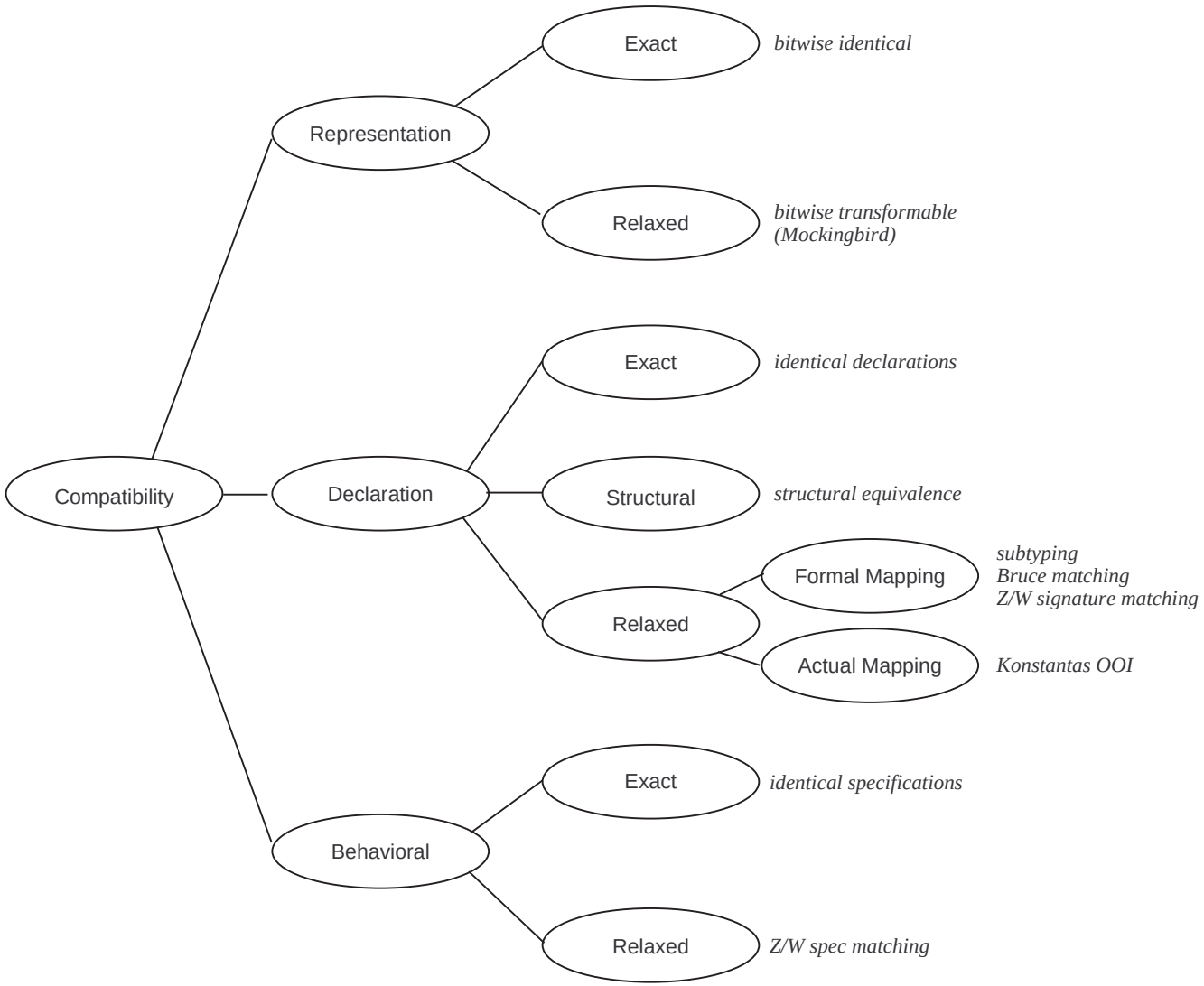


Figure 7.1. Classification of diverse notions of type compatibility.

7.3 A Functional Definition of Compatibility

As in Chapter 6, we shall define conformance and compatibility as relationships between two types, t and t'. This time, however, they will be expressed in terms of a function, κ, called a compatibility criterion. Intuitively, κ is a table that specifies which methods/parameters of type t are mapped to which methods/parameters of type t' (and possibly vice-versa). In other words, if (say) an application attempts to invoke a method of t on an instance of t', criterion κ specifies what should happen, e.g., which method of t' should be (automatically) invoked instead, and with what parameter values. Types t and t' are then said to be conformant or compatible "with respect to criterion κ" if certain properties hold, discussed in the next section.

Conformance and compatibility will be defined with respect to method invocations with actual parameters, rather than method declarations with formal parameters, since the former is more general, as argued briefly in Section 7.2. In order to construct an appropriate call to a "target" method in t', we may need to specify the actual parameter values of the target method, or at least a means for computing those values. For example, suppose type t has a method m with two parameters, and type t' has a method m' with three parameters. We want these methods to be matched so t and t' may be considered compatible, in that a call to $t'.m$ will be converted transparently into a call to $t'.m'$. In this case, the two parameters of m must be mapped somehow to the three parameters of m'. Specifically, some means for computing the *values* of the three parameters of m' (presumably as a function of the two parameters of m) must be specified. We would like to be able to specify a complex mapping of parameters such as:

> If $t.m$ is invoked, then $t'.m'$ should be called with its three parameters: parameter 1 is NULL, parameter 2 is the sum of the two parameters of $t.m$, and parameter 3 is t' itself.

This mapping discusses the actual parameters of method m' and thus cannot be expressed using formal parameters alone, using Zaremski/Wing matching for example. The following definitions will allow one to define compatibility criteria of this kind, expressed as functions.

7.3.1 Definitions

First we define what it means for a method to be applicable to a type.

Definition 7.1 (Applicable Method) *A method m is* **applicable** *to a type t if method m is defined in type t or inherited by type t.*

Definition 7.2 (Method Invocation) *A* **method invocation** *is a tuple, (t, m, p), where t is a type, m is some method applicable to t, and $p = p_0, p_1, p_2, \ldots, p_n$ is a sequence of actual parameters that could be passed to method m. $p_1, p_2, \ldots, p_n$ are the actual parameters corresponding to the signature of m, and p_0 represents the value of the return type.*

If a method is applicable to a type, so is an invocation of that method.

Definition 7.3 (Applicable Method Invocation) *If method m is applicable to type t, then we say also that any method invocation (t, m, p) is* **applicable** *to type t if the following is true:*

- *Let $\tau_1, \tau_2, \ldots, \tau_n$ be the types of the formal parameters of method m, in order.*
- *Let $p_1, p_2, \ldots, p_n$ be the actual parameters of parameter list p, in order.*
- *Parameter p_i is of type τ_i, $\forall i \in 1..n$.*

Now we define compatibility criteria which will form the basis for our definition of compatibility.

Definition 7.4 (Compatibility Criterion) *Let T be the set of all types, let M be the set of all methods, and let E be the set of all exceptions.*

A **compatibility criterion** *is a function, κ, of the form:*

$$\kappa : (T \times M \times T^*) \rightarrow ((T \times M \times T^*) \cup E \cup \{\emptyset\})$$

The domain of κ represents all possible method invocations on instances of a type. The range represents all possible actions that may occur in response to the method invocation given as input: either a method invocation on an instance of some type, or a raised exception, or no action.

Definition 7.5 (Conforms To) *A type t'* **conforms to** *a type t, with respect to some compatibility criterion κ, if for every method m applicable to t, and every sequence p of actual parameters that may be passed to method m, the method invocation $\kappa(t', m, p)$ is applicable to type t'.*

In other words, for every method invocation applicable to type t, there is an analogous method invocation applicable to type t'. This definition is a restatement of Definition 6.1 in terms of method invocations and a compatibility criterion. If Definition 7.5 holds, then type t' can be used in place of t in any context in which an instance of type t is valid, implying Definition 6.1. Likewise, if Definition 6.1 holds, then any method of t invoked on an instance of t' is applicable, implying Definition 7.5.

Note that this relationship is not symmetric: type t is not necessarily compatible with t'. This asymmetry makes it possible to define asymmetric kinds of compatibility such as subtyping. For symmetric relationships, we have an analog to Definition 6.3:

Definition 7.6 (Compatible) *Two types t and t' are* **compatible***, with respect to compatibility criterion κ, if t conforms to t' with respect to κ, and t' conforms to t with respect to κ.*

Note that we use a single function κ for conformance in both directions (i.e., t conforms to t', and t' conforms to t), rather than two compatibility functions, one for each direction. This is merely to simplify the notation. Mathematically, it does not matter whether we use one function or two. If, instead, we said that t conforms to t' with respect to compatibility criterion κ_1, and that t' conforms to t with respect to another criterion κ_2, we could define a single criterion κ as:

$$\kappa(\hat{t}, m, p) = \begin{cases} \kappa_1(\hat{t}, m, p) & \text{if } \hat{t} = t \\ \kappa_2(\hat{t}, m, p) & \text{if } \hat{t} = t' \end{cases}$$

Finally, we define type safety.

Definition 7.7 (Type-Safe) *A method invocation* (t, m, p) *is* **type-safe** *if it is applicable to type* t.

7.3.2 Examples

Let t and t' be types we want to consider compatible.

7.3.2.1 Type Equality

The strictest compatibility between two types t and t' is for both to have exactly the same methods. We specify this in two parts. First, for each method m applicable to type t, and each sequence of actual parameters p of method m, we specify that:

$$\kappa(t', m, p) = (t', m, p)^1$$

In other words, every method m applicable to type t, when invoked on an instance of t', is applicable to t'. Second, every method applicable to type t' must also be

[1] It is tempting—but incorrect—to believe that the first t' in $\kappa(t', m, p)$ should be t. Take careful note of the definition of κ. The term $\kappa(t', m, p)$ is the action to be taken if method m (which we said belongs to type t) is invoked on an instance of t'.

applicable to type t. For all each method m' applicable to t', and each sequence of actual parameters p' of method m',

$$\kappa(t, m', p') = (t, m', p')$$

Thus, both types have exactly the same methods, so the two types are equal.

7.3.2.2 Structural Equivalence

If two types t and t' are structurally equivalent, there exists a bijection between the methods of t and the methods of t' such that each method in t has the same signature and return type as its corresponding method in t'. The method names may differ, however. This is represented as:

$$\kappa(t', m, p) = (t', m', p)$$

illustrating that the method may change but the parameter list does not.

7.3.2.3 Subtyping

Type t' is behaviorally a subtype of t if all methods applicable to t are also applicable to t'. For all methods m applicable to t, and sequences of actual parameters p of method m:

$$\kappa(t', m, p) = (t', m, p)$$

Note that the Type Equality example is precisely this relationship but specified in both directions: that is, in Type Equality, types t and t' are subtypes of each other, and therefore the same type.

7.3.2.4 Zaremski and Wing Signature Match

Zaremski and Wing's relaxed matches are all of the form:

$$\kappa(t, m, p) = (t', m, p')$$

Not all compatibility criteria of this form can be represented as Z/W matches, however. For instance, Z/W matching can represent:

$$\kappa(t, m, p) = (t', m, \pi(p)) \qquad \text{Permutation of parameters}$$
$$\kappa(t, m, p) = (t', m, uncurry(p)) \quad \text{Uncurry parameters}$$

but not a more complex match involving actual parameters, such as:

$$\kappa(t, m, (p_1, p_2)) = (t', m, (p_1 + p_2, 37))$$

nor any other function in which actual parameters must be considered when matching two methods.

7.3.2.5 CORBA Dynamic Types

Although CORBA IDL types are supposed to handle "compatibility" by wrapping native types as IDL types, CORBA applications can create requests (method calls) dynamically and invoke them, leading to potential mismatches. So CORBA defines an exception to handle this case:

$$\kappa(t', m, p) = \begin{cases} (t', m, p) & m \text{ applicable to } t' \\ \texttt{NO\_IMPLEMENT} \in E & \text{otherwise} \end{cases}$$

7.3.2.6 Konstantas's OOI Match

The H-TMSL language of Konstantas's Object-Oriented Interoperability model [43] can describe compatibility criteria that maps the parameters of m to those of m'

in the specified manner, as in the example of Section 7.3. Let p_1 and p_2 be the two parameters of method m in that example.

$$\kappa(t', m, (p_1, p_2)) = (t', m', (\textsc{null}, p_1 + p_2, t'))$$

7.4 Type Compatibility in PolySPIN

Now that we have a vocabulary for describing conformance and compatibility in terms of method invocations, we return to our discussion of polylingual systems.

7.4.1 The Message-Not-Understood Problem

The *message-not-understood* problem is the classic type error in object-oriented programming: the attempt to invoke a non-applicable method on an object. If an object-oriented system is statically checkable, the problem cannot arise because the system will not compile. If the system is not statically checkable, the problem may arise, and run-time checks typically are generated at compile time to prevent such erroneous method calls. Interoperability exacerbates the problem because interoperating components are not only written in distinct languages, but also likely to have been developed independently, and statically checking such a system as a whole is beyond the current state of the practice.

In many approaches to interoperability, the message-not-understood problem does not arise because the supported type system is limited. For static typing in CORBA, for example, native types are wrapped by the interoperability engineer to produce IDL types, and from then on, application programs operate on the IDL types rather than the native types. If two native types t and t' are mapped to the same IDL type, then t and t' are considered "compatible." Because both native types are required to implement all methods specified of the IDL type, the message-not-understood problem cannot arise in this case. (Dynamic typing in CORBA is another matter, as shown in Section 7.3.2.5.)

In a polylingual system with relaxed matching, the message-not-understood problem can easily arise if method matching is not done carefully. Let c and c' be instances of distinct types t and t', respectively, and let m be a method applicable to type t but possibly not applicable to type t'. In order for types t and t' to be considered polylingually compatible, we must ensure that any attempt to invoke method m on instance c' must have a sensible result. In a polylingual system, it is the responsibility of the interoperability mechanism to ensure the type safety of such an invocation. This may be done in various ways.

Let (t', m, p) be an invocation of method m with parameters p on an instance of type t'. The possible outcomes of this invocation are:

- *Success*: the method is applicable to type t'. This is represented as $\kappa(t', m, p) = (t', m, p)$.
- *Error*: permit an erroneous invocation to be made, possibly causing a crash. This is represented as $\kappa(t', m, p) = ()$, the empty invocation.
- *Raise an exception.* This is represented as $\kappa(t', m, p) = e \in E$.
- *Modify component c'*: insert an implementation of method m into component c', thereby changing its type to be a subtype of t', say, t''. This is represented as $\kappa(t', m, p) = (t'', m, p)$.
- *Call a different method*: redirect the invocation to invoke a different, applicable method of type t'. This is represented as $\kappa(t', m, p) = (t', m', p')$. The alternative method m' and new parameters p' could be supplied automatically by the interoperability mechanism, or interactively by the interoperability engineer.

7.4.1.1 Missing Method Manager

The current version of the PolySPIN framework, PolySPIN V3, incorporates an abstract component for ensuring type safety. Its means of preventing the message-

not-understood problem is not preset; rather, PolySPIN V3 has a generic component, called a *Missing Method Manager* (MMM), that may be instantiated to have behaviors such as those above.[2] The MMM is based on the functional definition of compatibility given in Section 7.3, selecting an alternative action to occur in place of a type-unsafe method invocation.

The MMM is a subcomponent of the Matcher (Section 5.4.1) in PolySPIN V3. After the methods of two types have been matched, the MMM locates all unmatched methods and generates code to prevent them from being used in a type-unsafe manner, according to the behavior selected above. Suppose the types t and t' are being matched, and t has a method m that is unmatched by any method of t'. The MMM does the following:

- If *raise an exception* is chosen, the MMM generates a new method $t'.m$ that does nothing but raise a user-supplied exception.
- If *modify component* is chosen, the MMM adds a user-specified method m to the type.
- If *call a different method* is chosen, the MMM generates code to redirect a call of $t'.m$ to invoke a different, user-specified method m' of t', just as if $t.m$ and $t'.m'$ had been matched by the Matcher.

The Missing Method Manager is a work in progress. It currently requires a lot of effort on the part of the user. Every time PolySPINner V3 is run with the MMM enabled, the user must specify all MMM behavior from scratch: that is, the MMM does not store the user's choices for pairs of types it has previously encountered. An improved MMM would not only remember the user's selections from previous runs, perhaps storing them in a database for reuse, but also would be "trainable" to take

[2]Though crashing is obviously a poor choice.

the correct action in situations similar to those it has encountered before. But this is for the future. It is doubtful that the MMM can ever be completely automated. A tremendous amount of semantic knowledge is required to choose an appropriate alternative action for a given method call. But perhaps the MMM can learn to be useful in common cases.

7.5 Summary

Chapter 6 established a formal foundation for reasoning about type matching and compatibility in a multilanguage setting, particularly in polylingual systems. This chapter has considered type safety in more detail and describes a more generic mechanism for addressing type safety issues in PolySPIN. Together, they provide a solid foundation for future work the area.

This chapter also presented a taxonomy of approaches to type compatibility and illustrates how several well-known notions of type compatibility fit into the taxonomy. PolySPIN itself is characterized as focusing on declaration-level type compatibility. PolySPINner implements a preliminary Missing Method Manager to prevent unmatched methods from being invoked polylingually in a type-unsafe manner.

CHAPTER 8

POLYSPIN VS. CORBA

CORBA, invented and maintained by the Object Management Group (OMG), is currently the reigning champion of interoperability in industry. It is a juggernaut with a massive, 650-page specification[1] [55] and tremendous political clout. Vendors hasten to make their applications—and their own interoperability mechanisms—CORBA-compliant so they can share in the dream of universal "plug-and-play" software components, in which components from different vendors can interoperate seamlessly.

CORBA is also the most visible representative of the IDL (Interface Definition Language) approach to interoperability, in which a foreign type system is imposed upon a set of components. Each component's types are then translated into the foreign type system, allowing the components to interoperate by transmitting and receiving instances of those types. In contrast to CORBA, PolySPIN has no IDL, but the differences run deeper than this simple statement.

In this chapter, we compare and contrast CORBA according to the criteria of the Interoperability Manifesto, specified in Chapter 3, and judge their relative strengths and weaknesses. Relevant quotations from the CORBA 2.0 specification appear in the text, tagged with their section numbers from the specification.

---

[1] This does not include the equally massive CORBAfacilities [56] and CORBAservices [57] specifications that document OMG-sanctioned add-ons for CORBA, and several dozen other documents.

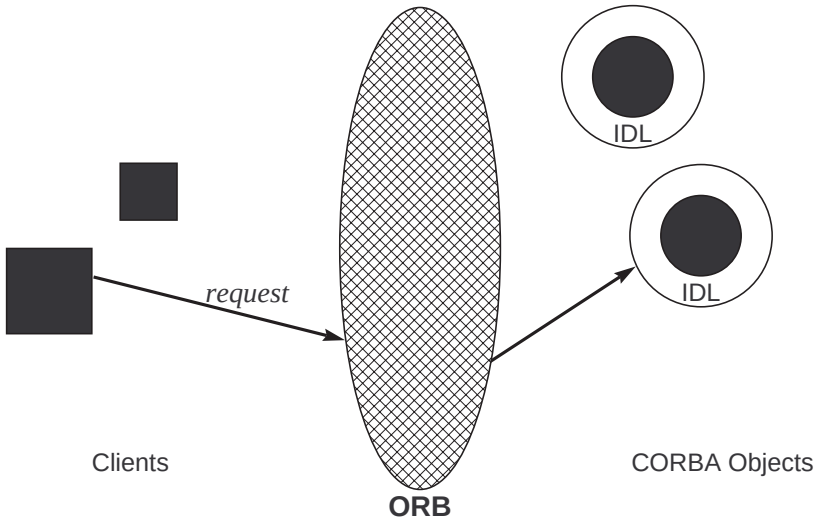


Figure 8.1. The components of CORBA.

8.1 CORBA Overview

Before undertaking the analysis, we introduce CORBA. This section covers its components, terminology, and view of interoperability.

8.1.1 CORBA Components

Interoperability in CORBA consists of multiple programs, called *clients*, transmitting messages, called *requests*, to uniquely identifiable, encapsulated entities, called *objects*, as shown in Figure 8.1. CORBA objects respond to requests by executing methods, called *operations*, optionally sending a return message back to the calling client. In other words, CORBA transmits information between clients and objects by remote procedure call.

Clients communicate with CORBA objects indirectly via server entities called *object request brokers* (ORBs). ORBs provide object name service and location service, transparently locating callees and forwarding requests to them. Clients, objects, and ORBs may be distributed on multiple machines. An ORB servicing one set of objects

```
interface Car : Vehicle {                    // inherits from Vehicle
  void Drive();                              // Drive the car
  Person Driver();                           // Who's the driver?
  Boolean Put_Into_Trunk(sequence of Item); // Fill trunk
  Boolean Add_Gas(integer gallons);          // Fill gas tank
};
```

Figure 8.2. An IDL interface.

may communicate with other ORBs, servicing other objects, to transmit data across physical and logical boundaries.

Each CORBA object has a type, called its *interface*, that defines the operations[2] available to clients via that CORBA object. Interfaces are written in CORBA's Interface Definition Language (IDL). IDL is declarative only, containing no control statements, and is based on the syntax of C++. It supports multiple inheritance, though only of interfaces, not of implementations of operations. Operations are declared by specifying their signatures, as in Figure 8.2. Once specified, operations must then be implemented using a traditional programming language, and an *IDL compiler* generates stubs to map the CORBA operations to the target language methods.

8.1.2 CORBA Interoperability

Like many RPC approaches, CORBA views interoperability primarily from a low-level standpoint, focusing on protocols for transmitting primitive types from ORB to ORB. At a higher level, Chapter 10 of the CORBA specification also specifies an "interoperability architecture," which is a more general view of interoperability, followed by possible strategies for implementing it. Unfortunately, no strategy is prescribed over others. Nevertheless, some information can be gleaned from this interoperability architecture.

[2]Interfaces may also have attributes—named values that are accessible via "get" and "put" operations—but these are not relevant to the discussion.

CORBA defines interoperability as a crossing of "boundaries" between "domains." A domain is a collection of CORBA objects that have some characteristic in common.

> *. . . a domain is a scope in which a collection of objects, said to be members of the domain, is associated with some common characteristic; any object for which the association does not exist, or is undefined, is not a member of the domain. A domain can be modeled as an object and may be itself a member of other domains.* [10.3.1]

A boundary is defined as:

> *...the limit of the scope in which a particular characteristic is valid or meaningful. When a characteristic in one domain is translated to an equivalent in another domain, it is convenient to consider it as traversing the boundary between the two domains.* [10.3.1]

Multilanguage interoperability, then, would be characterized by CORBA as the crossing of domains defined by programming languages: for example, from the "C++ domain" into the "Java domain." CORBA's notion of domain is more general than this, however, as language need not be the boundary delimiter.

CORBA requires that the object models of two domains be "compatible" in order for interoperability to be possible between the domains. The CORBA specification defines compatibility as strict compliance with the CORBA Object Model:

> *This specification assumes that both domains are strictly compliant with the CORBA Object Model and the CORBA V2.0 Core specifications. This includes the use of OMG IDL when defining interfaces, the use of the CORBA Core Interface Repository, and other modifications that were made to CORBA V1.2. Variances from this model could easily compromise some aspects of interoperability.* [10.3.2]

Thus, CORBA has no facility for relaxed matching of the sort described in Chapter 6. In the CORBA view, a legacy application must be modified to be 100% CORBA-compliant, or else no guarantees of interoperability can be made.

8.2 A Comparison Via the Interoperability Manifesto

The Interoperability Manifesto of Chapter 3 is designed to facilitate the comparing and contrasting of interoperability approaches. In this section, each feature of the Manifesto is applied to CORBA and PolySPIN to highlight their similarities and differences.

8.2.1 Type Expressiveness

Type expressiveness refers to support for sharing diverse kinds of types, including primitive types, structured types, pointer types, and abstract data types. CORBA-compliant applications may transmit and receive instances of any type that is expressible in IDL. This includes primitive types (integer, character, Boolean, floating point, octet), structured types (record, union, enumeration, array, string), an `Any` type that can represent an arbitrary value, and interface types, a.k.a. objects with methods. Noticeably missing are pointer types, though presumably pointers can be stored in `Any`s.

CORBA's support for primitive and structured types is unsurprising and typical of RPC-based approaches. We focus, then, on its support for abstract data types described in IDL.

IDL types are defined in an object-oriented manner, but with some severe restrictions on inheritance. First, although methods may be inherited, they may not be overridden, that is, redefined in a subtype. This can have a significant impact on the ease of making a legacy application CORBA-compliant. For example, suppose a

C++ legacy application from the automotive industry defines the type `Vehicle` and its subtype `Car` as follows:

```
// C++
class Vehicle {
  public:
    virtual void Drive();
};
class Car : Vehicle {
  public:
    void Drive();  // overloaded
};
```

in which the `Drive` operation has been overridden in the subtype. Suppose we want to make both `Vehicle` and `Car` into CORBA objects, by creating IDL interfaces for each to expose the `Drive` method. Since IDL forbids overriding, the IDL interfaces for these classes must take one of the following approaches to avoiding the problem:

1. Not expose the `Drive` method in the IDL interface of `Vehicle`. This is clearly a bad solution, as `Vehicle` requires this operation.

2. Not expose the `Drive` method in the IDL interface of `Car`, causing `Car` to inherit the operation's interface from `Vehicle`. This may seem a simple solution, but in fact it is type unsafe. CORBA does not support virtual methods. Thus, the type of the self parameter in the IDL operation `Car.Drive` will be `Vehicle`, not `Car` as in the original C++ method. If the body of `Car.Drive` refers to a member of `Car` not present in `Vehicle`, and `Car.Drive` receives a `Vehicle` instead of a `Car` as its self parameter, this will result in a type error.

3. Rename `Drive` in one of the IDL interfaces. (See below.)

4. Not mirror the C++ inheritance relationship. That is, create IDL interfaces for `Vehicle` and `Car` but don't have the `Car` IDL interface inherit from the `Vehicle` IDL interface. (See also below.)

The last two cases are also problematic. Consider a legacy operation that returns an object of type `Vehicle`, such as `GetNextVehicle`, an iterator for a set of `Vehicle` objects. The legacy application may well contain a code fragment like the following:

```
Vehicle *v;
while (v = GetNextVehicle())
  v->Drive();
```

This fragment will no longer be possible if `Vehicle` and `Car` are replaced by IDL types, since it relies on both the C++ inheritance relationship and the fact that both types have a `Drive` method. The fragment will need to be rewritten to use new logic if any of the vehicles returned by `GetNextVehicle` could potentially be `Car` objects.

A second restriction on inheritance is that an IDL type cannot inherit the same-named method from multiple supertypes, even though IDL supports multiple inheritance. So while a type may have multiple supertypes, those supertypes may not have any method names in common. This restriction is not found in other languages supporting multiple inheritance, such as C++ and CLOS, and can cause difficulty for interoperability engineers attempting to encapsulate legacy types in IDL interfaces.

Thus, we see that CORBA's IDL, being more restrictive than the programming languages it supports, can act as an unwanted filter of type expressiveness between two languages. We have seen that IDL cannot encapsulate a simple legacy application without losing expressiveness or interfering with existing inheritance relationships.

We now turn to PolySPIN. Given legacy applications written in two languages, PolySPIN permits them to communicate instances of any type that is representable in both programming languages. Since there is no intermediate IDL, the interoperability

designer has more flexibility in mapping the two representations to each other. In short, a direct mapping between two languages is at least as expressive as an IDL mapping between two languages, and likely more expressive.

The major tradeoff of this flexibility is that PolySPIN requires a quadratic number of language mappings, as discussed in Section 5.6.1, in order to achieve interoperability between n languages. This extra work, however, is done only once per language pair, not once per application. In addition, the number of languages supported by PolySPIN is not likely to be large; and the quadratic growth function is actually $n(n-1)/2$, which is not much larger than n for $n \leq 5$ languages.

Inheritance in PolySPIN is no different from inheritance as found in the languages it supports, since PolySPIN defers the handling of inheritance to the run-time system of each language. Thus, overriding, overloading, and traditional multiple inheritance do not cause new problems for PolySPIN, which lets each language's run-time system locate and invoke the correct method in response to a call.

8.2.2 Nonintrusiveness

Nonintrusiveness refers to the amount and kinds of modification necessary to legacy components in order for them to be made interoperable. In order to make two legacy applications A_1 and A_2 able to interoperate via CORBA, several steps are required.

1. Identify potentially compatible types t_1 and t_2, found in applications A_1 and A_2, respectively.

2. Create a single IDL type t_{IDL} to represent both t_1 and t_2. If it is possible to map both t_1 and t_2 directly to t_{IDL}, method by method, parameter by parameter, do so. Otherwise, write (by hand) all translation routines necessary to marshal instances of t_1 and t_2 into instances of t_{IDL} and back. This may require breaking

encapsulation to reveal the internals of complex parameter types in order to marshal them.

3. Run the CORBA IDL compiler on type t_{IDL} to create stubs.

4. Write (by hand) a second layer of stubs so that the compiled IDL stubs will invoke their corresponding methods in t_1 and t_2.

5. Throughout the applications A_1 and A_2, replace instances of t_1 and t_2 with instances of t_{IDL}, as needed. This includes replacing all method calls on instances of t_1 and t_2 with their corresponding t_{IDL} calls.

6. Modify the main programs of applications A_1 and A_2 to register with an ORB at the beginning and unregister at the end.

Most of these steps can be accomplished by wrapping, linking, or other nonintrusive coding strategies, but step 5 is highly intrusive. Due to the introduction of a wholly new type in step 2, step 5 may require significant modifications of the legacy code. For example, let us return to the example of Section 8.2.1:

```
class Car : Vehicle {
  public:
    void Drive();  // an overridden method
};
```

and let the previously-mentioned type t_1 be a simple C++ type representing an automobile. Suppose an IDL type `ICar` is defined to represent this type. Note that type `Car` is a subtype of type `Vehicle`, which is not involved in the IDL mapping. Application A_1 might very well contain the example code fragment from Section 8.2.1:

```
Vehicle *v;
while (v = GetNextVehicle())
  v->Drive();
```

that iterates through a set of vehicles, invoking `Drive` on each, presumably a method defined on superclass `Vehicle` as well. The new IDL type `ICar`, however, is not a subtype of `Vehicle`; thus, when `ICar` replaces `Car`, this elegant code fragment must be rewritten because it depends on the subtype/supertype relationship of `Car` and `Vehicle`. Likewise, the function `GetNextVehicle` relies on this subtype/supertype relationship and will need to be replaced.

The use of primitive types in CORBA also may be intrusive in step 5. The CORBA specification advises "programmers concerned with portability" [16.5] to use CORBA primitives, such as `CORBA::Char` and `CORBA::Long`, instead of the native language's primitives, implying further modification of legacy code. Interlanguage compatibility of primitive types is currently handled by PolySPIN in an ad hoc manner (see the future work in Chapter 12), but the intent is to handle them in the language-to-language mappings for each pair of languages, so that developers may use the primitive types supplied in each language.

Step 2 mentions the possibility that data encapsulation may need to be violated to marshal an instance of an abstract type. Suppose we are given the class:

```
class Vehicle {
  public:
    Engine e;
};
```

where `Engine` is defined elsewhere as, say:

```
typedef unsigned char Engine;
```

In order to create an IDL wrapper for `Vehicle`, the definition of `Engine` must be exposed to the developer of the wrapper, so its instances (now known to be unsigned chars) can be marshaled. This not only breaks encapsulation but also leads to a

consistency maintenance issue. If the `Engine` definition were to change, the `Vehicle` IDL wrapper must be updated as well, even though the C++ `Vehicle` class has not changed.

Finally, step 6 is also intrusive. In the best case, however, it will consist of adding only a few lines of code to the main program.

In order to make two legacy applications able to interoperate via PolySPIN, different steps are required:

1. Identify potentially compatible types t_1 and t_2 in the two applications.

2. Define a compatibility criterion for the mapping between types t_1 and t_2. If the types are highly similar, this may be straightforward, but if they are very different, this can be quite complex. However, this is a direct mapping between the types, not filtered through an intermediate IDL with restricted type expressiveness, and it does not require the legacy code to be modified.

3. Run PolySPINner to modify the method implementations of t_1 and t_2 with respect to the compatibility criterion.

4. Modify the main programs of A_1 and A_2 to register with the PolySPINner nameserver at the beginning and unregister at the end, and to fetch any remote objects via the nameserver.

Of these steps, only step 4 is intrusive to the legacy programs, but no more so than CORBA's analogous step 6. In fact, any application that communicates with remote objects requires a step like this. As stated in Section 1.4, it is assumed that software components must be capable of referring to objects of other languages, and such steps are merely implementations of this assumption.

For primitive types, PolySPIN uses only those found in each language. It still lacks a robust mechanism for handling matchings between primitive types, but the subject has been well-studied by others [4].

PolySPIN does not need to break the encapsulation of an abstract data type in order to marshal it, unlike CORBA. PolySPIN's naming mechanism [40] permits foreign objects to be passed between applications by transparently marshaling only their names.

8.2.3 Type Safety

8.2.3.1 The Message-Not-Understood Problem

Recall from Chapter 7 that the message-not-understood problem is the attempted invocation of a non-applicable method on an object. By using an IDL, CORBA tightly controls the applicability of methods and prevents the message-not-understood problem from occurring statically, as follows. Because the IDL compiler generates stubs for every method of every IDL type, an application that invokes one of these methods statically is guaranteed that the method is applicable to the target object. This guarantee comes at the cost of lost type expressiveness discussed in Section 8.2.1.

Problems arise, however, for method calls created via CORBA's Dynamic Invocation Interface. The DII permits an application to construct method calls dynamically, creating a "request" object and inserting "parameter" objects into it, and then sending the request. There is no guarantee that a dynamically-created request corresponds to any implemented method; it is the responsibility of the application to create applicable requests. If an ORB receives a request for a non-applicable method, the ORB returns a predefined exception, `CORBA::NO_IMPLEMENT`. A similar exception, `CORBA::BAD_OPERATION`, is predefined for method calls that are ill-formed in some other way.[3]

PolySPIN is much more susceptible to the message-not-understood problem, due to its support for relaxed matching criteria. As shown in Chapter 6, it is possible to

[3]The CORBA specification does not precisely define the conditions under which these exceptions are raised.

create type unsafe matching criteria, in which case PolySPINner's Missing Method Manager (Chapter 7) will detect the unsafety and prompt the user for instructions on how to resolve the problem. If the user's instructions do not guarantee type safety, PolySPINner blindly obeys them. If only type safe matching criteria are used, however, as explored in Chapter 6 and supplied with PolySPINner V3, a polylingual system is guaranteed to be free of the message-not-understood problem.

8.2.3.2 `Ptr` and `Var` Discrepancies

When a CORBA IDL type is compiled into C++, two C++ types are generated. Their names are the same as the original type, suffixed with `ptr` and `var`. For instance, an IDL type `Car` would become the C++ types `Car_ptr` and `Car_var`. Despite the names, both are pointers. They differ only in resource tracking: `A_var` instances, when deallocated or reassigned, automatically release their object references. `A_ptr` instances don't; the programmer must allocate and free memory manually.

The CORBA specification states:

> *For many operations, mixing data of type A_var and A_ptr is possible without any explicit operations or casts. However, one needs to be careful in doing so because of the implicit release performed when the variable is deallocated.* [16.3.1]

Thus, `A_var` and `A_ptr` are "sometimes" mixable in operations, and "programmer beware" is the only rule. This is not safe. As a result, C++ applications that are type-safe with one CORBA implementation might be unsafe in others, because:

> *These types need not be distinct—A_var may be identical to A_ptr, for example—so a compliant program cannot overload operations using these types solely.* [16.3.1]

PolySPIN has no analogous issue, since it uses an application's original types and does not have an IDL.

8.2.3.3 Inheritance

In Section 8.2.1, an example was given of type unsafety in CORBA when attempting to encapsulate overridden methods. On the other side of the coin, CORBA also disallows some operations that are normally considered type safe. An example is narrowing the type of an instance to one of its supertypes, which is forbidden for `var` instances. For example:

```
// CORBA
interface Vehicle {...};
interface Car : Vehicle {...};

// C++
Vehicle_var vehicle;
Car_var car;
car = vehicle;     // widening, type-unsafe as usual
vehicle = car;     // narrowing, type-safe but illegal in CORBA
```

This narrowing is illegal only for `var` instances. It is legal for `ptr` instances, further exacerbating the problem discussed in Section 8.2.3.2, where `var` and `ptr` types are "sometimes" interchangeable.

8.2.3.4 The `Any` Type

CORBA specifies a type, called `Any`, that may store values of any other type. It is similar to `void*` in C and C++, except that `void*` stores only pointer values, whereas `Any` is not restricted to pointers. An instance of `Any` stores both a value and a typecode, which is an ID representing the value's type. The typecode is not visible to the interoperability engineer, as it is generated and maintained internally by the `Any`.

The `Any` type is called "type-safe" by the CORBA spec, in that an instance of `Any` will never have a mismatched typecode and value. In the C++ mapping, this is ensured (the specification claims) by overloading the `Any` assignment operator for every possible type that could be on the right-hand side of an assignment. The C++ mapping implementation is responsible for generating these overloaded operations.

In CORBA's C++ mapping, `Any` is treated differently from all other built-in types, in that it does not use the traditional assignment operator, `=`, as follows:

```
Any a;
a = 19;  // illegal
```

Instead, assignment is accomplished using the `<<=` operator, and retrieval via the `>>=` operator.

```
Any a;
a <<= Long(19);
```

This is problematic for type safety. If the value in the `Any` is not of the same type as the type of the right-hand side, CORBA specifies that the `>>=` operation should return the constant `FALSE`. However, `FALSE` need not be in the range of the legal return values of an assignment! The CORBA C++ mapping assumes that the `>>=` operator will be used only in the following construction:

```
Long value;       // Or any other CORBA type
Any a;
a <<= Long(42);
if (a >>= value) {
  // ... use the value ...
}
```

but conceivably, the `>>=` operator could be used in other contexts. Since the `>>=` operator returns the value of its right hand side, the following code will compile without error:

```
class MyClass { ... };  // something suitably complex
MyClass x, y;
Any a <<= MyClass(x);
y = (a >>= x);
```

However, this code is not type safe, because the last line could evaluate to:

```
y = FALSE;
```

which is not type correct, since variable `y` is of type `MyClass`, and `FALSE` (a.k.a, zero in C++) is not an instance of `MyClass`.

The above problem does not occur in every language mapping. In the Java IDL mapping, for example, the `Any` type is mapped to a real class, `org.omg.CORBA.Any`, that raises an exception (`CORBA::BAD_OPERATION`) if extraction is attempted with mismatched types.

8.2.3.5 Name Conflicts

It is possible, at least in the C language mapping, for the IDL compiler to generate identifiers that are coincidentally the same as other identifiers used in a legacy application, generating code that will not compile.

> *The fact that [IDL] constants are #defined may lead to ambiguities in code. All names mandated by the mappings for any of the structured types [omitted here] start with an underscore.* [14.6]

PolySPINner reserves the prefix `__POLYSPIN`. A legacy application that begins an identifier with this prefix runs the risk of the same naming conflict as CORBA, and

would therefore be in violation of the PolySPIN spec. However, the likelihood that an application will coincidentally have an identifier beginning with `__POLYSPIN` is less than that of having an identifier beginning with an underscore, CORBA's reserved prefix for identifiers.

8.2.4 Type Compatibility

Type compatibility refers to the tolerance of differences between types in order for them to be considered the same type by an interoperating application. CORBA is concerned with two kinds of type compatibility:

1. Compatibility among CORBA types defined in IDL.

2. Compatibility between a programming language type and its corresponding IDL type.

8.2.4.1 Compatibility Among IDL Types

IDL types can be evaluated with respect to the declaration and behavioral levels of type compatibility, as discussed in Chapter 7. There is no representation-level compatibility between IDL types, since CORBA considers only the abstract interface of a type—that is, its set of operation signatures—to be relevant to interoperability.

At the declaration level, two IDL types are compatible if they are the same type, or if they have a subtype relationship. In the terminology of Chapter 7, the compatibility criterion for two types t and t' is:

$$\kappa(t', m, p) = (t', m, p)$$

where m is any method of type t, and p is any legal set of formal parameters of method m. This is exactly the same criterion as in Section 7.3.2.3.

At the behavioral level, two IDL types are compatible if they both implement a given method. An ORB, upon receipt of a request, may forward that request to any object capable of servicing it: that is, any object implementing a method that can service the request. For example, suppose an application wants to print a file. It sends a `Print` request with a filename argument, and the receiving ORB may route that request to any object providing a printing service. These objects need not be of the same type, nor have a subtype relationship, nor be related in any way apart from providing a `Print(filename)` operation.

8.2.4.2 Compatibility Between IDL and Other Languages

A CORBA language mapping often specifies the representation-level compatibility between IDL types and their corresponding programming language types. For example, the C language mapping specifies that the CORBA primitive type `boolean` is mapped to the C type `unsigned char` with the values zero (false) and one (true). At other times, CORBA does not precisely specify representation-level compatibility. For example, IDL structs (records) are mapped to C structs, but the C structs may or may not include padding on the end.

At the declaration level, CORBA does not specify compatibility between IDL and programming language types. An interoperability engineer has great flexibility in implementing the methods of an IDL type, as long as the implementation is consistent with the IDL interface. For example, a single IDL method may map directly to a single programming language method, parameter by parameter; or it could map to a function that calls the methods of multiple types.

The parameters of an IDL method, however, are restricted to be of types that IDL can represent, which are not likely to cover all types representable in the programming language. This can lead to awkward mappings of types. For example, as mentioned in Section 6.5.1, IDL has mappings to both C and C++, but it has no pointer type.

Thus, C and C++ applications cannot directly communicate pointer types, even if these types are identical at the representation level.[4] Another example is that IDL cannot represent the difference between an empty string and a null string, although some of its supported languages, such as C++ and Java, make that distinction.

At the behavioral level, CORBA specifies nothing concerning the compatibility of IDL types and programming language types. It is up to the interoperability engineer to ensure that IDL methods are semantically compatible with the native language methods they encapsulate.

8.2.4.3 Compatibility in PolySPIN

Type compatibility in PolySPIN has been extensively described in Chapters 6 and 7. At the representation level, PolySPIN assumes the existence of some means of equating primitive types across languages. (PolySPINner uses an ad hoc mechanism; see the future work in Chapter 12.) At the declaration level, PolySPIN supports the creation of arbitrary, user-supplied compatibility criteria. PolySPINner supplies a library of such criteria. At the behavioral level, PolySPIN currently specifies nothing (again, see future work).

Thus, declaration-level compatibility in PolySPIN is much more flexible than that of CORBA. This flexibility comes at a price, however, because the message-not-understood problem is more likely to occur for non-type-safe compatibility criteria, as discussed in Section 8.2.3.

8.2.5 Diversity

Diversity refers to the number of languages supported by an interoperability approach or mechanism. Language bindings for CORBA have been officially specified for C, C++, Ada, Java, Smalltalk, and COBOL. ILU [36], a CORBA-compatible inter-

[4] If components are interoperating across address spaces, pointer objects must be passed abstractly regardless.

operability mechanism, defines additional language bindings for CLOS and Python, but these are not sanctioned by the OMG.

PolySPIN has language bindings for C++ and CLOS, and a Java binding is in the works. Thus, at present, CORBA clearly outshines PolySPIN for diversity of language.

8.2.6 Transparency

Transparency refers to the invisibility of interoperability support in the source code of interoperating components. A uniform approach hides language differences, and a seamless approach hides interoperability completely.

CORBA is uniform but not seamless. Since all remote method calls are processed by ORBs, they maintain a consistent interface regardless of the language of the callee. CORBA calls, however, are distinguishable from ordinary method calls because of their use of CORBA types as parameters. In addition, CORBA requests made via the Dynamic Invocation Interface have a syntax entirely different from ordinary method calls. (To be fair, the Dynamic Invocation Interface provides a capability missing from many languages: to construct method calls at run-time.)

Numerous aspects of CORBA lack seamlessness, that is, reveal the presence of interoperability within components. Its biggest seam is IDL itself; in order to make a program CORBA-compliant, one must specify the program's external interface using a foreign type system, not the type system of the language in which the program is implemented.

CORBA requires the use of at least two distinct inheritance hierarchies: that of IDL, and those of the language(s) being generated by the IDL compiler. Interoperability engineers must be aware of both because the two hierarchies clash at times, as shown in the `Car/Vehicle` example of Section 8.2.1. In fact, CORBA specifies that inheritance among IDL types need not be implemented as inheritance among target

language types. This problem does not arise in PolySPIN, since it does not have an inheritance hierarchy of its own.

CORBA's `Any` type does not support the assignment operator, as explained in Section 8.2.3.4. Thus, developers must treat `Any` objects differently from all other types where assignment is used. Hence, another seam.

PolySPIN method calls, on the other hand, are seamless. Polylingual method calls are indistinguishable from local method calls. For example, in C++, the following fragment could seamlessly be accessing `Vehicle` objects of different languages:

```
Vehicle *v[3];
// initialization
for (i=0; i<3; i++)
  v[i]->Drive();
```

Initialization of objects is only uniform, not seamless. Local objects are created in the normal manner, but foreign language objects must be fetched via a nameserver. As stated in the assumptions of Section 1.4, it is a requirement of any interoperability approach that software components can somehow contact objects of other languages. For CORBA, an ORB call is required to fetch a remote object. In PolySPINner V3, the previous example with initialization looks like this:

```
NameServer nameserver;
Vehicle *v[3];
v[0] = new Vehicle;
v[1] = nameserver.Fetch("my vehicle");
v[2] = nameserver.Fetch("another vehicle");
for (i=0; i<3; i++)
  v->Drive();
```

So although one can distinguish between local and remote objects, one cannot discern language differences from this source code. For example, the object named "my vehicle" could be a C++ object and "another vehicle" could be a CLOS object. Hence, uniformity.

8.2.7 Automation

Automation is the extent to which components can be made interoperable without human intervention. Both CORBA and PolySPINner support some form of automation. CORBA's IDL compiler generates stubs by which IDL methods are translated into target language methods. These stubs, however, are not usable as is. They must be integrated with legacy components via a second set of stubs, which are created manually by the interoperability engineer. This is due to a naming problem. Since IDL does not represent the mapping of an IDL method to the target method it encapsulates, the IDL compiler does not know the name of the target method and cannot generate a call to it. Instead, it generates a call to a nonexistent method in the target language. The interoperability engineer must then implement that new method, and in its body, invoke the desired target method.

PolySPINner generates replacement method implementations for types, permitting their instances to be accessed polylingually. Since there is no change to the method specifications, these generated bodies are drop-in replacements for the original ones. Once PolySPINner has been run, a type needs only to be recompiled and relinked in order for its instances to be polylingually interoperable. The current implementation of PolySPINner V3 does not perform the physical replacement of the old implementations with the new, however, but generates them into a separate file, so the interoperability engineer must perform the replacement by hand. This is due only to a bad design decision in the parser which, instead of storing everything it reads in the input, stores only the code it needs to process, as elaborated in Section 11.6.

As will be seen in Chapter 10, PolySPINner may also generate, on request, a main program to function as a server for a set of objects, if such a program is desired. This program may be used as is, or—more likely in the case of megaprogramming—integrated with the existing main program to make its objects available to other components.

8.2.8 Performance

Performance refers to the speed and space overhead present in a system due to an interoperability mechanism. As CORBA and PolySPIN are models, not implementations, it is not possible to gauge their performance precisely. Nevertheless, they should be of roughly the same computational complexity as both are dominated by low-level RPC or foreign function call mechanisms. Further performance-related information will be given in Chapter 11.

8.2.9 Extensibility

Extensibility refers to the ease of adding support for a new language to an interoperability mechanism. Unlike virtually all other interoperability approaches, CORBA is extensively documented, from its low-level ORB-to-ORB communication protocol to the mappings for six different languages. To add support for a new language, it is necessary to map all features of IDL to those of the target language, and then supply an IDL-to-target-language compiler.

PolySPIN also has a documented means of adding support for a new language: Section C.7.3 in Appendix C. While this appendix documents the addition of language support, it has never been used fully by anyone other than the author,[5] whereas CORBA's specification has been used by others to create six language mappings. Thus, PolySPIN's ease of extensibility has yet to be rigorously tested. In addition, as

[5] A Java language mapping is in progress by a third party.

stated in Section 5.6.1, the addition of an nth language to PolySPIN requires $n-1$ mappings to the other languages, whereas CORBA requires only a single mapping between the language and IDL.

8.3 Summary

The features of CORBA and PolySPIN are summarized in Table 8.1. Because CORBA is so intrusive and non-transparent, it seems best suited for the "easiest case" interoperability scenario, described in Section 5.2.1, in which applications are built from scratch to work with CORBA. It is also a viable, though inconvenient, solution for common case and megaprogramming interoperability when the number of languages involved is large; few other approaches support so many languages.

PolySPIN seems better suited for the "common case" and "megaprogramming" scenarios, since it is seamless and largely nonintrusive. But it is not suitable for all applications. There are only 2 language bindings, so currently PolySPIN is not useful for more than these languages. PolySPIN has great type expressiveness, but the ease of creating compatibility criteria, versus that of creating IDL types, has not fully been assessed through experimentation.

Table 8.1. CORBA vs. PolySPIN.

Feature	**CORBA**	**PolySPIN**
Type Expressiveness	Types representable in IDL.	Types representable by the intersection of the languages.
Type Safety	No message-not-understood problem in static calls, but dynamic calls may encounter it.	Message-not-understood is possible if compatibility criteria are not type safe.
Type Compatibility	Subtype compatibility between IDL types. User-defined mappings between IDL stubs and target language methods, using only IDL types.	Relaxed compatibility as in Chapter 6.
Diversity	6 languages.	2 languages.
Transparency	Uniform.	Seamless.
Nonintrusiveness	May be extremely intrusive to legacy applications, requiring the replacement of legacy types in name and semantics.	May be slightly intrusive to main program of legacy application.
Automation	IDL compiler generates stubs, which must unfortunately be integrated by hand to invoke legacy methods.	PolySPINner instruments the bodies of methods transparently.
Performance	Dominated by RPC.	Dominated by Locator, be it RPC, FFC, etc.
Extensibility	Large, unwieldy CORBA specification has been followed to create IDL bindings for six languages.	Documentation for adding a language binding, but not extensively tested.

CHAPTER 9

POLYSPINNER

PolySPINner is a toolset implementing the PolySPIN approach to interoperability. The current version, PolySPINner V3, consists of generic components: Parsers for converting types into an intermediate representation, Matchers for matching types using exact and relaxed criteria, and Generators for creating modified method bodies to allow types to be accessed polylingually.

This chapter provides an overview of PolySPINner's history, specification, design, and implementation. In addition, two appendices go into detail about PolySPINner's internals (Appendix B) and use (Appendix C).

9.1 History

9.1.1 Version Zero

The original PolySPINner V0 was a proof-of-concept prototype created by Alan Kaplan [40] prior to the present research. It consisted of parsers for a small subset of C++ and CLOS, a minimal intermediate representation for types, and a hard-wired code generator. It did not understand inheritance relationships among types, required the strictest possible matching between C++ and CLOS types, supported only one type definition per language at a time, assumed that all CLOS methods in a file belonged to the single CLOS class in the file, and had many static, hard-coded limits. In addition, the application had an ad hoc architecture and was not easily extendible. PolySPINner V0 was applied to a few small, customized examples. It was not tested with third-party code.

9.1.2 Version One

PolySPINner V1 was the first version developed as part of this dissertation. A transitional version, PolySPINner V1 was largely a reorganization and encapsulation of PolySPINner V0, leaving Kaplan's parsers and data structures intact, but organizing the application into distinct, generic components. Generalized parser, matcher, and code generator components were created, and a mechanism was put in place to instantiate these generic components for C++ and CLOS, along with hooks for future languages.

The generic matcher component was augmented to support several kinds of relaxed matches, such as subset/superset match and intersection match, and a library of functions for support of relaxed matching was initiated. PolySPINner V1 was described in Chapter 5.

9.1.3 Version Two

PolySPINner V2 was a significant revision of PolySPINner V1. Both the C++ and CLOS parsers were rewritten from scratch using Flex and Bison (Kaplan used hand-coded, recursive descent parsers) and augmented to track the inheritance relationships among types. The intermediate representation used by PolySPINner, described in Section 9.3.1, was enhanced to support language-specific features such as pointers, visibility (e.g., public, private, and protected members), and virtual methods. This was done by treating the intermediate representation as a base class and creating specialized subclasses for each target language, as will be explained in Section 9.3.1.

A rudimentary but interactive user interface was developed to demonstrate PolySPINner at the 1997 ACM International Conference on Software Engineering (ICSE) in Boston. The interface allowed the user to match methods and parameters interactively, but it did not enforce type safety. This interface also foreshadowed the development of the Missing Method Manager described in Section 7.4.1.

Extensive monitoring features were built into the code to aid in debugging, including several levels of diagnostic messages, and hooks to induce Flex and Bison to produce additional diagnostics, all controlled by command-line flags.

9.1.4 Version Three

PolySPINner V3 is the current version as of this writing. It has two main extensions beyond V2. The first is an emphasis on type safety. This includes the implementation of type-safe relaxed matching criteria given in Chapter 6, and a prototype Missing Method Manager as described in Chapter 7.

The second extension is support for interoperability among non-persistent (transient) objects. This will be described extensively in Chapter 10, as its development raised many new issues about the nature of polylingual systems.

9.2 Specification

PolySPINner is specified as a black box with the following inputs:

1. A set of n types, each written in a distinct language, that are intended to be compatible.[1]

2. A compatibility criterion, specifying the definition of "compatibility" among the input types. The user may select from a library of previously created criteria, or create a new one from scratch.

The output is a set of n types with the same interfaces as those of the input types, but with their method bodies instrumented to permit polylingual access. Functionally, PolySPINner would be represented as:

[1] The user interface of the current version restricts n to be 2.

$$\texttt{PolySPINner} : (T^n \times K) \rightarrow T^n$$

where T is the set of types, T^n is the set of sequences of n types, and K is the set of compatibility criteria.

9.3 Design

9.3.1 Intermediate Representation

PolySPINner represents classes, methods, and parameters internally as objects. It uses an intermediate representation built on three base types: `Class`, `Method`, and `Parameter`. For example, type `Method` is defined as:

```
class Method {
public:
  String& Name();             // Name of method
  int NumberOfParameters();   // Number of parameters
  Parameter *Retrieve(int);   // Return the ith parameter
  // ...and so on...
};
```

Types `Class`, `Method`, and `Parameter` are base types that represent features common to all object-oriented languages supported by PolySPINner. If a language has additional features to be represented, type `Class` is specialized by the interoperability designer through subtyping. For instance, CLOS methods may be defined within a `defclass` as accessors for slots, or outside the `defclass` function as regular methods using `defmethod`. These possibilities are reflected in the definition of type `CLOS_Method` ("CLOS method"):

```
class CLOS_Method : Method {      // A CLOS method
  Boolean SlotSpec();   // Am I a defmethod or a slot accessor?
  // ...and so on...
};
```

Unlike an IDL, PolySPINner's intermediate representation is largely transparent to the interoperability engineer. It is visible only if the engineer must design new criteria for type compatibility. For example, PolySPINner's default C++-to-CLOS mapping specifies that only the public methods of a C++ type may be matched with CLOS methods, which are all public. Should the interoperability engineer wish to change this behavior, a new compatibility criterion would be designed using the types of the intermediate representation, e.g.:

```
MatchResult MyCustomMatch(CPP_Method *m1, CLOS_Method *m2) {
  if (m2->SlotSpec()) ...
}
```

PolySPINner's intermediate representation originated with PolySPINner V0 but has been considerably modified and extended. Detailed definitions of types `Class`, `Method`, and `Parameter` appear in Appendix B.

9.3.2 Reliance on Run-time Systems

In a polylingual system, the invocation of one method m may transparently be transformed into an invocation of another method m' in another language, $L(m) \neq L(m')$. When the target language $L(m')$ supports polymorphism, the issue of selecting the most applicable method m' leads to an interesting design tradeoff for PolySPINner. On one hand, PolySPINner could calculate the most applicable method m' that should be invoked. This means PolySPINner would trace inheritance hierarchies, resolve late binding issues, trace `:before`, `:after`, and `:around` specifiers (in CLOS),

and make other, similar computations. This would allow an interoperability engineer great control over the matching of methods. These computations, however, can be very difficult and error-prone in practice, may sacrifice the benefits of dynamic binding, and may involve trying to second-guess the compilers involved.

On the other hand, PolySPINner could leave the choice of the most applicable method up to each language's run-time system. This would avoid the need for PolySPINner to attempt to duplicate the exact procedure each language's run-time system uses to dispatch methods. It is also upwardly compatible in the event of changes in the compilers or language specifications. This technique sacrifices precision, however, which an interoperability engineer might desire when matching methods.

The current implementation of PolySPINner opts for the latter choice: each language's run-time system is responsible for locating the most applicable method. When matching a C++ method m with a CLOS method m', for example, PolySPINner matches method m with the generic function underlying m', letting the CLOS run-time system select the most applicable method m'.

9.4 Implementation

PolySPINner is implemented mostly in C++, with some components implemented using Flex and Bison. This section describes the implementation of PolySPINner's components, leaving the most specific details to Appendix B.

9.4.1 Generic Architecture

Through the use of subtyping and C++ templates, PolySPINner has a generic architecture, as shown in Figure 9.1, a reproduction of Figure 5.5. The Parser, Matcher, and Generator components may be instantiated to provide specialized behavior for particular programming languages.

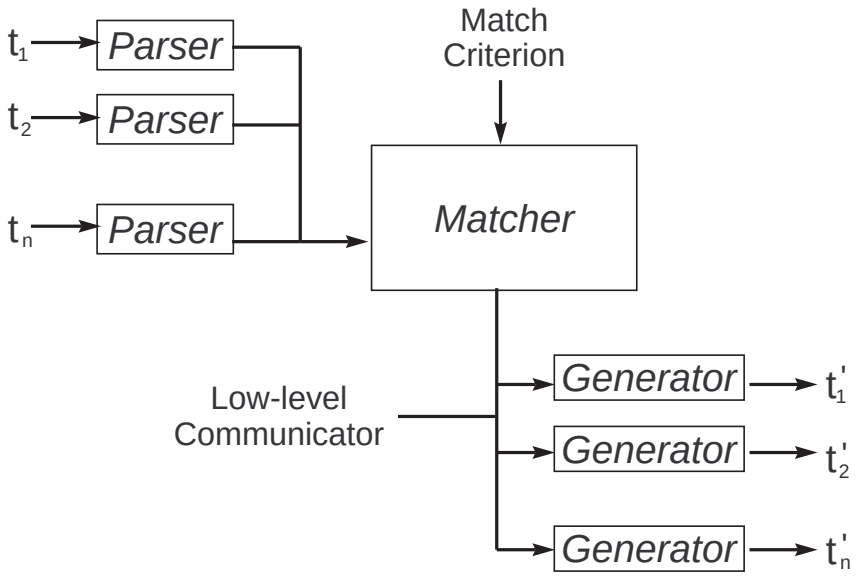


Figure 9.1. The PolySPINner architecture.

In order for PolySPINner to support a language, the interoperability designer supplies language-specific components to instantiate PolySPINner for that language. This instantiation is done only once per language, not once per application.

9.4.2 Parsers

PolySPINner's generic Parser component is a C++ template, intended to be instantiated with language-specific parsers written in C++. The existing C++ and CLOS parsers are implemented in C++ using Flex and Bison, as is a prototype Java parser now in progress. Flex and Bison were chosen because their scanners and parsers are usually easier to write, debug, and maintain than hand-crafted ones.

9.4.3 Matcher

The Matcher is part of PolySPINner proper and is implemented in C++. It consists of:

- The generic matching engine.

```
class MatchResult {
public:
  // Add a successful method match to the results.
  void AddMatch(Method *m1, int pos1, Method *m2, int pos2);

  // Add a failed method match, along with an explanation.
  void AddMismatch(Method *m1, int pos1, Method *m2, int pos2,
                   String why);
};
```

Figure 9.2. Type `MatchResult`, abbreviated.

- A class `MatchResult`, introduced in Chapter 6, that represents the state of a successful or unsuccessful match.
- A library of match criteria.
- A library of support functions to aid the interoperability engineer in designing new match criteria.

The generic matching engine does little more than iteratively apply the match criteria to the input types. The engine is instantiated by passing it a pointer to a compatibility criterion, implemented as a C++ function.

The `MatchResult` class, shown in abbreviated form in Figure 9.2, stores the result of applying matching criteria to pairs of types. The original PolySPINner V0 used a Boolean value to represent the overall success or failure of a match of two types. Given two types, the current PolySPINner uses the `MatchResult` class to represent:

- The overall success or failure of a match of two types.
- The success or failure of each attempted method match within the types.
- The success or failure of each attempted parameter match within each method of the two types.

- Reasons, in English text, why any matches failed.

As PolySPINner is extended, it is likely that the `MatchResult` type will be extended as well, to encapsulate a richer set of matching relationships. A more complete definition of `MatchResult` is given in Appendix B.

9.4.4 Generators

Generators are the generic components that produce PolySPINner's output. Once the input types have been processed by a Matcher, Generators produce the modified method bodies that permit those types to be polylingually accessible. Generic generators are instantiated twice: once to specify a target programming language, and once to specify the communication medium to be used for interlanguage method calls, such as message-passing or RPC. For example, the original `Generator` type, when specialized for C++, becomes `CPP_Generator`, and when specialized to use the Open Object-Oriented Database for communication, becomes the `CPP_OODB_Generator` type. The internals of Generators are discussed in detail in Appendix B.

9.4.5 Interlanguage Communication

In PolySPINner V0 through V2, interlanguage communication was implemented on top of low-level foreign function calls. PolySPINner is not dependent on this mechanism, however, and PolySPINner V3 was hosted instead on top of remote procedure calls, as will be discussed in detail in Chapter 10. In theory, other communication mechanisms could be substituted, such as a CORBA ORB.

Additionally, PolySPINner V0 through V2 made heavy use of the TI/ARPA Open Object-Oriented Database (Open OODB) [72] to provide a means of storing and retrieving objects implemented in different languages. The Open OODB has APIs for managing both C++ and CLOS objects. PolySPINner V3, as will be shown in Chapter 10, was freed of the Open OODB and generates its own server programs for managing objects.

9.5 Assumptions and Limitations

PolySPINner does not support all features of all programming languages for which it is currently instantiated. Some features are not covered for theoretical reasons, and others for practical reasons. This section covers the language features not supported.

9.5.1 C++ Language Binding

PolySPINner's C++ parser is cognizant only of classes, methods, and parameters. It skips much of the other code found in a C++ file, such as external variable declarations and preprocessor directives. As will be seen in Chapter 11, this limitation causes small technical problems (unrelated to interoperability) in several experiments because the Generator cannot reproduce this code on output. A full C++ parser would eliminate this issue.

C++ templates are not supported, in the sense that the Matcher cannot compare them with other types. It is not at all clear how to "match" templates with regular types, or with similar constructs in other languages, such as CLOS macros or Ada generics. This would make an interesting area for future theoretical work.

Exceptions are not supported, in the sense that methods are compared by the Matcher without taking their exceptions into account. Two methods with compatible signatures may have incompatible behavior if their exception behavior is different. As with templates, this is left as an area for future research.

9.5.2 CLOS Language Binding

The most significant assumption made by PolySPINner with respect to CLOS is that it must be told the return types of CLOS methods. In CLOS, `defmethod` does not specify the return type of the method it defines. In order to be PolySPINner-compatible, a `defmethod` must include the following code immediately after the parameter declarations:

```
(declare (return-values TYPE))
```

where `TYPE` is the return type of the method. While it may be possible in some cases to infer the return type of a method via static analysis, it is not possible in general due to the immense flexibility of Lisp, since the return type might not be determined until run-time. This particular solution to the problem is intrusive, but it need not be: the interoperability engineer could provide a separate file specifying the return types of methods, for example. (This implementation decision is left over from PolySPINner V0.)

PolySPINner's CLOS parser is cognizant of `defclass` and `defmethod` functions only. It ignores `defun`, `defgeneric`, `defmacro`, and other means of constructing Lisp functions. In addition, it does not support method names that are evaluated at run-time, such as in:

```
(defmethod (setf name) ...)
```

9.6 Summary

PolySPINner is a collection of generic Parsers, Matchers, and Generators that automate the development of polylingual systems. These subcomponents share information via a language-neutral, object-oriented, intermediate representation, that may be specialized to track language-specific features.

Although PolySPINner does not support every feature of the languages for which it is instantiated, it is sufficiently powerful to provide interoperability support for software systems. Chapters 10 and 11 will show sample applications of PolySPINner and the code it generates, permitting local method calls to be transformed seamlessly into interlanguage calls.

CHAPTER 10

TOWARD TRANSIENCE

Chapter 9 described the general architecture of PolySPINner as it has existed since PolySPINner V1. This chapter describes PolySPINner V3 and its support for interoperability among transient objects. It contains significant examples of PolySPINner's generated output, and it introduces the "address book" types to be used in the experiments of Chapter 11.

10.1 PolySPIN and Persistence

An underlying assumption of PolySPIN since its inception has been that objects are *persistent*: that is, they survive beyond the termination of the software component that created them.[1] Familiar examples of persistence include disk files created by an application, or tuples stored in a relational database. In PolySPIN, abstract objects persist, much as they do in an object-oriented database. For example, a polylingual program could create a persistent instance of type `House`, from Chapter 6, and then terminate, leaving the object accessible by other programs later.

Although PolySPIN's conceptual framework in Figure 5.3 does not mention persistence, the influence of persistence was pervasive in PolySPINner, its implementation. This is not surprising, as PolySPIN grew out of previous work with the Open Object-Oriented Database (Open OODB) [72], and PolySPINner was implemented on top of the Open OODB. What *is* surprising, however, is that an unstated assumption of

[1] In fact, the 'P' in "SPIN" stands for "Persistence": Support for Persistence, Interoperability and Naming.

persistence appears to have influenced the conceptual framework as well. For example, the framework assumes that objects are somehow always accessible. This may be true if the objects persist in a database, maintained by a dedicated, multithreaded database server, but it is not true in all complex software systems. Another assumption is that objects have names. This is natural enough if one expects to retrieve objects from a persistent store, but highly unusual for the diverse, non-persistent objects found in a typical application.

This chapter describes an experiment to separate PolySPIN from persistence, adding support for polylingual interoperability among *transient* (non-persistent) objects. This experiment is only a first step toward transience, and this chapter should be seen primarily as an elucidation of issues. The proposed solutions themselves are not necessarily novel. It is more important to see how this initial experiment has stretched or violated inherent assumptions of PolySPIN and PolySPINner. It raises numerous new questions about the nature of polylingual interoperability, but leaves many open for future work. As part of this experiment, a new implementation of PolySPINner was created by removing the dependency on the Open OODB and rehosting PolySPINner on top of Sun ToolTalk [39], a remote procedure call (RPC) mechanism with broadcast capability.

Pertinent issues will be discussed from three perspectives:

- *Data*: Issues of object naming.
- *Control*: Issues of servers, threads, deadlock, and termination.
- *Distribution*: As a side-benefit of the port to ToolTalk, PolySPINner gained the ability to generate distributed polylingual systems, with software components running on multiple computers or in multiple address spaces.

This chapter will use a running example of an address book application, using the C++ and CLOS types of Figure 10.1. These types will appear again in the experiments

```
// C++
class Phonebook {
  void Insert(char *, char *);  // Insert name and number
  char *Retrieve(char *);       // Given name, retrieve number
  void Print();                 // Print book
  void Clear();                 // Erase book
};


;; CLOS
(defclass address-book () ...)

;; Insert number and name
(defmethod add ((book address-book) number name)
  (declare (return-values nil)) ...)

;; Given name, retrieve number
(defmethod lookup ((book address-book) name)
  (declare (return-values string)) ...)

;; Print book
(defmethod print-book ((book address-book))
  (declare (return-values nil)) ...)

;; Erase book
(defmethod erase ((book address-book))
  (declare (return-values nil)) ...)
```

Figure 10.1. Types from a polylingual address book application.

of Chapter 11. Many other code fragments will be shown in this chapter, but it is important to remember that PolySPINner-generated code is transparent to the interoperating applications and, unless otherwise noted, invisible to the interoperability engineer.

10.2 The Data Perspective

In a polylingual system, objects must be identifiable or else they cannot be accessed. Thus every object must have a unique identifier, which is often called a *name*

```
class NameableObject {
  void Bind(Name);           // Assign my name
  void Publish();            // Announce my name publicly
  Name MyName();             // What's my name?
  ProgrammingLanguage& Language();  // What's my language?
  int& Remote();             // Am I real or surrogate?
};
```

Figure 10.2. The `NameableObject` type of PolySPINner V3. An analogous type is defined in CLOS.

or OID (object identifier). Given an object name, PolySPIN's Locator component fetches the associated object.

When objects are persistent, the idea of assigning names to them comes naturally because we must refer to those objects later, just as we name files for later use. Transient objects in a computer program, on the other hand, are frequently "named" only by their memory addresses, or by local variables that contain or point to the objects. Such "names" often are not visible outside the software components that create the objects; thus, an external component might not be able to access objects via those names.

The naming mechanism for persistent objects within PolySPIN V0 was the subject of Kaplan's doctoral dissertation [40]. To make the instances of a type t polylingually interoperable, PolySPIN introduced a new supertype, `NameableObject`, and caused type t to inherit from it. Type `NameableObject` had methods for assigning a name to an object and retrieving that name. Its public interface was slightly tied to persistence and the naming mechanism,[2] so a slightly more generic interface was developed for PolySPINner V3, shown in Figure 10.2.

[2]It required the type `TID`, or Transient Identifier, a component of Kaplan's name management interface.

The `Bind` operation assigns a name to an object, and `Publish` exports the name to be visible to other components. The next three operations—`MyName`, `Language`, and `Remote`—return the object's name, programming language, and whether it is a local or remote (surrogate) object. This new interface is intended to be general enough that the original PolySPINner naming functionality can be encapsulated within it, so an interoperability engineer may choose to support either persistent or transient objects or both.

`NameableObject` is an abstract type. An application never creates an instance of it, and in fact, application designers and the interoperability engineer need not even be aware that the type exists. PolySPINner transparently modifies a given type t so that it inherits from `NameableObject`. The interoperability engineer never codes a single call to a `NameableObject` method: all calls are generated automatically. The interoperability designer, however, must create a subtype of `NameableObject` specialized for a particular Locator mechanism. In PolySPINner V3, the Locator is ToolTalk, so a type `tt_NameableObject` has been created. It has one additional method,

```
virtual int _PolySPINner_Dispatch(Tt_message);
```

which serves as a message dispatching function, translating the ID of a remote procedure call into a real call of the corresponding method, as will be explained in more detail in Section 10.4. The body of this method is generated for each interoperable type by PolySPINner, as will be shown later in Figure 10.6.

Once names are assigned to objects by `Bind`, and the names are announced by `Publish`, potential accessors can retrieve objects by querying a nameserver, an object of type `NameServer`, shown in Figure 10.3. The `Register` and `Unregister` methods allow an application to use (and cease using) the nameserver. For simplicity of implementation, a polylingual system has one global namespace, and every instance of `NameServer` serves this space.

```
class NameServer {
  void Register();
  void Unregister();
  NameableObject *Fetch(Name);
};
```

Figure 10.3. The `NameServer` type of PolySPINner V3. An analogous type is defined in CLOS.

The `Fetch` operation retrieves a given object by name. If the object is remote, `Fetch` retrieves a surrogate. Because `Fetch` has the return type `NameableObject*`, the return value must be coerced into the type of the desired object, e.g.,

```
object = (Phonebook *) MyNameServer.Fetch("my phonebook");
```

In summary, transient objects need names in order to be accessed across language boundaries. PolySPIN V3 continues to use the `NameableObject` abstraction of previous versions, slightly generalized, and a new `NameServer` type for fetching remote, transient objects in the absence of a database. PolySPINner V3 implements these types instantiated for use with ToolTalk.

10.3 The Control Perspective

The PolySPIN framework originally was not concerned with *control*: the path(s) of execution in a software system. PolySPIN had an unstated assumption that objects were always available to be retrieved and accessed. This assumption was never challenged because it was true in the implementation, PolySPINner, for two reasons:

- Object retrieval was centrally managed by the Open OODB.
- Interlanguage method invocation was centrally managed for all languages by the Lisp interpreter, using foreign function calls.

In the move to transience and ToolTalk, this reasoning no longer holds. Object retrieval is no longer centrally managed via a database, because transient objects exist only as long as the applications that created them, and therefore they cannot be kept in a database after those applications have terminated. Thus, access to transient objects must be managed by the applications that create them. However, an application does not necessarily provide an API for external access to its objects, whereas an OODB exists for the purpose of providing that API. In order for PolySPIN to support transient objects, it must work within this new constraint.

In addition, interlanguage method invocation can no longer be managed by the Lisp interpreter. If objects are to be managed by their containing applications, the objects will be executing in different address spaces. Thus, foreign function calls can no longer be used to communicate between components of distinct languages, and another communication substrate will be needed.

This section discusses some control-related modifications to PolySPIN and PolySPINner that were necessary in order to support polylingual interoperability between transient objects. These modifications are in the areas of:

- Serverization
- Threads
- Deadlock
- Termination

10.3.1 Serverization

As previously stated, the original PolySPINner used the Open OODB, which was effectively an object server, to fetch persistent objects on demand, as in Figure 10.4(a). For PolySPIN to support transient objects, the containing application must take the role of object server. In other words, in order for transient objects to be accessed in a

polylingual system, they must be created and contained within application programs that have been *intentionally constructed to permit* external access to those objects. This presents a problem, as PolySPIN is intended to be useful for megaprogramming, or after-the-fact interoperability. It is unlikely that independently-developed programs will be written to facilitate interoperability in this manner.

For interoperability to remain seamless, then, an implementation of PolySPIN must transparently create, modify, or replace the main programs that create or contain transient objects, or in some other way create a channel through which interoperability may take place with a legacy application. This can have a significant impact on the behavior of the affected programs. Many other interoperability approaches wrestle with this same issue, requiring the interoperability engineer to "serverize" main programs, or even individual objects, in order to permit external method calls to reach objects.

Thus, PolySPINner V3 not only modifies the implementations of methods as before, but also generates application code to manage calls to those methods, as in Figure 10.4(b). It generates a method dispatcher in the form of an event loop, shown in Figure 10.5, to manage access to objects. It intercepts method calls and forwards them to the appropriate object by invoking the object's `__PolySPIN_Dispatch` method, shown in Figure 10.6. The event loop may be used within an existing main program, or it may replace the main program, depending on the needs of the interoperability engineer. Either way, some hand-coding will be required to insert the event loop or replace the main program. PolySPINner V3 also generates a usable main program, shown in Figure 10.7, that invokes the event loop.

Generating this main program and event loop, however, is not possible without some additional input to PolySPINner. Originally, the input was a set of type definitions and a compatibility criterion, as shown in Figure 5.5. This provides information about the types used by a software component, but not the *instances* of those types.

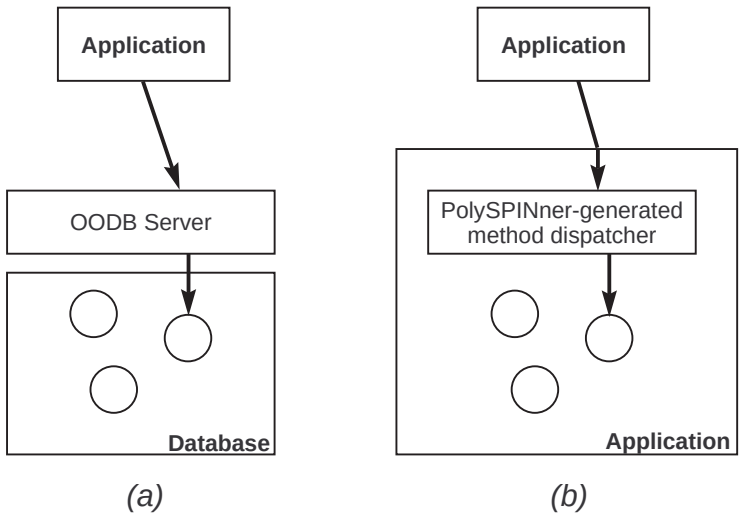


Figure 10.4. PolySPINner control structure, persistent version (a) and transient version (b).

In order to generate the event loop for an application, PolySPINner needs to know names for the objects to be made externally accessible. In addition, to generate the main program, PolySPINner needs the names, types, and languages of those objects. For the time being, this information is provided to PolySPINner by the interoperability engineer, via an object definition file created by hand, as shown in Figure 10.8. This solution is common in interoperability mechanisms (e.g., Polylith [60]). In the future, it may be possible to generate this object definition file by analyzing the source code of the legacy applications.

10.3.2 Threads of Control

The original PolySPIN assumed that objects were always available. This assumption implies the existence of not just an object server, but a *multithreaded* object server, or at the very least, one that can queue incoming accesses when an object is busy servicing another access.

```
while (1) {
 // Receive the next message.
  if ((msg = tt_message_receive()) == NULL)
    continue;

 // Unmarshal the ID of the invoked method.
  tt_message_arg_ival(msg, 0, &methodID);

 // Termination message received?  Then terminate.
  if (methodID == __POLYSPIN_DIE_DIE_DIE)
    return;

 // A language query from the NameServer?  Then respond.
  if (methodID == __POLYSPIN_WHATS_YOUR_LANG) {
    tt_message_arg_val_set(msg, 1, LANG_CPP);
    tt_message_reply(msg);
    continue;
  }

 // Otherwise, it's a remote method call.
 // Dispatch the method to the appropriate object.
 // Notice that no type-related details are present here;
 //  they are fully encapsulated within each object.
  if (ForMe(__POLYSPIN_Obj0->MyName(), msg))
    __POLYSPIN_Obj0->__POLYSPIN_Dispatch(msg);

  else if (ForMe(__POLYSPIN_Obj1->MyName(), msg))
    __POLYSPIN_Obj1->__POLYSPIN_Dispatch(msg);

  else if (ForMe(__POLYSPIN_Obj2->MyName(), msg))
    __POLYSPIN_Obj2->__POLYSPIN_Dispatch(msg);

  else
    fprintf(stderr, "Error - unknown object\n");
}
```

Figure 10.5. Event loop generated by PolySPINner V3. Objects must be created and assigned names before this function is invoked. Analogous code is generated in CLOS.

```
void Phonebook::__POLYSPIN_Dispatch(Tt_message msg) {
  MethodID mid;  /* method id */

 // Unmarshal unique method ID, and call appropriate method.
  tt_message_arg_ival(msg, 0, &mid);
  switch (mid) {
    case __POLYSPIN_CPP_Phonebook_Insert_6: {
     // Unmarshal parameters
      char *t1;
      t1 = tt_message_arg_val(msg, 2);
      char *t2;
      t2 = tt_message_arg_val(msg, 3);

     // Call the real method
      this->Insert(t1, t2);
    }
    break;

    case __POLYSPIN_CPP_Phonebook_Retrieve_7: {
     // Unmarshal parameters
      char *t1;
      t1 = tt_message_arg_val(msg, 2);

     // Call the real method
      char *retval;
      retval =
        this->Retrieve(t1);

     // Marshal return value
      tt_message_arg_val_set(msg, 1, retval);
      tt_message_reply(msg);
    }
    break;

    case __POLYSPIN_CPP_Phonebook_Print_8:
   // ...
    default:
      fprintf(stderr, "Error - unknown method id %d\n", mid);
  }
}
```

Figure 10.6. Dispatcher method generated by PolySPINner V3 for each modified type.

```
// The objects
Phonebook *__POLYSPIN_Obj0;
Phonebook *__POLYSPIN_Obj1;
Phonebook *__POLYSPIN_Obj2;

main() {
  // Beginning setting up ToolTalk
  char *procid   = tt_open();
  int ttfd       = tt_fd();

  // Create all objects
  __POLYSPIN_Obj0 = new Phonebook;
  __POLYSPIN_Obj1 = new Phonebook;
  __POLYSPIN_Obj2 = new Phonebook;

  // Set their languages
  __POLYSPIN_Obj0->Language() = strdup(LANG_CPP);
  __POLYSPIN_Obj1->Language() = strdup(LANG_CPP);
  __POLYSPIN_Obj2->Language() = strdup(LANG_CPP);

  // Register all objects
  __POLYSPIN_Obj0->Bind("book1");  __POLYSPIN_Obj0->Publish();
  __POLYSPIN_Obj1->Bind("book2");  __POLYSPIN_Obj1->Publish();
  __POLYSPIN_Obj2->Bind("book3");  __POLYSPIN_Obj2->Publish();

  // Sign up with ToolTalk
  tt_session_join(tt_default_session());

  // Run the event loop shown previously
  MainEventLoop();

  tt_close();
  exit(0);
}
```

Figure 10.7. Object declarations and initialization, and invocation of the event loop, in the main program generated by PolySPINner V3. Analogous code is generated in CLOS.

```
// NAME    TYPE            LANGUAGE
// --------------------------------
   book1   Phonebook       C++
   book2   Phonebook       C++
   book3   Phonebook       C++
   book4   address-book    CLOS
   book5   address-book    CLOS
```

Figure 10.8. Object definition file given as input to PolySPINner V3.

With the move to transience, objects are maintained by the programs that create them, and this assumption no longer holds. While it would not be difficult to generate a multithreaded version of the event loop of Figure 10.5, it is unlikely that a complex, single-threaded legacy application could be made multithreaded simply by augmenting or replacing its main program by a multithreaded event loop. More likely the program would need a significant change of algorithm. This is a special case of the general problem of parallelizing sequential programs, which is still an active area of research.

PolySPINner V3 does queue method calls automatically when their intended recipient is busy processing other method calls. This is a built-in feature of the ToolTalk server. As we shall see in the next section, however, this does not solve all control problems.

10.3.3 Deadlock

Suppose objects A and B are created by an ordinary, monolingual, single-threaded application program. The program invokes the method $A.m(B)$, as in Figure 10.9, passing object B as a parameter of a method of A. Suppose that the body of method m then invokes a method of its parameter, B. This is a typical case of nested method calls via a call stack. The execution of method m suspends while the method of object B is invoked, and when it returns, execution of method m resumes.

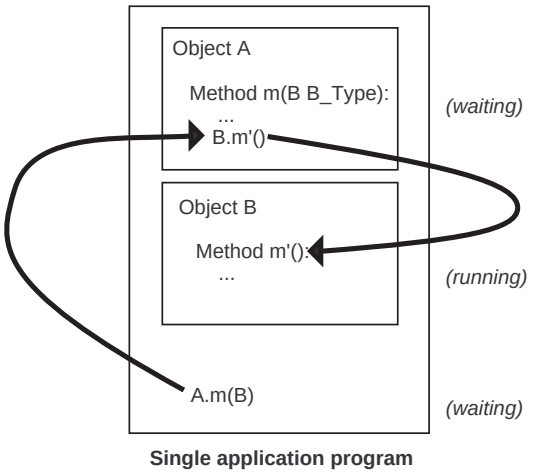


Figure 10.9. Ordinary nested method calls in an application program.

In the original PolySPINner, a similar scenario ensues if persistent objects A and B are of distinct languages, as in Figure 10.10. Suppose object A is written in CLOS, object B is written in C++, and the calling program is also in C++. Thus, object A, to the calling program, is actually a surrogate C++ object. When the method $A.m(B)$ is invoked, the call passes through surrogate A to reach the true (CLOS) A. When method m invokes a method of B, the call takes place within the object server. The original PolySPINner kept all these objects in a single address space and under a single thread of control, so all calls succeeded as in the monolingual case.

When transient objects enter the picture, managed by separate object servers, more dire consequences become possible, as shown in Figure 10.11. Suppose that the calling application and object B are not only both written in C++, but also that B is contained in the application. When the application invokes $A.m(B)$, the application suspends and waits for method m to return. The call passes through surrogate A to the true A, implemented in CLOS, which invokes a method of (surrogate) B, causing the application containing A to suspend. This method call goes through the surrogate

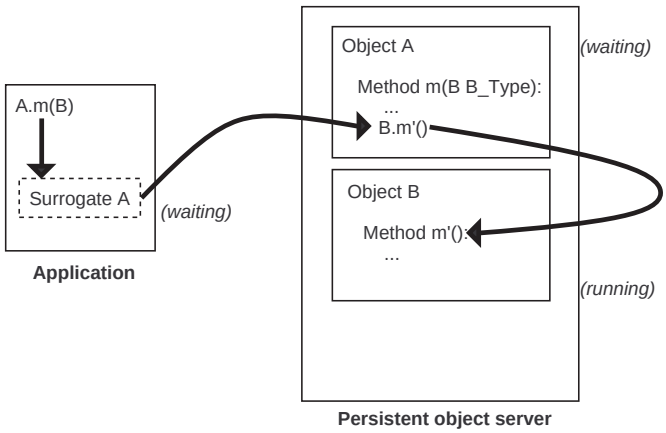


Figure 10.10. Polylingual application accessing persistent objects via an object server.

B and attempts to reach the true (C++) B. However, the original application, which is itself the server of B, is suspended! The two applications have reached a deadlock.

This problem can be addressed in several ways:

- Assign a separate server to each object, as in SoftBench [24], and queue incoming method calls, so the callee object is always available.
- Make all method calls asynchronous, as in CORBA [55], so the accessor does not suspend.
- Limit method parameters to be of primitive types only, as in ToolTalk, so the accessed object cannot invoke a method of the accessor through a parameter.

Each of these solutions could impose drastic changes on the original applications containing the objects, and thus they are unacceptable solutions for an approach geared toward megaprogramming. Currently, however, there is no universally better solution in practice. PolySPINner V3, in its current form, has adopted the third approach because it was easiest to implement. As PolySPINner V2 did not have this limitation, it would be desirable to eliminate it.

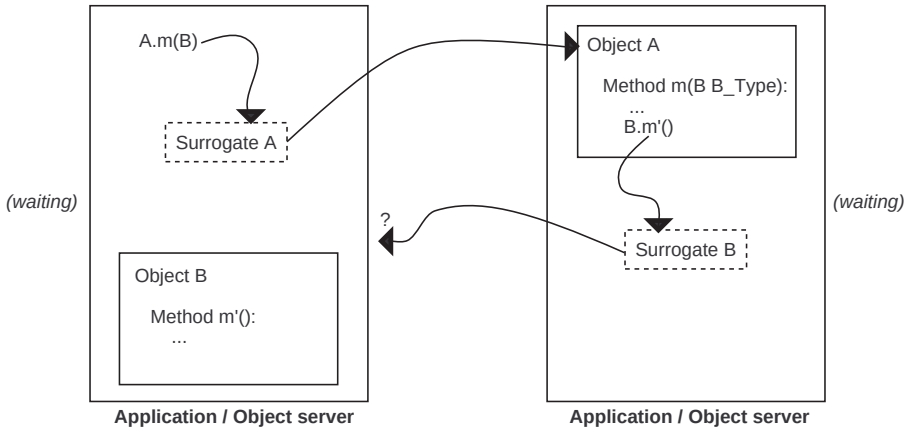


Figure 10.11. Independent object servers can lead to deadlock.

10.3.4 Termination

Under the original PolySPIN, a polylingual system had one thread of control, and when that thread ended, the system terminated. Now that PolySPIN consists of multiple object servers, each with a separate thread of control, it is no longer so simple to terminate the system.

To address this problem, PolySPINner V3 generates additional code in the event loop of every object server program. Before passing an incoming message to an object, the event loop checks whether it is a special termination message, as shown in Figure 10.5. If so, the event loop terminates.

Any application in the system may send this message and bring individual servers, or the whole system, to a halt. The PolySPINner toolset also includes a program, **killservers**, that iterates through all objects in the object definition file (Figure 10.8), sending the termination message to each.

10.4 The Distribution Perspective

The original PolySPIN's Locator component, described in Section 5.2.2, locates an object by its name. This abstract description does not specify whether the Locator covers objects in a single address space, in multiple address spaces, or on multiple computers. Thus, it was easy (at the time PolySPIN was conceived) to leap to the conclusion that the Locator would be independent of such concerns at the framework level, should PolySPIN ever be used for polylingual interoperability among distributed components. Since PolySPINner was implemented using foreign function calls, within a single address space, to communicate between accessors and objects, this assumption went unchallenged.

ToolTalk is a remote procedure call mechanism that, unlike foreign function calls, can communicate between components in different address spaces or on different computers. By hosting PolySPINner V3 on top of ToolTalk, PolySPINner gained the ability to generate distributed polylingual systems. Distribution is not inherently tied to transience, but the use of transient objects implies the need for one or more server programs to manage them, and the phrase "one or more server programs" implies distribution.

When PolySPIN became distributed, several unstated assumptions of polylingual systems were unexpectedly violated:

- The assumption that polylingual method calls have method-call semantics.
- The assumption that polylingual method calls are synchronous.
- The assumption that same-language method calls do not cross address spaces.

These assumptions will be discussed in the following sections.

10.4.1 Marshaling

The original PolySPINner used foreign function calls for interlanguage communication, and they have true method-call semantics of call and return, as shown in Figure 10.12(a). ToolTalk, however, uses remote procedure calls which, at their heart, are based on message passing in the manner of Sun RPC [70]. A remote procedure call is a message containing a method identifier (an integer ID) and a list of marshaled parameters. In order to simulate method-call semantics using ToolTalk, the technique of Figure 10.12(b) was used. The accessor marshals parameters, sends the message, and waits for a reply. The callee waits until it receives a message, unmarshals the parameters, looks up the target method in a table, calls the target method, marshals the return value, and sends a reply message back to the accessor. The accessor unmarshals the return value, and all is well. The trick is to make all of this effort transparent to the accessor, the object, and the interoperability engineer.

Figure 10.13 shows a typical marshal/unmarshal of a method call, minus some necessary error-checking code, as it is generated by PolySPINner. Upon invocation, the method checks whether its object is local or remote. If remote, it creates a message, creates a slot for a return value (if any), marshals the parameters (taking into account any relaxed matching at this point), and sends the message. If the method should return a value, it waits for a reply message, unmarshals the return value, and returns it.

PolySPINner V3 identifies methods by assigning them unique integer IDs tied to mnemonic names. For example, the value `__POLYSPIN_CLOS_address_book_lookup_33` identifies a method, `lookup`, in CLOS type `address-book`. (The "33" is a unique integer used to distinguish between overloaded methods if present.) In Figure 10.13, this ID represents the CLOS method compatible with the C++ `Retrieve` method.

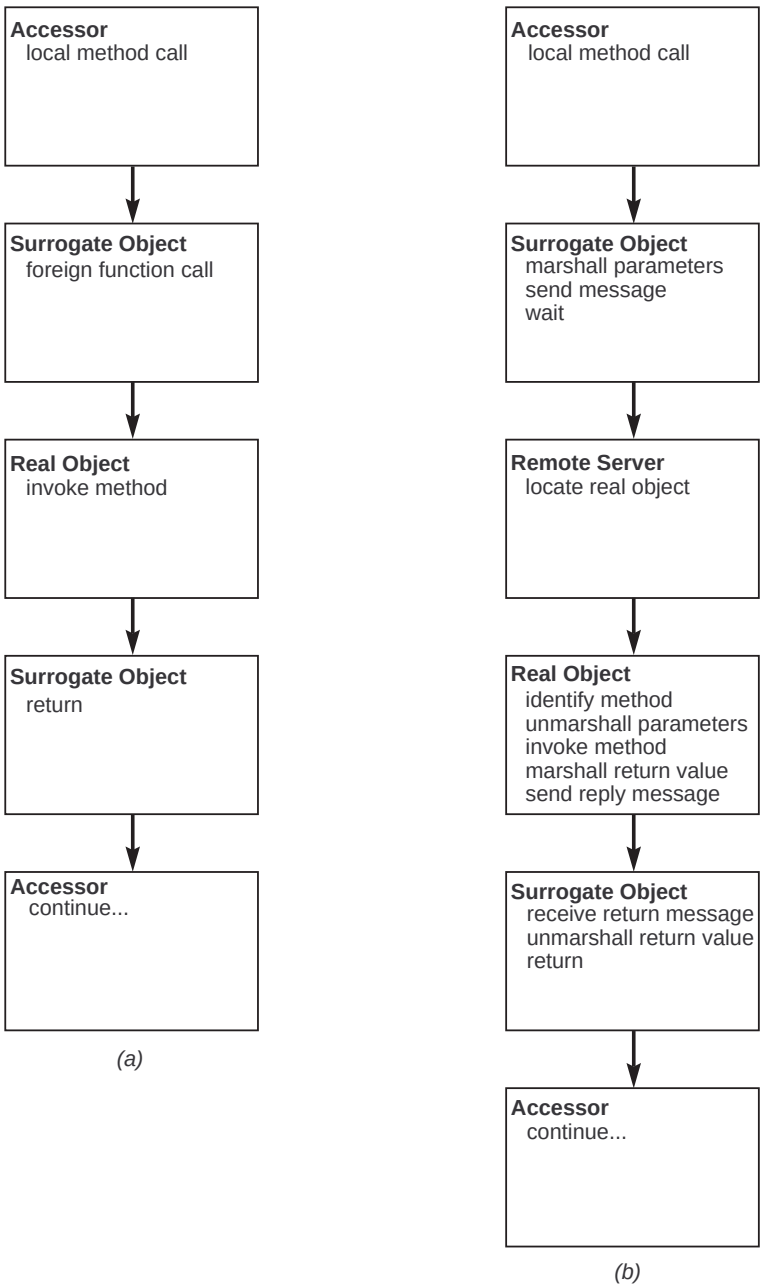


Figure 10.12. Implementation of a remote method call in original PolySPINner (a) and PolySPINner V3 (b).

```
char *Phonebook::Retrieve (char *name) {
  Tt_message msg, returnmsg;

  if (this->Remote() == 0) {
   // Original method body, verbatim.
  }
  else if (SameLanguage(this->Language(), LANG_CLOS)) {
   // Create message for the remote object named MyName().
    msg = tt_prequest_create(TT_SESSION, this->MyName());

   // Marshal a unique method identifier, generated by PolySPINner
    tt_message_iarg_add(msg, TT_IN, "int",
                        __POLYSPIN_CLOS_address_book_lookup_33);

   // Make slot for return value
    tt_message_arg_add(msg, TT_OUT, "char *", 0);

   // Marshal parameters, permuting/substituting if needed
    tt_message_arg_add(msg, TT_IN, "char *", name);
    tt_message_send(msg);

   // Get the return value
    returnmsg = WaitForReturn();
    char *returnvalue;
    returnvalue = tt_message_arg_val(returnmsg, 1);

    // Clean up and return
    tt_message_destroy(msg);
    return returnvalue;
  }
  else if (SameLanguage(this->Language(), LANG_CPP)) {
    // Similar logic as CLOS case, but different parameter marshaling
  }
  else {
    // This should never happen.
    return 0;
  }
}
```

Figure 10.13. Marshaling and unmarshaling in the ToolTalk-based PolySPINner V3.

10.4.2 Asynchrony

Polylingual method calls have been assumed to be synchronous. ToolTalk message passing, however, is asynchronous only. To address this difference, a synchronous message-passing construct was implemented for ToolTalk. It is simplistic: the accessor busy-waits until it receives a return message from the object. A more flexible and less CPU-intensive solution would be desirable, but this simple construct has sufficed for initial use of PolySPINner V3. The construct simulates method-call semantics and is completely transparent to the interoperability engineer and the interacting components, so it can conveniently be replaced when a better technique is implemented.

10.4.3 Same-Language Surrogates

Previously, PolySPIN's surrogate objects (Section 5.3) always represented an object of another language. As a result, the method bodies instrumented by PolySPINner had the first form shown in Figure 10.14. When invoked, an instrumented method would first determine whether its language matched its accessors'. If so, a local call was performed, and if not, an interlanguage call was performed. In PolySPIN V3, however, surrogate objects may represent any remote object, regardless of language. Thus, an instrumented method body now has the second form given in Figure 10.14. If a remote object is of the same language as the accessor, a remote method call is made with exact matching between the parameter lists. If the languages differ, relaxed matching is performed according to the user-supplied compatibility criterion.

There is no framework-level reason why PolySPIN cannot perform relaxed matches between types of the same language. This could be beneficial, in fact, as a means of providing monolingual interoperability or type evolution within a software system, permitting old and new versions of the same type to be considered compatible. Unfortunately, the current implementation of the Matcher in PolySPINner impedes this because it stores only one relaxed match, assuming it to be interlanguage.

```
// Original PolySPINner-generated method body

  if (the accessor's language == my language)
    original method body
  else
    remote invocation with relaxed, interlanguage match


// PolySPINner V3 generated method body:

  if (I am a surrogate object)
    if (the accessor's language == my language)
      remote invocation with exact, single-language match
    else
      remote invocation with relaxed, interlanguage match
  else  /* local object */
    original method body
```

Figure 10.14. Pseudocode illustrating the body of a method after being instrumented by the original PolySPINner (top) and PolySPINner V3 (bottom).

This aspect of PolySPINner has potential for optimization. If some accessors and objects can exist in the same address space, PolySPINner could potentially generate foreign function calls, rather than RPCs, between them and gain efficiency.

10.5 Other Implementation Details

ToolTalk version 1.2 is a message-passing mechanism for C and C++. In order for it to support Lisp, I created a ToolTalk API to support all ToolTalk calls required by PolySPINner. This was accomplished by wrapping the original ToolTalk C functions in an additional layer of software, so that they are accessible by foreign function calls from Lisp.

10.6 Summary

By moving PolySPIN's focus from persistent objects in a database to transient objects maintained by programs, numerous assumptions of PolySPIN have needed to be reevaluated:

- The naturalness of naming objects.
- The existence of an implicit, multithreaded (or queuing) object server.
- The issue of deadlock.
- The issue of system termination.

The support for transient objects in PolySPIN was only a preliminary experiment. In order for transient objects and distribution to be supported more fully by PolySPINner, more needs to be done:

- Automatic generation of the object definition file, if possible.
- Removal of busy-waiting in the implementation of synchronous remote method calls.
- An improved solution to the deadlock problem.
- Implementation of same-language relaxed matching.
- Optimization of same-language method calls.

Chapter 11 will demonstrate the use of PolySPINner V3 in its current form.

CHAPTER 11

EXPERIMENTAL RESULTS

11.1 Overview

This chapter describes three experiments in which PolySPINner V3 was applied to independent software components to make them interoperate. Each experiment corresponds to one of the three interoperability scenarios defined in Section 5.2.1:

1. *Easiest case* interoperability, in which the decision to interoperate is made prior to building the applications. In this experiment, two small, standalone "telephone directory" applications were built from scratch, one in C++ and one in CLOS, and made to interoperate polylingually.

2. *Common case* interoperability, in which newly written applications are integrated with legacy applications. In this experiment, the two telephone directory applications above were made to interoperate polylingually with CL-HTTP [51], a CLOS-based web server from the Massachusetts Institute of Technology, creating a web-based telephone directory.

3. *Megaprogramming*, in which two legacy applications are made to interoperate. In this experiment, CL-HTTP was made to interoperate polylingually with a regular expression class from the GNU C++ class library [44], giving CL-HTTP a new capability to perform string-matching with regular expressions.

In addition, in each of these experiments, at least one of the applications was executed on a different computer from the others, making each software system distributed.

Code fragments from each experiment are presented. The PolySPINner-generated, ToolTalk-based code is of the same form as the fragments of Chapter 10. Some code fragments have been cleaned up for presentation purposes: changing spacing and indenting, and removing non-illustrative code such as error checking and class constructors/destructors. In addition, Section 11.6 explains the need for some small modifications to the generated source due solely to inadequacies of PolySPINner V3's C++ parser.

As a final note, we present some performance results. The speed of PolySPINner's generated code is compared to that of native ToolTalk function calls. As will be shown, the overhead imposed by PolySPINner is quite small.

11.2 Testbed

All experiments were conducted on two Sun workstations—one SPARCstation 5 and one SPARCstation 10—in the Convergent Computing Systems Lab at the University of Massachusetts. Both workstations ran the SunOS 5.5 (Solaris) operating system and were connected via Ethernet.

11.3 Scenario One: Easiest Case Interoperability

In this experiment, two application programs were created: online telephone directories that store names and phone numbers for later retrieval. One program was implemented in C++ and the other in CLOS. The decision for these programs to interoperate was made before the programs were written. However, each program was constructed as a standalone application and contains no interoperability-related code. The leverage gained by the foreknowledge of interoperability is that the programs' types were constructed to be similar.

The result of this experiment, depicted in Figure 11.1, is that both telephone directories can be accessed seamlessly from both languages. Neither accessor is aware

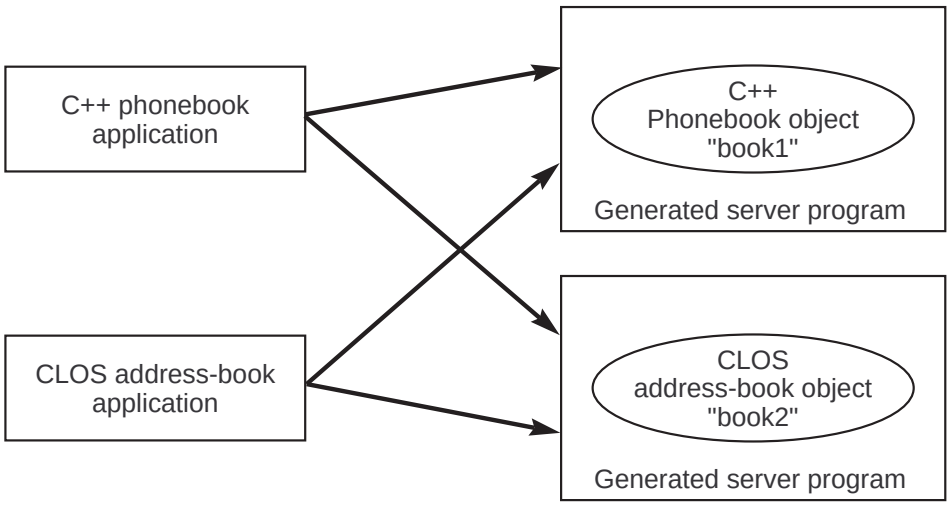


Figure 11.1. Easiest case: Polylingual interoperability between telephone directory applications.

of language differences when invoking methods to access the other's data. Each application works with instances of its *locally defined* telephone directory type, called `Phonebook` in C++ and `address-book` in CLOS, but actually accesses directories of both languages, as in Figures 11.2 and 11.3. The only visible change to the main programs is the fetching of the remote directory via a nameserver. As stated in the assumptions of Section 1.4, any approach to interoperability must have some means of locating objects, so such a modification is necessary to turn a standalone application into an interoperating one.

This experiment also demonstrates relaxed matching between methods. Two of the methods matched have their parameters in a different order, and PolySPINner generates the appropriate marshaling code to reverse them before sending them across the language barrier.

```
main(int argc, char *argv[]) {
  NameServer ns;
  Phonebook *book;

  // Language-transparent fetch by nameserver
  book = (Phonebook *)ns.Fetch(argv[1]);

  // Polylingual method calls
  book->Insert(argv[2], argv[3]);
  printf("Inserted %s", book->Retrieve(argv[2]));
  book->Print();
}
```

Figure 11.2. Main program of C++ telephone directory application, illustrating seamless interoperability.

```
(defun insert (bookname phonenumber person)
  (setq ns (make-instance 'NameServer))
  (Register ns)

  ;; Language-transparent fetch by nameserver
  (setq book (change-class (Fetch ns bookname) 'address-book))

  ;; Polylingual method calls
  (add book phonenumber person)
  (format t "Inserted ~A~%" (lookup book person))
  (print-book book)

  (Unregister ns))
```

Figure 11.3. Main program of CLOS telephone directory application, illustrating seamless interoperability.

```
// C++ class
class Phonebook {
private:
  char **names, **numbers;  // Lists of names and phone numbers
  int numentries, max;      // Current and maximum size of directory
public:
  void Insert(char *name, char *number);
  char *Retrieve(char *name);
  void Print();
  void Clear();
};

void Phonebook::Insert(char *name, char *number) {
  if (numentries < max) {
    names[numentries]   = strdup(name);
    numbers[numentries] = strdup(number);
    numentries++;
  }
}
```

Figure 11.4. The C++ `Phonebook` class, with its `Insert` method shown.

11.3.1 The C++ `Phonebook` Type

The C++ telephone directory application is based on the type `Phonebook`, shown in Figure 11.4, that encapsulates a list of names and phone numbers. Public methods supported on `Phonebook` objects include insertion of a name and phone number into the directory (`Insert`), retrieval of a phone number given a name (`Retrieve`), printing the directory (`Print`), and erasing the contents of the directory (`Clear`).

11.3.2 The CLOS `address-book` Type

The CLOS telephone directory application is based on the type `address-book`, shown in Figure 11.5, that similarly encapsulates a list of names and phone numbers. The methods supported on `address-book` objects correspond one-to-one with with those of the C++ `Phonebook` type, including insertion of a name and phone number into the directory (`add`), retrieval of a phone number given a name (`lookup`), printing

```
(defclass address-book ()
  ((entry-list :accessor entry-list
               :initarg :entry-list
               :initform nil)))

(defmethod add ((book address-book) number name)
  (declare (return-values nil))
  (setf (entry-list book)
        (cons (list name number)
              (entry-list book))))

(defmethod lookup ((book address-book) name)
  (declare (return-values string)) ...)

(defmethod print-book ((book address-book))
  (declare (return-values nil)) ...)

(defmethod erase ((book address-book))
  (declare (return-values nil)) ...)
```

Figure 11.5. The CLOS `address-book` class, with its `add` method shown in full.

the directory (`print-book`), and erasing the contents of the directory (`erase`). The signatures of most methods are identical to their C++ counterparts except for `add`, which has its parameters in the reverse order when compared to the C++ `Insert` method. This presents an opportunity for PolySPINner to implement a relaxed match of `Insert` and `add`.

11.3.3 Integrating `Phonebook` and `address-book`

PolySPINner V3 was applied to the `Phonebook` and `address-book` types, generating the modified type declarations shown in Figure 11.6. This was accomplished using PolySPINner's interactive type-matching interface, known as "arbitrary match" (Section C.5.3), that prompts the user for which methods and parameters to match with one another. It was at this time that the permuted parameters of the C++ `Insert`

```
// C++
class Phonebook : public tt_NameableObject {
private:
  char **names, **numbers;  // Lists of names and phone numbers
  int numentries, max;      // Current and maximum size of directory
public:
  void Insert(char *name, char *number);
  char *Retrieve(char *name);
  void Print();
  void Clear();

  // PolySPINner-generated methods
  void __POLYSPIN_Dispatch(Tt_message);
};


;; CLOS
(defclass address-book (tt-NameableObject)
  ((entry-list :accessor entry-list
               :initarg :entry-list
               :initform nil)))
```

Figure 11.6. The C++ `Phonebook` type and CLOS `address-book` type after processing by PolySPINner V3.

method and the CLOS `add` method were indicated to PolySPINner, so it would know to generate appropriate marshaling code to reconcile the difference.

On completion, PolySPINner generated five new files: a modified `.C` and `.h` file for type `Phonebook`, a modified `.lisp` file for type `address-book`, and two standalone server programs, one in C++ and one in CLOS, for managing objects of the two types. The modified type files were identical to the originals in Figures 11.4 and 11.5 except that they now inherited from a subtype of `NameableObject` specialized for ToolTalk. The C++ type also had a new method, `__POLYSPIN_Dispatch`, for use in the PolySPINner-generated event loop as described in Section 10.3.1. (Its body appears in Figure 10.6.)

The method bodies of the two classes were instrumented by PolySPINner in the same manner as explained in Chapter 10. The generated method body first checks whether the object is real or a surrogate. If real, the original body is executed unchanged. Otherwise, the parameters are marshaled in a manner appropriate to the language of the surrogate. Figures 11.7 and 11.8 show the resulting, generated code for the C++ method `Insert` and its corresponding CLOS method, `add`, including the marshaling for the permuted parameters before they are sent across the language boundary. (Compare the marshaling order of `name` and `number` in either figure.)

The generated files were then compiled[1] and linked. The end result is that both main programs shown in Figures 11.2 and 11.3 seamlessly access objects of types `Phonebook` and `address-book` as if they were instances of the local type for telephone directories. Thus, the two applications have achieved polylingual interoperability in the easiest case scenario.

11.4 Scenario Two: Common Case Interoperability

In this experiment, the telephone directory applications from Section 11.3 were integrated with a legacy application, the Common Lisp Hypermedia Server, known as CL-HTTP [22, 51]. The result of this experiment, depicted in Figure 11.9, is a web page, generated by CL-HTTP, that permits the user to query either of two telephone directories. The two directories are implemented in distinct languages (C++ and CLOS), and the web page accesses either directory, polylingually, based on user input.

11.4.1 CL-HTTP

CL-HTTP, the Common Lisp Hypermedia Server, is a World Wide Web server consisting of approximately 30,000 lines (1.2 megabytes) of CLOS. It was implemented

[1] After the slight hand-editing described in Section 11.6.

```
void Phonebook::Insert (char *name, char *number) {
  Tt_message msg;
  if (this->Remote() == 0) {
    // The original method body.
    if (numentries < max) {
      names[numentries] = strdup(name);
      numbers[numentries] = strdup(number);
      numentries++;
    }
  }

  // Execute if I am a surrogate for a CLOS object.
  else if (SameLanguage(this->Language(), LANG_CLOS)) {
    msg = tt_pnotice_create(TT_SESSION, this->MyName());
    tt_message_iarg_add(msg, TT_IN, "int",
                        __POLYSPIN_CLOS_address_book_add_16);
    tt_message_iarg_add(msg, TT_OUT, "DUMMY", 0);
    tt_message_arg_add(msg, TT_IN, "char *", number);
    tt_message_arg_add(msg, TT_IN, "char *", name);
    tt_message_send(msg);
    tt_message_destroy(msg);
  }

  // Execute if I am a surrogate for a C++ object.
  else if (SameLanguage(this->Language(), LANG_CPP)) {
    msg = tt_pnotice_create(TT_SESSION, this->MyName());
    tt_message_iarg_add(msg, TT_IN, "int",
                        __POLYSPIN_CPP_Phonebook_Insert_6);
    tt_message_iarg_add(msg, TT_OUT, "DUMMY", 0);
    tt_message_arg_add(msg, TT_IN, "char *", name);
    tt_message_arg_add(msg, TT_IN, "char *", number);
    tt_message_send(msg);
    tt_message_destroy(msg);
  }
  else ;  // This should never happen.
}
```

Figure 11.7. The modified body of the C++ `Insert` method after application of PolySPINner V3. The `name` and `number` parameters are marshaled in reverse order for relaxed compatibility with the CLOS `add` method.

```
(defmethod add ((book address-book)(number t)(name t))
  (cond
    ((not (Remote book))
     ;; Original method body, unchanged
     (setf (entry-list book)
           (cons (list name number)
         (entry-list book))))

    ;; Execute if I am a surrogate for a C++ object
    ((equal (Language book) LANG_CPP)
      (setq msg (tt-pnotice-create TT-SESSION (MyName book)))
      (tt-message-iarg-add msg TT-IN "integer"
                           __POLYSPIN_CPP_Phonebook_Insert_6)
      (tt-message-iarg-add msg TT-OUT "DUMMY" 0)
      (tt-message-arg-add msg TT-IN "t" name)
      (tt-message-arg-add msg TT-IN "t" number)
      (tt-message-send msg)
      (tt-message-destroy msg))

    ;; Execute if I am a surrogate for a CLOS object
    ((equal (Language book) LANG_CLOS)
      (setq msg (tt-pnotice-create TT-SESSION (MyName book)))
      (tt-message-iarg-add msg TT-IN "integer"
                           __POLYSPIN_CLOS_address_book_add_16)
      (tt-message-iarg-add msg TT-OUT "DUMMY" 0)
      (tt-message-arg-add msg TT-IN "t" number)
      (tt-message-arg-add msg TT-IN "t" name)
      (tt-message-send msg)
      (tt-message-destroy msg))

    (t
      (format t "ERROR: unknown language~%"))))
```

Figure 11.8. The modified body of the CLOS `add` method after application of PolySPINner V3. The `number` and `name` parameters are marshaled in reverse order for relaxed compatibility with the C++ `Insert` method.

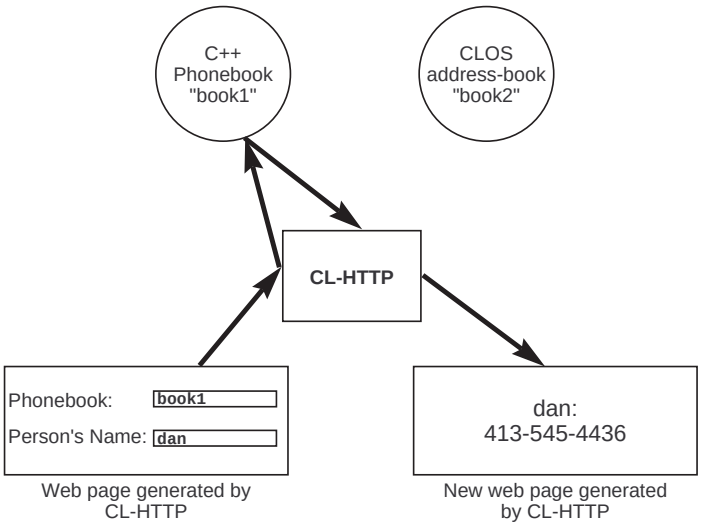


Figure 11.9. Common case: CL-HTTP seamlessly accesses the telephone directory selected by the user.

at the Massachusetts Institute of Technology. The features of CL-HTTP include traditional web service, automatic generation of web pages based on user-specified criteria, and the use of Common Lisp as a scripting language. This experiment made use of version 58.12 of CL-HTTP running under Liquid Common Lisp[2] version 4.2.

11.4.2 Integrating CL-HTTP and the Telephone Directories

In order to integrate CL-HTTP with the telephone directories of Section 11.3, a pair of CLOS methods were written. The first generates a web page to prompt the user for the name of a telephone directory ("book1" or "book2", indicating the C++ and CLOS directories, respectively) and the name of a person to look up in the directory. The second method, shown in Figure 11.10, fetches the instance of `address-book` named by the user, which (seamlessly) might be implemented in C++ or CLOS, and invokes the `lookup` method on that instance. This is a polylingual

[2]Formerly Lucid Common Lisp.

method call. The result is displayed on a new web page. Thus, the legacy application CL-HTTP communicates seamlessly with telephone directory objects of two distinct languages.

11.5 Scenario Three: Megaprogramming

In this experiment, the CL-HTTP CLOS web server of Section 11.4 was integrated with a C++ type from the GNU C++ library [44]. The type chosen was `Regex`, a representation of regular expressions [32]. The result of this experiment is that CL-HTTP gained the ability to use regular expressions in CLOS computations by seamlessly accessing regular expression objects implemented in C++. PolySPINner performs a subset match to make these two applications interoperate.

A sample application, depicted in Figure 11.11, was designed to verify the syntactic correctness of a Uniform Resource Locator (URL) [29] by comparing it to a regular expression. A web page generated by CL-HTTP prompts the user to type a URL. The string representing the URL is passed seamlessly to a C++ program with a `Regex` object containing a regular expression for URLs. The Boolean result of the regular expression test is returned to CL-HTTP to display to the user.

This experiment does integrate two legacy applications, but it was necessary to add a type to one legacy application (CL-HTTP) in order to achieve interoperability. Thus, while the experiment does demonstrate megaprogramming with respect to legacy applications, it does not demonstrate megaprogramming with respect to legacy types. The available selection of freely distributable C++ and CLOS source code is surprisingly limited, making it difficult to locate a pair of legacy applications—one in C++, one in CLOS—with a non-trivial, semantically similar pair of types. In an industrial setting, where proprietary source code is readily available, it is more likely that such a pair of applications could be found.

```
(defmethod address-book-engine ((url url:http-form)
                                stream
                                query-alist)
  ;; Receive address book name and person name from user input
  ;; on a web page.
  (bind-query-values
   (book name)
   (url query-alist)

   ;; Register with nameserver and fetch an address book.
   (setq ns (make-instance 'NameServer))
   (Register ns)
   (setq phonebook
         (change-class (Fetch ns book) 'address-book))

   ;; POLYLINGUAL METHOD CALL
   ;; on the address book selected by the user.
   (setq number (lookup phonebook name))

   ;; Generate a web page to display the results.
   (with-successful-response
    ;; CL-HTTP web-page layout calls omitted.
    ;; Display the retrieved telephone number.
      (cond ((equal number "")
             (format stream
                     "Sorry, no entry for ~A in book '~A'."
                     name book))
            (t
              (format stream
                      "~A's phone number in book '~A' is:"
                      name book)
              (cool-display number stream))))
   (Unregister ns)))
```

Figure 11.10. CL-HTTP "main" method that polylingually accesses a telephone directory of unstated language.

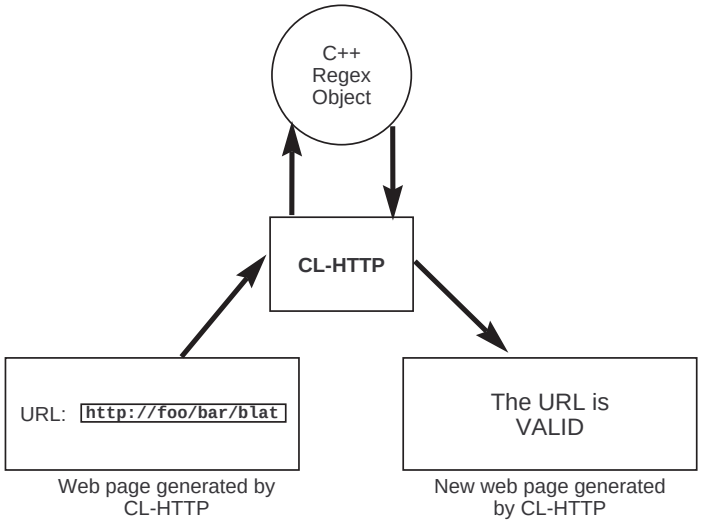


Figure 11.11. Megaprogramming: Polylingual interoperability between CL-HTTP and `Regex` object from the GNU C++ class library.

11.5.1 `libg++`: The GNU C++ Class Library

The GNU C++ library, `libg++`, is the work of numerous contributors to the Free Software Foundation's GNU project [69]. It is a collection of general-purposes C++ classes, such as stacks, queues, sets, bags, multiple precision integers, input/output streams, and random number generators. The type `Regex`, shown in Figure 11.12, also appears in the library and represents a *regular expression* [32].

A regular expression is similar to the familiar wildcards used to match filenames in UNIX and DOS. It is a sequence of symbols, such as `[f|s]*ox`, that represents a family of strings, the members of which are said to "match" the regular expression. This particular regular expression represents all strings that begin with zero or more occurrences of '`f`' or '`s`', followed by the letters `ox`. So this regular expression would match `ox`, `fox`, `sox`, and `sssffsfox`, but not `oxx` or `fax`.

The methods of the `Regex` type include a constructor (`Regex`) that compiles its regular expression argument for later matching, a matching function (`match`), and

```
// Copyright (C) 1988 Free Software Foundation
//    written by Doug Lea (dl@rocky.oswego.edu)
class Regex {
private:
  Regex(const Regex&) {}
  void operator= (const Regex&) {}

protected:
  re_pattern_buffer* buf;
  re_registers*      reg;

public:
  Regex(const char* t,
        int fast = 0,
        int bufsize = 40,
        const char* transtable = 0);

  ~Regex();

  int match(const char* s, int len, int pos = 0) const;
  int search(const char* s, int len,
             int& matchlen, int startpos = 0) const;
  int match_info(int& start, int& length, int nth = 0) const;
  int OK() const;
};
```

Figure 11.12. The C++ `Regex` type from the GNU C++ class library.

several others. Only the `match` method was used as a point of interoperability in the experiment.

11.5.2 Integrating CL-HTTP and `libg++`

In this experiment, a regular expression was constructed to match valid URLs. It is an overly permissive version of the true URL regular expression, a simplification that does not affect the interoperability aspects of the experiment. The URL used was:[3]

```
(http|ftp|gopher|telnet|mailto|news)://..*(:[0-9]*)?.*
```

which matches any string that begins with `http`, `ftp`, `gopher`, `telnet`, `mailto`, or `news`, followed by a colon, two slashes, an Internet hostname (any text), an optional colon and port number, and anything following it. For example:

```
http://relativity.cs.umass.edu:8000/polyspin
```

In order to integrate CL-HTTP with C++ `Regex` objects, a "dummy" `Regex` type was implemented in CLOS, as shown in Figure 11.13. It consists of a minimal `defclass` and a single method, `match`, with the same parameter types as the C++ `match` method but a nil body. This CLOS type was then given as input to PolySPINner V3 along with the real C++ `Regex` type, and PolySPINner was requested to perform a subset match between the types, since only the `match` method appeared in both. PolySPINner V3 then generated the CLOS `match` method shown in Figure 11.14.

As part of its normal execution, PolySPINner also generated a C++ main program and event loop to manage instances of the C++ `Regex` class, as explained in

[3]The GNU `Regex` syntax requires a backslash before each parenthesis and vertical bar, but these are omitted here for clarity.

```
(defclass Regex () ())

(defmethod match ((rx Regex) (s string) (len integer) (pos integer))
  (declare (return-values integer))
  nil)
```

Figure 11.13. The "dummy" `Regex` class implemented in CLOS.

```
(defmethod match ((rx Regex)(s string)(len integer)(pos integer))
  (cond
    ((not (Remote rx))
     nil )             ;; original dummy body, never executed

    ((equal (Language rx) LANG_CPP)
      (setq msg (tt-prequest-create TT-SESSION (MyName rx)))
      (tt-message-iarg-add msg TT-IN "integer"
                           __POLYSPIN_CPP_Regex_match_6)
      (tt-message-iarg-add msg TT-OUT "integer" 0)
      (tt-message-arg-add msg TT-IN "string" s)
      (tt-message-iarg-add msg TT-IN "integer" len)
      (tt-message-iarg-add msg TT-IN "integer" pos)
      (tt-message-send msg)
      (setq returnmsg (tt-wait-receive))
      (setq returnvalue (tt-message-arg-ival-return returnmsg 1))
      (tt-message-destroy msg)
      returnvalue)))
```

Figure 11.14. The CLOS `match` method body generated by PolySPINner V3. Only the CLOS-to-C++ case is shown.

Section 10.3.1. This program was linked with the `Regex` class to produce a server application. One line—a constructor call—was changed by hand to instantiate a `Regex` object with the regular expression for URLs. The C++ code was then compiled and linked.[4]

Next, a pair of CLOS methods was written for CL-HTTP to fetch a CLOS `Regex` object (actually, a C++ object hidden by a CLOS surrogate), seamlessly invoke the `match` method to verify a user-supplied URL against the regular expression, and print the result of the match on a web page. Thus, the CLOS web server gained the ability to match strings against regular expressions by interoperating seamlessly with a C++ program, without needing any regular expression-related code to be written in CLOS.

11.6 Shortcomings of the Experiments

In each of the experiments, a limitation of PolySPINner V3's C++ parser necessitated some hand-editing of the generated code. Due to an unfortunate design decision made in PolySPINner V2, the C++ parser stores only the C++ code that it recognizes as relevant to interoperability, such as class definitions, and ignores other code such as preprocessor directives and global (`extern`) variable declarations. Thus, the C++ Generator cannot completely reproduce all parts of the input. It correctly generates the classes and methods needed for polylingual interoperability, but obviously cannot generate the parts that were ignored by the parser.

For example, the original file `Regex.C`, containing the definition of the `Regex` methods of Section 11.5, had an `extern "C"` statement that was necessary to import some C code into the file. PolySPINner's C++ parser ignored it. Thus, it was necessary to reinsert this statement by hand after PolySPINner generated its code. Each experiment had a few instances of similar cutting and pasting from the original

[4]Subject to the slight hand-editing described in Section 11.6.

sources to the generated files, but *in no case* was the code related to interoperability. A full C++ parser for PolySPINner would eliminate this problem.

11.7 Performance

PolySPINner is intended to be built on top of another communication substrate, such as an RPC mechanism (e.g., ToolTalk), a database (e.g., the Open OODB), or a CORBA ORB. Thus, when measuring the performance of PolySPINner, the most relevant statistic is how much overhead PolySPINner adds beyond that of the substrate. It is less relevant to compare the raw speed of PolySPINner's generated code against the speed of another interoperability mechanism—say, a particular implementation of CORBA—because if the other mechanism were faster, PolySPINner could be rehosted on top of it. Thus, in this section, we measure PolySPINner's overhead.

Since PolySPINner V3 adds only a few conditionals and function calls to the ToolTalk code it generates, one would expect the performance of its generated method calls to be approximately the same as that of unadorned ToolTalk calls. Table 11.1 summarizes the results of executing 1000 PolySPINner method calls versus 1000 ToolTalk message send/receive pairs. Both test programs marshaled and unmarshaled the same parameters. Ten trials of each test were conducted, and the mean and standard deviation were calculated. Overall, the additional code generated by PolySPINner added 6–8% overhead to the ToolTalk calls, quite acceptable considering the transparency gained by using PolySPINner. Additionally, no efforts have been made to optimize PolySPINner's generated code, so improved performance is likely.

The large discrepancy between Real Time and the other two measurements is due to the current implementation of synchronous remote method calls used by PolySPINner's code generator and the test programs. Busy-waiting is used by both the

Table 11.1. Performance of PolySPINner-generated code vs. native ToolTalk code, in seconds, as calculated by UNIX `/bin/time`. Ten trials of 1000 message send/return pairs were measured.

Test	Real Time		User Time		System Time	
	Mean	Std Dev	Mean	Std Dev	Mean	Std Dev
PolySPINner	39.22	1.29	6.43	0.26	3.51	0.28
Native ToolTalk	36.82	1.49	5.91	0.37	3.24	0.25
% Slowdown	6.1%		8.1%		7.7%	

callee process (waiting for remote method calls) and caller process (waiting for return values), and the competition for the CPU drives up the Real Time.

11.8 Summary

The experiments in this chapter have demonstrated that PolySPINner is a viable tool for integrating software components to interoperate seamlessly. PolySPINner V3 was applied in all three interoperability scenarios—easiest case, common case, and megaprogramming[5]—and was capable of integrating components with little effort on the part of the developer. In addition, the seamlessness of interoperability was demonstrated in each case. PolySPINner is also nearly nonintrusive. Currently, a small amount of non-interoperability-related code must be copied from the original files into the generated files. If PolySPINner's parser were replaced with one covering the full C++ grammar, this problem would disappear and PolySPINner would be nonintrusive.

[5]Megaprogramming with respect to applications, rather than types, as discussed.

CHAPTER 12

CONCLUSIONS

This dissertation has presented a broad examination of polylingual systems: their history, definition, theory, strengths, weaknesses, and practical application. Before the present research was undertaken, polylingual systems concepts were only marginally defined and minimally tested. Now there is a significant body of knowledge that establishes polylingual systems as a legitimate and interesting field of study.

12.1 Summary of Results

The original contributions of this dissertation include:

- In general, expanding the frontiers of polylingual systems research in several directions, thereby establishing polylingual interoperability as a viable means of integrating multilanguage software.
- The continued development of PolySPIN, a seamless, nonintrusive approach to interoperability that does not require an IDL.
- A formal foundation, based on object-oriented type theory and relaxed matching, for interlanguage compatibility of types.
- A rewritten and greatly enhanced PolySPINner tool, now supporting relaxed matching, inheritance, and type safety features within a modular, extensible architecture, with only 6–8% performance overhead beyond the underlying communication mechanism.

- Experiments demonstrating that PolySPINner may be used to achieve interoperability in the easiest case, common case, and megaprogramming scenarios.
- The Interoperability Manifesto, a set of criteria by which interoperability approaches may be compared and contrasted, and initial statistical evidence of its suitability.
- A detailed comparison of CORBA and PolySPIN.
- A fleshing out of previously unstated coupling between polylingual interoperability and persistence, by augmenting PolySPIN and PolySPINner to support interoperability among transient objects.

12.2 Future Work

Although polylingual systems have now been defined, formalized, and implemented, there are still many open issues in the field.

12.2.1 General Issues

Dynamic behavior. PolySPIN, as implemented in PolySPINner V3, is currently applied to software components before they are compiled. As a result, polylingual interoperability can be based only on static features of the components. It is not possible, for example, for PolySPINner to infer the return type of a CLOS method statically; so CLOS methods are required to `declare` their return values before PolySPINner can process them, as explained in Section 9.5.2. If PolySPIN could be applied at run-time rather than compile-time, instrumenting the method implementations on demand, perhaps it could handle such issues, or even cease requiring the modification of source code altogether. Languages with a reflection capability, such as Java, would facilitate such an implementation.

Distribution. When hosted on top of ToolTalk, an RPC mechanism, PolySPINner gained the ability to generate distributed polylingual systems. It is clear from Section 10.3.3, however, that deadlock issues have not been resolved, and issues of concurrency control have not been examined to see how (or if) they pertain to PolySPIN. In any event, a more robust mechanism for safe yet flexible distribution in polylingual systems is needed.

12.2.2 Type-Related Issues

Specification matching. PolySPIN's formal framework of Chapter 6 is based on the work of Zaremski and Wing on signature matching. These authors have extended this work to specification matching [76], in which formal specifications in the Larch language[1] [27] are attached to functions. Pairs of functions are then matched, using relaxed matching on the specifications rather than the signatures. It is natural to suppose that specification matching could be applied across programming languages as well.

Exception handling. The interlanguage method matching of Chapter 6 does not consider the exceptions raised by methods.[2] Two methods with identical signatures may raise different exceptions and therefore not be strictly compatible. PolySPIN should evaluate the compatibility of exceptions as well as signatures. This will present difficulties particularly in languages that do not explicitly declare the types of exceptions raised by a method, such as CLOS.

Variable-length parameter lists. C++, CLOS, and other languages support the definition of methods that have parameter lists of variable length. C++ uses the `stdargs` construct [46] which cannot be statically type-checked in general. CLOS

[1] Actually, it is Larch/ML, a larch interface language for the ML programming language.

[2] Neither do many approaches to monolingual, object-oriented type theory.

uses the `&optional` construct [26] which, again, cannot be statically type-checked but at least indicates the maximum number of parameters permitted. Compatibility between methods with variable-length parameter lists, or between variable-length and fixed-length parameter lists, is a direction for future research, but it is not at all clear how the formalism of Chapter 6 could be applied.

Improved handling of primitive type compatibility. PolySPINner does not have a sophisticated mechanism for matching primitive types between languages (e.g., integers, chars), focusing instead on abstract data types. Currently it uses a simple lookup table that cannot be changed by the interoperability engineer. Application of a more robust, flexible mechanism—particularly one that supports a form of relaxed matching among primitive types, such as Mockingbird [4]—should be explored.

Attributes. The PolySPIN technique fundamentally relies on the separation of interface and implementation, and so it may be applied to methods. Attributes, however, do not have this separation, and thus they cannot be accessed polylingually, except via a method. It is possible that PolySPIN could support the generation of additional, invisible methods that mimic the setting and getting of attributes, and thereby permit polylingual access of attributes.

Parameter modes. Abadi and Cardelli address the issue of parameter modes (e.g., in, out, inout) in detail [1] for a single object-oriented language. This should be extended to the multilingual setting in the manner of Chapter 6. While PolySPINner V3 is not limited to "in" parameters, the formalism currently is.

Cross-language type generation. Rather than having the user supply matching type definitions in several languages, we would like PolySPINner to automate this process, generating corresponding type definitions to as great an extent as possible.

This is a difficult task, particularly when one language supports arbitrary levels of pointer indirection, such as C++, and the other has no pointers, such as Java or CLOS.

One-to-many type matching. PolySPINner V3 currently supports matches between pairs of types. It is conceivable, however, that a match may be one-to-many: the methods of a single type t could match methods found in two or more other types. For example, type t may contain two methods that match others found in type t', and three methods that match others in type t''. Nothing in the PolySPIN technique inherently prevents a one-to-many match, but the topic has not been explored.

12.2.3 Language Support

Necessity of object-orientedness. PolySPIN is currently applicable only to software components written in object-oriented languages. This is because PolySPIN requires the specification and implementation of methods to be separate. Vast numbers of legacy programs, however, are written in non-object-oriented languages. The potential application of PolySPIN (or PolySPIN-like techniques) to these languages should be explored.

PolySPIN support for Java. This is in progress as part of the JSPIN project [62] in the Convergent Computing Systems Lab. Also, the addition of a third language to PolySPINner's arsenal will test the toolset's robustness and its specification for adding new language support.

12.2.4 Interoperability Manifesto

Additional interoperability features. No claim has been made that the nine features of the Interoperability Manifesto (Chapter 3) are comprehensive. Software engineers should be surveyed to determine whether other important features can be isolated.

Finer points of distinction. The criteria of the Interoperability Manifesto could be made finer. For example, interoperability approaches are currently rated either as "intrusive" or "nonintrusive," but there are undoubtedly levels of grey. Should an approach that simply adds statements to the main program of an application be considered just as intrusive as one that modifies type declarations? Arguably not. The other interoperability criteria should be examined in a similar manner, and the results evaluated by another survey of professionals.

Creation of scales. Once the criteria have been made finer, it would be desirable to have actual scales by which the support for each feature could be measured. This should include the surveying of large numbers of interoperability designers and engineers in order to calculate weights for the different aspects of interoperability. For instance, the Manifesto currently counts the number of languages supported by an interoperability approach, deigning that more is better. But should support for 10 highly similar languages be considered "better" than support for three drastically different languages?

Further comparison with CORBA. It is unknown whether the creation of compatibility criteria in PolySPINner is easier or harder, in practice, than the integration of IDL interfaces with a legacy application. This should be examined through experimentation by assigning programmers to integrate legacy applications with both techniques and fill out questionnaires about their experiences.

12.2.5 Implementation

More thorough C++ parsing. As described in Section 11.6, the C++ parser of PolySPINner V3 has technical limitations that prevent the C++ Generator from completely recreating the input files. A full, robust C++ parser should be used instead,

ideally developed from a Bison grammar file used in an actual compiler, not written from scratch.

Additional tools. Polylingual system construction is in its infancy. It would benefit from better tools, such as a multilanguage debugger and an interlanguage static analyzer.

Better user interface. A graphical user interface to PolySPINner would be desirable, especially for interactive selection of methods and parameters to match.

Automatic generation of the object definition file. PolySPINner V3 cannot generate a main "server" program for objects without knowing the names, types, and languages of the objects. This information current is supplied by the interoperability engineer. PolySPINner should infer this information whenever possible.

Support for same-language relaxed matching. PolySPINner's Matcher compares types while they are in a language-neutral intermediate representation. Thus, the Matcher can operate on two types of the same language. Thus it is reasonable to expect that PolySPINner could perform relaxed matching between types of the same language, perhaps in the service of type evolution. Currently, a design limitation of the Matcher in PolySPINner impedes this.

Optimization. PolySPINner's generated code has not been optimized. In particular, same-language method calls in PolySPINner V3 should be automatically converted from RPC to foreign function calls when possible, and busy-waiting should be eliminated.

ToolTalk passing of objects. PolySPINner V3, when hosted on top of ToolTalk, lost the ability of PolySPINner V2 to pass objects as parameters due to the deadlock problem cited in Section 10.3.3. This is not acceptable and should be fixed.

Upgrade and repair the Open OODB implementation. A series of software environment changes, detailed in Section B.4.2.1, prevents the code generated by the Open OODB-based implementation from compiling. PolySPINner should be rehosted on Open OODB version 1.0, rather than 0.2, and its code generator updated for the new environment.

APPENDIX A

INTEROPERABILITY SURVEY

A.1 Survey Goals

This interoperability survey described in Chapter 3 was designed to verify (or not) that the features of the Interoperability Manifesto are considered generally important by software developers. Software developers were asked to rate the importance of various interoperability features, including those of the Interoperability Manifesto. If any feature was consistently regarded as unimportant, it could be dropped from the Manifesto. Likewise, if no features were generally regarded as unimportant, this would be evidence that the Manifesto's features were well chosen.

A.2 About the Survey

A.2.1 Sampling Procedure

The survey was posted in eight Usenet newsgroups, listed in Table A.1, on June 16, 1997. In addition, eight individuals in the field were sent the survey directly. Responses were collected until July 14, 1997. A total of 13 people responded: 9 from Usenet and 4 of the individuals directly solicited. A $20 prize was offered as incentive to participate and was awarded to one randomly chosen respondent.

A.2.2 Survey Overview

The survey, presented in full in Section A.3, consisted of five parts:

I. Background information about the respondent.

Table A.1. Usenet newsgroups for the survey.

Newsgroup	Topic
comp.client-server	Client-server computing
comp.software-eng	Software engineering
comp.object.corba	The CORBA approach
alt.soft-sys.tooltalk	The Sun ToolTalk mechanism
comp.unix.programmer	UNIX programming
comp.os.ms-windows.programmer.ole	Microsoft OLE
comp.os.ms-windows.programmer.misc	Microsoft Windows programming
comp.sys.mac.programmer.misc	Macintosh programming

II. Experience with particular interoperability mechanisms.

III. Ratings of interoperability features.

IV. Choosing the top five interoperability features from the previous part.

V. General comments.

Parts II and II collected demographics. Part I asked about the respondent's age, sex, and other basic information. Part II asked about the level of experience the respondent had with each of 14 interoperability mechanisms. Experience was judged by the size of the largest project in which the respondent had used the mechanism.

Part III was the most important section: the respondent's opinions on the importance of 25 interoperability features. All features of the Interoperability Manifesto were represented here, as well as other features related to interoperability but not directly relevant to this dissertation. Features were phrased as questions that one could ask about a particular interoperability mechanism, such as: is it transparent? How well does it perform? Does it support interoperability of array types? Participants rated the importance of questions on a 5-point Likert scale [33]. Part IV forced the respondent to choose his/her five most important features from the original 25. This was done to avoid a respondent's tendency to rate many features as equally important. Finally, Part V was included for respondent feedback.

A.3 Survey Form

PART I: BACKGROUND INFORMATION

A1) What is your age?

A2) What is your sex?

A3) What is the highest level of schooling you have completed: high school, college, masters, doctorate?

A4) Are you currently employed in industry, academics, both, or neither?

A5) How many years' experience do you have with multilanguage interoperability on the job?

PART II: Interoperability Experience

Instructions: I'd like to find out which interoperability systems you have used, and your level of expertise. Answer each question by typing a number from 1 through 5 to indicate:

1	*No experience with the system whatsoever.*
2	*I have read about the system, but never used it.*
3	*Used the system for a small project (up to 5,000 lines of code).*
4	*Used for a medium project (up to 25,000 lines of code).*
5	*Used for a large project (more than 25,000 lines of code).*

S1) CORBA

S2) OLE or ActiveX

S3) SoftBench

S4) FIELD

S5) ToolTalk

S6) Isis

S7) Polylith

S8) ILU

S9) JavaBeans

S10) Any database

S11) Any object-oriented database

S12) Any RPC (remote procedure call) system

S13) Any foreign function call mechanism

S14) An interoperability system you created yourself (and what is the name of the interoperability system?)

PART III: Feature Ratings

Instructions: Suppose you are working on a large software project that requires interlanguage communication. You have the opportunity to use an unfamiliar interoperability product called XYZ for this purpose.

You arrange a meeting with an XYZ expert who can answer any questions you may have about XYZ. Below is a list of questions that you might ask. Please rate the IMPORTANCE of each question to your decision to use, or not use, interoperability product XYZ. Each question should be rated on a scale of 1 to 5:

1	*Completely irrelevant—I do not need this information to make a valid decision.*
2	*Interesting, but of little importance.*
3	*Of average importance.*
4	*Very important.*
5	*Extremely important—I could not make a valid decision without this information.*

Q1) How many programming languages are supported by XYZ?

Q2) Which programming languages are supported by XYZ?

Q3) Which programming language models are supported, e.g., procedural, functional, object-oriented, logic programming?

Q4) How easy is it to extend XYZ myself to add support for a new language?

Q5) How fast does XYZ perform while communicating data across the language barrier?

Q6) How much memory does XYZ require?

Q7) How much disk space does XYZ require?

Q8) What platforms does XYZ run on?

Q9) How transparent is XYZ to developers? That is, how visible will XYZ be in my application source code?

Q10) How "intrusive" is XYZ to my legacy systems? That is, how much modification will my legacy systems need in order to be made interoperable?

Q11) What type-safety guarantees does XYZ make when transforming data from its representation in one language to its representation in another?

Q12) How strict is type compatibility in XYZ? That is, how similar do objects in different languages need to be in order to be considered compatible?

Q13) How similar are XYZ APIs for different languages? For instance, does XYZ's API for C++ look similar to the one for Lisp?

Q14) In order to make a legacy system interoperable, does XYZ require access to the legacy system's source code, or can it integrate executables directly, without needing to recompile them?

Q15) Does XYZ have an IDL, module interconnection language, or other foreign type system, requiring developers to translate their programming language types into XYZ types?

Q16) Does XYZ support interlanguage communication of primitive types, such as integers and reals?

Q17) Does XYZ support interlanguage communication of structured types, such as arrays, records and unions?

Q18) Does XYZ support interlanguage communication of pointer types?

Q19) Does XYZ support interlanguage communication of abstract data types (e.g., classes, modules, packages)?

Q20) Does XYZ support first-class types, that is, objects of type "Type"?

Q21) How automated is XYZ? That is, to what extent can interoperability be added to a software system without human intervention?

Q22) Who created XYZ?

Q23) How satisfied are other users of XYZ?

Q24) How much does XYZ cost?

Q25) Is the source code to XYZ available so I can modify it if necessary?

PART IV: Your Top Five Picks

Instructions: Please identify the five questions from Part III, above, that you would MOST want to ask the XYZ expert, in order of importance, beginning with the most important. List them by number.

PART V: Additional Details (Optional)

If you would like to elaborate on any of your answers, please do so below. Identify the question by number (e.g., Q16) and then add your comments.

A.4 Analysis

A.4.1 Participants' Rankings

The first part of the analysis focused on participants' ratings of the importance of interoperability features, as given in Parts III and IV of the survey, and the results are summarized in Table A.2. First, I counted the number of appearances of each interoperability feature in respondents' "Top Five Picks" lists (Part IV). All five picks were assigned equal weight. The highest rated features were Q2 (10 appearances), Q8 (7), Q9 (4), Q16 (4), and Q17 (4).

Table A.2. Six highest-rated interoperability features. Boxes indicate the highest-rated features for each scale.

Feature	"Top 5" Lists	Mean Importance
Q1	0	2.667
Q2	10	4.692
Q3	3	3.231
Q4	2	2.923
Q5	3	3.615
Q6	0	2.923
Q7	0	2.462
Q8	7	4.462
Q9	4	3.846
Q10	3	4.154
Q11	1	3.846
Q12	0	3.417
Q13	0	3.000
Q14	1	3.231
Q15	2	3.769
Q16	4	4.308
Q17	4	4.154
Q18	0	3.615
Q19	2	3.750
Q20	0	3.000
Q21	0	3.077
Q22	0	2.000
Q23	3	4.077
Q24	3	3.615
Q25	3	3.083

Next, as a sanity check, I calculated the mean importance rating for each feature as given in Part III, yielding another scale by which to assess the importance of the features. Using this procedure, four of the same five features again appeared as the highest-rated: Q2, Q8, Q16, and Q17. A fifth feature, Q10, was rated highly but was not among the five highest of the previous scale.

Finally, the five highest-rated features according to the two scales were determined. The results in Table A.2 indicate that Q2, Q8, Q16 and Q17 were the highest-rated features according to both scales. In addition, Q9 was one of the five highest-rated features in the "Top Five Picks" scale, and Q10 was among the five highest in the mean importance scale. The union of the results from both tabulation procedures was taken to be the six highest-rated interoperability features given in Section 3.5.

Although some features did not appear in any "top five" list, every feature received at least one rating of 4 or higher, as shown in Table A.3. Thus, each feature was considered very important by at least one practitioner. This provides evidence that the features in the Interoperability Manifesto are generally considered important by other software developers.

A.4.2 Effects of Experience

Next, I assessed whether experience with an interoperability mechanism affected participants' choice of important interoperability features. Participant's years of experience with the interoperability mechanism in Part II were correlated with their preferences for the interoperability features of Part III, using a Spearman rank-order correlation [33]. The correlations that were significantly different from zero appear in Tables A.4 and A.5.

Table A.3. Number of times each feature was assigned each rating. Features Q1, Q12, Q19, Q20 and Q25 were each skipped by one respondent.

	Ratings				
Feature	**1**	**2**	**3**	**4**	**5**
Q1	2	3	5	1	1
Q2	0	0	0	4	9
Q3	2	2	3	3	3
Q4	1	4	5	1	2
Q5	0	2	5	2	4
Q6	0	5	5	2	1
Q7	1	6	5	1	0
Q8	0	0	1	5	7
Q9	0	0	6	3	4
Q10	0	1	1	6	5
Q11	1	1	3	2	6
Q12	1	1	4	4	2
Q13	0	5	3	5	0
Q14	0	5	3	2	3
Q15	1	0	3	6	3
Q16	1	0	1	3	8
Q17	1	0	1	5	6
Q18	1	0	5	4	3
Q19	1	1	3	2	5
Q20	1	4	3	2	2
Q21	1	4	2	5	1
Q22	5	4	3	1	0
Q23	0	0	4	4	5
Q24	1	0	4	6	2
Q25	2	1	4	4	1

Table A.4. Spearman rank-order correlations of interoperability features vs. years of experience with interoperability mechanisms. Only significant correlations are shown.

#	CORBA	OLE	SoftBench	FIELD	ToolTalk	Isis	Polylith
Q1							
Q2							
Q3			−.4715		−.5675		
Q4							
Q5			.5035				
Q6			.6068				
Q7							
Q8							
Q9		−.8183		−.5866		−.7165	
Q10							
Q11						−.5594	
Q12						−.5108	
Q13							
Q14		.4859					
Q15		−.5191					
Q16							
Q17							
Q18							
Q19							
Q20						−.6163	−.4936
Q21				−.5374			−.6082
Q22							
Q23				−.5745		−.4841	−.6664
Q24							
Q25					−.6504		

Table A.5. More Spearman rank-order correlations of interoperability features vs. years of experience with interoperability mechanisms. Only significant correlations are shown.

#	ILU	JavaBeans	Databases	OODBs	RPC	FFC	Custom
Q1		.5394					
Q2							
Q3							
Q4							
Q5							
Q6						−.5678	
Q7							
Q8							
Q9		−.4850					
Q10	.6041						
Q11							
Q12		−.5078			−.5410		
Q13							
Q14							
Q15					.4842		
Q16		−.6785					
Q17							
Q18							
Q19							
Q20							−.7542
Q21							−.6407
Q22		.5596					
Q23							
Q24	.4897						
Q25			−.5065	−.5747			

Several of the correlations yielded by the analysis suggest interesting causal relationships:[1]

- The more experience that respondents had with OLE, the less important they considered transparency (Q9).[2] Users of FIELD and Isis, other non-transparent interoperability mechanisms, also seemed unconcerned with transparency.

- The more experience they had with OLE, the less respondents cared whether an interoperability mechanism used an IDL (Q15). This is interesting since the premier IDL approach, CORBA, is the primary competitor of Microsoft's OLE.

- Experienced users of databases, both traditional and object-oriented, thought it unimportant to have access to the source code of the interoperability mechanism (Q25), perhaps reflecting the division between the database and programming language communities.

- JavaBeans users gave a low importance rating to interlanguage communication of primitive types (Q16), perhaps reflecting Java's emphasis on abstract data types. JavaBeans users also considered transparency unimportant, perhaps reflecting the fact that JavaBeans is used within the Java language, making interlanguage transparency a moot point.

- Most people who had created their own interoperability mechanisms (column "Custom") did not think that automation (Q21) is important, an unsurprising attitude for those with the skills to develop an interoperability mechanism.

---

[1] Although correlations cannot determine causality (i.e., the statement "A and B are correlated" implies neither "A causes B" nor "B causes A"), it is unlikely that participants' preferences for certain features caused them to have experience with particular mechanisms. Thus, it is likely that participants' experience influenced their preferences.

[2] Perhaps this reflects Microsoft's seeming attitude that if Microsoft sets the standards, interoperability is solved, and thus transparency is moot.

Finally, an analysis was performed to determine whether users of similar interoperability mechanisms had similar preferences for interoperability features. This was accomplished by correlating the vectors of experience-feature correlation coefficients across interoperability mechanisms. All correlation coefficients used in the analysis were transformed using r-to-z transformations [33].

The results, summarized in Table A.6, suggest strong similarities in the opinions of SoftBench and ToolTalk users (.8341), two commercial mechanisms based on the same abstraction, event-based software integration. Three research implementations of event-based integration—FIELD, Polylith, and Isis—also had very high correlations among their users' preferences (.8705, .9042, .7410). Users of databases and object-oriented databases agreed strongly (.7869). Other significant correlations, however, do not fit such convenient explanations, such as the strongly negative correlation ($-.5340$) between the opinions of OODB users and designers of custom interoperability mechanisms.

A.5 Summary

Every feature of the Interoperability Manifesto was rated as "very important" or higher by at least one of the 13 respondents. In addition, no participant suggested that any features were missing from the survey. These observations provide evidence that the Manifesto's features are defensible.

The opinions of respondents appear to be influenced to some extent by their experience with particular interoperability mechanisms. Although this does not directly affect the utility of the Manifesto, it is interesting to observe.

It would be worthwhile to repeat this experiment with a larger sample to obtain inferential, rather than just descriptive, results.

Table A.6. Significant correlations of correlations, 0.5 or higher. + means positive, – means negative. Table is symmetric about the diagonal, so only the upper half is shown.

Mechanism	CORBA	OLE	SoftB	FIELD	ToolT	Isis	Polyl	ILU	JavaB	Datab	OODBs	RPC	FFC	Custo
CORBA														
OLE				+		+	+		+			+		
SoftBench					+									
FIELD						+	+	+				+		
ToolTalk										+	+			
Isis							+	+	+			+		+
Polylith								+						
ILU									+			+		
JavaBeans												+		
Database											+			
OODBs														–
RPC														
FFC														
Custom														

APPENDIX B

POLYSPINNER'S COMPONENTS

B.1 Overview

PolySPINner's architecture consists of three major kinds of components: parsers, matchers, and generators. Parsers translate types from their native programming language into PolySPINner's language-neutral, internal representation. Matchers compare types to determine their compatibility with one another. Generators create modified versions of types that may be accessed polylingually. All three component types are generic and must be instantiated for particular uses. This implementation hides all details of programming language differences and low-level communication from the main PolySPINner "engine."

B.2 Parser

The `Parser` class, depicted in Figure B.1, is a template. The interoperability designer instantiates the template by providing a parser for a particular language—written in Bison, for example—encapsulated as a class with a `Parse` method. `Parse` must return a list of classes, represented as `Class` objects in PolySPINner's intermediate representation. A sample parser for C++ is depicted in Figure B.2. To parse a C++ type, PolySPINner instantiates an object of type `Parser<CPP_Class>` and invokes its `Parse` method.

The output of a `Parser` is a list of abstract classes stored in PolySPINner's intermediate representation. This consists of objects of types `Class`, `Method`, and

```
template <class YourParser>
class Parser {
private:
  Filename InFileName;
  YourParser TheParser;
public:
  Parser(Filename *);
  ~Parser();
  List<Class *> *Parse(void);
};
```

Figure B.1. The generic `Parser` template.

```
class CPP_Class : public Class {
public:
  List<Class*> *Parse(const String SourceFileRoot);
};
```

Figure B.2. A C++ parser used to instantiate the `Parser` template.

`Parameter`, interconnected as shown in Figure B.3.[1] These are discussed in the following sections.

B.2.1 Type `Class`

An instance of type `Class`, depicted in Figure B.4, represents a class: that is, a named set of methods. It is a node of the inheritance graph of a software component, with edges (pointers) connected to the class's supertypes and subtypes.

Primitive types, such as integer and real, also are represented by PolySPINner as instances of type `Class`. This design decision allows PolySPINner's other components, such as matchers, to process all data types uniformly, whether primitive or abstract.

[1] This representation provides capabilities similar to Lisp's Meta-Object Protocol or Java's Reflection interface, but which are not present in C++.

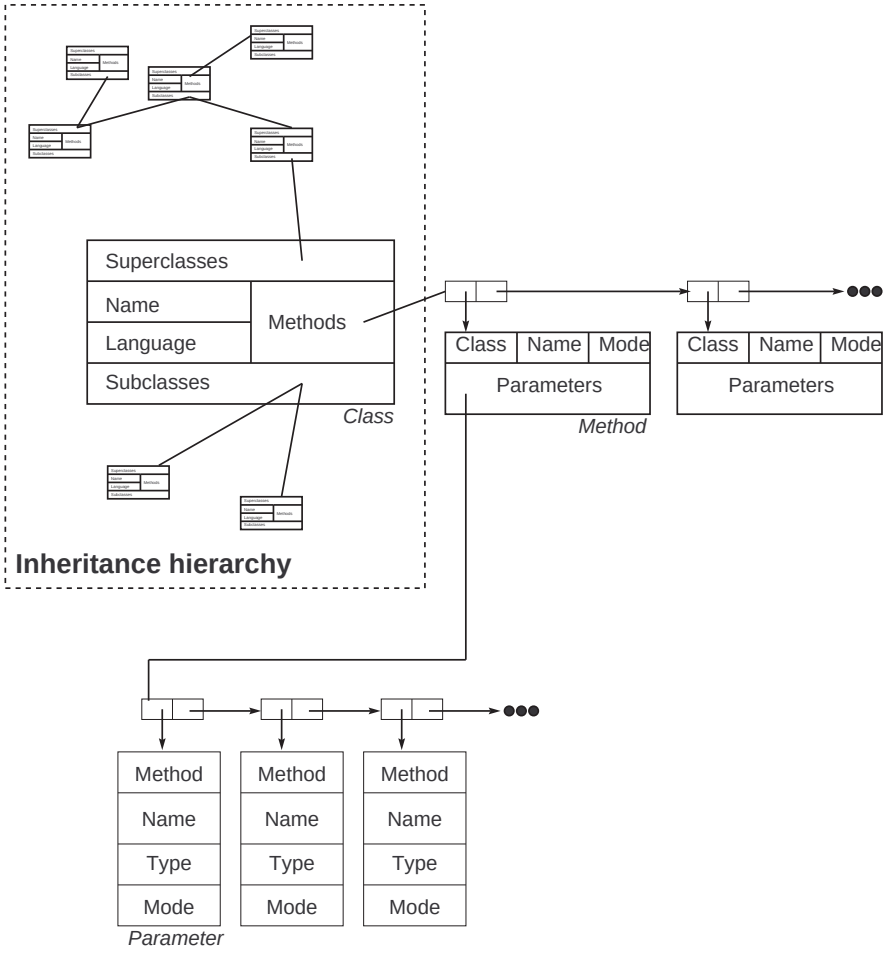


Figure B.3. PolySPINner's intermediate representation of interconnected `Class`, `Method`, and `Parameter` objects.

```
class Class {
// non-public members omitted
public:
  Class ();                           // constructors
  Class (char *name);
  Class (String &name);
  int operator== (Class x);           // class equality - structural

  void Insert (Method *op);           // insert a new method
  Method* Retrieve (int index);       // retrieve a method by index
  String& Name ();                    // retrieve the class name
  void Print();                       // print the class
  int NumberOfMethods ();             // how many methods?
  String& Language ();                // retrieve the language

  Boolean AddSuperclass(Class *);     // add a superclass
  Boolean AddSubclass(Class *);       // add a subclass
  Boolean IsSuperclass(Class *);      // is this a superclass?
  Boolean IsSubclass(Class *);        // is this a subclass?
  Boolean IsAncestor(Class *);        // is this an ancestor?
  int NumSuperclasses();              // how many superclasses?
  int NumSubclasses();                // how many subclasses?
  Class *IthSuperclass(int i);        // retrieve Ith superclass
  Class *IthSubclass(int i);          // retrieve Ith subclass

  void ResetMethods();                // iterator for methods
  virtual Method *GetNextMethod();
};
```

Figure B.4. The `Class` type of PolySPINner's intermediate representation.

Notice that the `CPP_Class` type (C++ class), depicted in Figure B.2, is a subtype of `Class`. In other words, the language-specific `Parse` method is placed into a subtype of `Class`. This relationship was a design decision made in the original PolySPINner V0 and was kept solely for compatibility with the legacy code.

B.2.2 Type `Method`

An instance of type `Method`, depicted in Figure B.5, represents a method: that is, a named, ordered list of parameters, and a body of code to execute. Instances of `Method` can also be assigned a `ProtectionMode` such as public, private, or protected.

Fields (attributes) of a type also are represented as instances of `Method`. This design decision permits fields and methods to be processed uniformly by PolySPINner's other components; it is not a type theoretic decision. The operation `Field` indicates whether a given instance is a method (`Field() = False`) or a field (`Field() = True`).

The `Method` type is not rich enough to represent all needed aspects of methods in all object-oriented languages. For example, it has no facility for representing a virtual method in C++, or for distinguishing `defmethod`s from slot methods in CLOS. The interoperability designer is expected to create subtypes of `Method` to handle such language-specific features. Figure B.6 illustrates a subtype for CLOS methods.

B.2.3 Type `Parameter`

An instance of type `Parameter`, depicted in Figure B.7, represents a formal parameter: that is, a named instance of a class appearing in a parameter list. Implicit parameters, such as the `this` parameter of C++, are distinguished from ordinary parameters. Parameter-passing modes, such as "call by value" and "call by address," are represented as well, though PolySPINner V3 currently does not consider these modes when performing type matching.

```
class Method {
// non-public members omitted
public:
  Method ();                          // constructors
  Method (Class* definingClass);
  Method (char* name, char* returnType, Class* definingClass);
  int operator==(Method x);           // method equality - structural

  String& Name ();                    // name of method
  String& PublishedName ();           // unique name across languages,
                                      // made & used by PolySPINner
  String& ReturnType ();              // return type of method
  virtual Boolean Returns();          // do I return a value?
  Boolean& Field();                   // is this a method or field?

  Class*  DefiningClass();            // class that defined method
  int NumberOfParameters ();          // number of parameters
  ProtectionMode& Protection();       // protection mode, e.g., private

  String *GetBody();                  // retrieve method body
  void SetBody(String *s);            // store method body

  void Insert (Parameter *parm);      // add a parameter
  Parameter* Retrieve (int index);    // retrieve a parameter

  virtual void Print();               // print the method

  void ResetParams();                 // iterator for parameters
  virtual Parameter *GetNextParam();
};
```

Figure B.5. The `Method` type of PolySPINner's intermediate representation.

```
class CLOS_Method : public Method {
public:
  Boolean& SlotSpec(); // Am I a defmethod or slot method?
  Boolean Returns();   // Specialized so "nil" means no return value
};
```

Figure B.6. Highlights of subtype `CLOS_Method`, specialized for CLOS methods.

```
class Parameter {
// non-public members omitted
public:
  Parameter ();                        // constructors
  Parameter (char* typeName, char* formalName,
             Mode passBy, Method* definingMethod);
  Parameter (Class *c, char* formalName,
             Mode passBy, Method* definingMethod);

  int operator==(Parameter x);         // equality - structural

  Method* DefiningMethod ();           // method containing me
  void SetDefiningMethod (Method *);

  Class *GetClass ();                  // type of the parameter
  String& Formal ();                   // name of the parameter
  Mode PassBy ();                      // mode of the parameter

  void MakeImplicit ();                // designate as implicit
  Boolean IsImplicit ();               // Is parameter implicit?

  virtual void Print();                // print the parameter
};
```

Figure B.7. The `Parameter` type of PolySPINner's intermediate representation.

```
class CPP_Parameter : public Parameter {
public:
  void AddPointer();                  // Add a level of indirection
  int NumPointers();                  // How many levels of pointers?
  void AddReference();                // Make me a reference parameter
  Boolean IsReference();              // Am I a reference parameter?
};
```

Figure B.8. Highlights of subtype `CPP_Parameter`, specialized for C++ methods.

```
class Matcher
{
// non-public members omitted
public:
  Matcher(MatchResult *(*)(Class *, Class *));
  MatchResult *Match(Class *, Class *);
};
```

Figure B.9. The generic `Matcher` class.

The `Parameter` type, like the `Method` type, is not general enough to represent all features of parameters in all languages. For example, it does not store the fact that a parameter type is a pointer or a reference in C++. Specialized classes, like `CPP_Parameter` in Figure B.8, are created by the interoperability designer to store this information.

B.3 Matcher

Matchers in PolySPINner, like parsers, are represented as a generic class that is instantiated with a particular match algorithm, a.k.a, a compatibility criterion. Whereas the type `Parser` is a template, the type `Matcher` is a regular class, depicted in Figure B.9, that is instantiated by passing a function pointer to its constructor.

To PolySPINner, a compatibility criterion is a C++ function of the form:

```
MatchResult *f(Class *, Class *)
```

with two arguments representing two `Class` objects to be compared, and a return type, `MatchResult`, representing the results of the comparison. Such a function is passed to the constructor to instantiate a `Matcher`, and invoked by PolySPINner via the method `Match`. Diverse compatibility criteria are supplied with PolySPINner, such as the structural equivalence criterion shown in Figure B.10. The functions `Match_Method_Iterator` and `Match_Parameter_Iterator` iteratively apply a user-supplied function to a list of methods or parameters, respectively, in the manner of Lisp's `mapcar` function [26]. These are supplied by PolySPINner to aid the interoperability designer or engineer in the creation of compatibility criteria.

B.3.1 Type `MatchResult`

The type `MatchResult`, shown in Figure B.11, is an abstract representation of a successful or failed match between two types. The focus of its interface is the `AddMatch` method, which stores information about the successful comparison of two methods. The complementary method `AddMismatch` stores information about a failed comparison of two methods.

If all method matches are successful, the corresponding `MatchResult` will be passed to a `Generator` object so it can create polylingually-callable methods. If some method matches are unsuccessful, the `MatchResult` will contain textual error messages explaining why each match failed, guiding the interoperability engineer to take corrective action.

B.3.2 Type `MethodMapping`

In a `MatchResult`, the result of comparing two methods is stored as an object of type `MethodMapping`, shown in Figure B.12. Its interface is very similar to that of `MatchResult`, including analogous `AddMatch` and `AddMismatch` methods for storing compatibility information about pairs of parameters.

```
MatchResult *Match_Structural(Class *c1, Class *c2) {
  MatchResult *result;
  if (c1->NumberOfMethods() != c2->NumberOfMethods()) {
    result = new MatchResult(c1, c2);
    result->SetFail("differing number of methods");
    return(result);
  }
  return(Match_Method_Iterator(c1, c2, c1->NumberOfMethods(),
                               Match_Method_Structure));
}


MethodMapping *Match_Method_Structure(Method *m1, int posm1,
                                      Method *m2, int posm2) {
  MethodMapping *result;

  if (m1->ReturnType() != m2->ReturnType()) {
    result = new MethodMapping(m1, posm1, m2, posm2);
    result->SetFail("return types differ");
    return(result);
  }

  if (m1->NumberOfParameters() != m2->NumberOfParameters()) {
    result = new MethodMapping(m1, posm1, m2, posm2);
    result->SetFail("differing number of parameters");
    return(result);
  }
  else
    return Match_Parameter_Iterator(m1, posm1, m2, posm2,
                                    m1->NumberOfParameters(),
                                    Match_Parameter_Structure);
  return result;
}
```

Figure B.10. Compatibility criterion for structural equivalence. The iterator functions `Match_Method_Iterator` and `Match_Parameter_Iterator` are not shown.

```
class MatchResult
{
// non-public members omitted
public:
  MatchResult(Class *c1, Class *c2);   // the constructor

  Class *Source();                     // first class to match
  Class *Target();                     // second class to match

  void SetFail(char *);                // designate failure
  Boolean Success();                   // did it succeed or fail?

  // Store a successful match between two methods.
  void AddMatch(Method *m1, int pos1, Method *m2, int pos2);
  void AddMatch(MethodMapping *);

  // Store an unsuccessful match, and a reason why it failed.
  void AddMismatch(Method *m1, int pos1, Method *m2, int pos2,
                   String why);
  void AddMismatch(MethodMapping *);

  MethodMapping *MethodMatch(int i);   // Retrieve a method match
  int NumMethodMatches();              // how many method matches?

  void Report();                       // Print the match results
};
```

Figure B.11. The abstract `MatchResult` type.

```
class MethodMapping
{
// non-public members omitted
public:
  MethodMapping();                    // constructors
  MethodMapping(Method *m1, int pos1, Method *m2, int pos2);
  MethodMapping(Method *m1, int pos1, Method *m2, int pos2,
                String why);

  Method *Source();                   // first method to match
  Method *Target();                   // second method to match

  int SourcePosition();               // index of first method
  int TargetPosition();               // index of second method

  void SetFail(char *);               // designate failure
  Boolean Success();                  // did it succeed or fail?

  // Store a successful match between two parameters.
  void AddMatch(Parameter *p1, int pos1, Parameter *p2, int pos2);
  void AddMatch(ParamMapping *);

  // Store an unsuccessful match, and a reason why it failed.
  void AddMismatch(Parameter *p1, int pos1, Parameter *p2, int pos2,
   String why);
  void AddMismatch(ParamMapping *);

  ParamMapping *ParamMatch(int);       // retrieve a method match
  int NumParamMatches();               // how many method matches?

  // Return the parameter in the other method that matches
  // my parameter number i.
  Parameter *GetMappedParamSource(int i);
  Parameter *GetMappedParamTarget(int i);
};
```

Figure B.12. The abstract `MethodMapping` type.

```
class ParamMapping
{
// non-public members omitted
public:
  ParamMapping::ParamMapping();      // constructors
  ParamMapping(Parameter *p1, int pos1, Parameter *p2, int pos2);
  ParamMapping(Parameter *p1, int pos1, Parameter *p2, int pos2,
       String why);

  Parameter *Source();               // first parameter to match
  Parameter *Target();               // second parameter to match

  int SourcePosition();              // index of first parameter
  int TargetPosition();              // index of second parameter

  void SetFail(char *);              // designate failure
  Boolean Success();                 // did it succeed or fail?
};
```

Figure B.13. The abstract `ParamMapping` type.

B.3.3 Type `ParamMapping`

Just as `MethodMapping` stores information about matching pairs of methods, type `ParamMapping` stores information about matching pairs of parameters, as shown in Figure B.13.

B.4 Generator

The `Generator` type represents a code generator that creates modified method bodies, so that objects may be accessed polylingually, and creates server programs to maintain objects in a polylingual environment. Like the `Parser` and `Matcher` types, `Generator` must be instantiated before use. The original type, shown in Figure B.14, is abstract and must be specialized according to two parameters:

- The programming language for which it will generate code.

```
// A subtype of Indent, which automatically handled text indenting.
class Generator : public Indent
{
// non-public members omitted
public:
  // Generate code into the given output filename.
  Generator(char *filename);
  ~Generator();

  // Generate comments.  Must be specialized per language.
  virtual void Comment(char *) = 0;
  virtual void CommentLine(char *) = 0;
  virtual void CommentHeading(char *);

  // Print a comment explaining that PolySPINner was here.
  void BroughtToYouBy();

  // Various printf-like for generating code.
  void Print(char *fmt, ...);
  void PrintLine(char *fmt, ...);
  void PrintPlain(char *fmt, ...);

  // Print a blank line.
  void Blank();

  // Generate a polyspun method body for this matched method.
  // Must be specialized per language per communication medium.
  virtual void Spin(char *sourcefile, MatchResult *mr) = 0;

  // Generate a main program.
  // Must be specialized per language per communication medium.
  virtual void MakeMain(char *sourcefile, MatchResult *mr,
                        ObjectTable *ot) = 0;
};
```

Figure B.14. The generic `Generator` type.

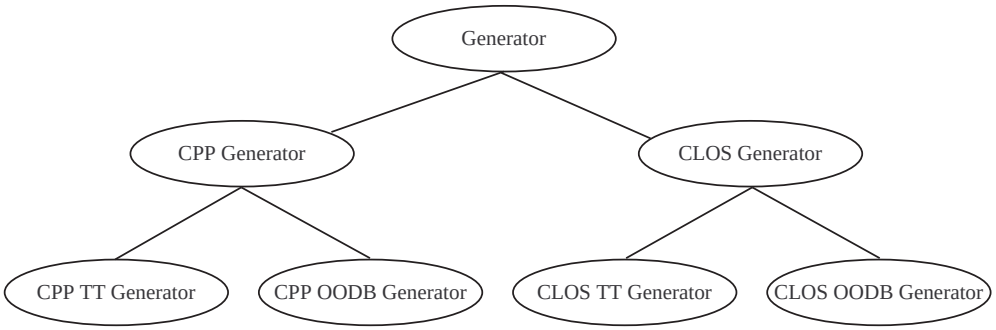


Figure B.15. `Generator` and its subtypes. `TT` means ToolTalk and `OODB` means Open OODB.

- The Communicator, or low-level communication substrate, used to implement interlanguage method calls.

Instantiation of a `Generator` is accomplished through subtyping, as shown in Figure B.15. For example, generators for C++ are of type `CPP_Generator`, and those that generate ToolTalk marshaling code are of type `CPP_TT_Generator`.

B.4.1 Language Mappings

The language-specific specializations of the generator type, `CPP_Generator` and `CLOS_Generator`, are designed to generate comments of the appropriate format for their respective languages, C++ and CLOS. In addition, `CPP_Generator` is aware of the existence and different purposes of .h (header) and .C (source) files, generating code for each. Both subtypes are still abstract and must be specialized to use a particular Communicator.

B.4.2 Communicator Mappings

B.4.2.1 Open OODB

Generators of types `CPP_OODB_Generator` and `CLOS_OODB_Generator` create polylingual types that interoperate via the Open Object-Oriented Database (Open

OODB), using foreign function calls within a single address space. The resulting system is capable of sharing persistent objects across languages.

The original PolySPINner V0 generated code that interoperated via Open OODB version 0.2, and PolySPINner V1 and V2 did likewise. Over time, our laboratory has upgraded to Open OODB 1.0 and has updated several other key pieces of software: SunOS (from 4.0 to 5.5), the Sun CC compiler (from 2.0 to 4.0), and Lucid Common Lisp (from 4.1 to 4.2). Unfortunately, the OODB-based code generated by PolySPINner does not function properly in this new software environment, due to undocumented interactions between the OODB, operating system, and compilers. This was partly the inspiration to create the ToolTalk-based implementation of PolySPINner, described in the next section, which is fully functioning.

B.4.2.2 ToolTalk

Generators of types `CPP_TT_Generator` and `CLOS_TT_Generator` create polylingual types that interoperate via Sun ToolTalk remote procedure calls across multiple address spaces. The resulting applications are capable of sharing transient (non-persistent) objects across languages. PolySPINner converts method calls into message passing, marshaling and unmarshaling parameters to cross the language (and address space) barrier intact. This was not necessary in the OODB-based PolySPINner, which simply allowed C++ and CLOS objects to be stored and retrieved via the database. The ToolTalk-based implementation is described in detail in Chapter 10.

APPENDIX C

USING POLYSPINNER

C.1 Overview

The PolySPINner toolset has two parts. The first part consists of analysis and code generation tools. This includes **poly**, the main application that generates polylingually accessible types as explained in this dissertation, and two auxiliary programs, **cpptest** and **clostest**, that permit convenient testing of PolySPINner's C++ and CLOS parsers. The second part is the PolySPINner Support Package, which is used by applications generated by **poly**. It consists of a link library, **libpolyspin.a**; a collection of C++ header files and CLOS files to be included by polylingual applications; the programs **killservers** and **killobject** for shutting down ToolTalk servers; and **ttsend**, a diagnostic tool for sending arbitrary ToolTalk messages. This appendix describes how to build, modify, and use the PolySPINner toolset.

C.2 Locating PolySPINner

PolySPINner is located in the directory

```
/ccsl/share/SunOS5/PolySPINner
```

The directory contains the following subdirectories:

`bin`	Executable programs
`doc`	Documentation
`examples`	Example applications of PolySPINner
`include`	Header files for C++ applications
`lib`	Link libraries for C++ applications
`lisp`	Support files for Lisp applications
`src`	Source code of PolySPINner

When a relative directory name, such as `bin`, is mentioned in this appendix, it is assumed to be in the main PolySPINner directory given above.

C.3 Building PolySPINner

This section describes the building and installing of the main PolySPINner application, **poly**, and its two diagnostic utilities, **cpptest** and **clostest**, all found in the `src` directory. Section C.4 will describe how to build and install the PolySPINner Support Package.

C.3.1 Requirements

PolySPINner compiles under SunOS 5.5 (Solaris) on a Sun SPARCstation. In order to compile PolySPINner, the following programs must be in your search path:

- SPARCworks **CC** 4.0 or higher
- GNU **flex** 2.5.4 or higher
- GNU **bison** 1.24 or higher
- GNU **make** 3.74 or higher
- UNIX **rm** and **sed** (used in the Makefile)

In order to maintain or modify PolySPINner, you'll also need:

- UNIX **makedepend**
- Revision Control System (RCS)

C.3.2 PolySPINner Source Files

PolySPINner V3 consists of approximately 10,000 lines (220K) of code, not including comments.[1] Ninety percent is written in C++, and the remainder is written in Common Lisp, Bison, and Flex. The source files needed for compiling **poly**, found in the `src` directory, are listed in Table C.1.

C.3.3 The Makefile

To build the PolySPINner applications **poly**, **cpptest**, and **clostest**, simply type:

```
make
```

To install PolySPINner in a directory of your choice, set the value of `DESTDIR` in the Makefile to be the destination directory, and type:

```
make install
```

If PolySPINner is modified, the list of file dependencies in the Makefile should be updated with the following command before recompiling:

```
make depend
```

To delete all files generated by compiling PolySPINner, except for the template database, type:

```
make clean
```

[1] In comparison, PolySPINner V0 was 2800 lines long (72K).

Table C.1. PolySPINner's source code.

Filename	Class Defined	Description
Boolean.h	Boolean	A true/false type
checklist.h	Checklist	Note the presence/absence of items in a set
Class.h	Class	Language-neutral class representation
clos-class.h	CLOS_Class	Class.h, specialized for CLOS
clos-generator.h	CLOS_Generator	Generator.h, specialized for CLOS
clos-method.h	CLOS_Method	Method.h, specialized for CLOS
clos-oodb-generator.h	CLOS_Generator	clos-generator.h, specialized for Open OODB
clos-parameter.h	CLOS_Parameter	Parameter.h, specialized for CLOS
clos-parser.y	—	Bison parser for CLOS
clos-scanner.l	—	Flex scanner for CLOS
clos-tt-generator.h	CLOS_TT_Generator	clos-generator.h, specialized for ToolTalk
clostest.C	—	Main program for testing CLOS parser
CPointer.h	CPointer	Pointer information for C++ classes
cpp-class.h	CPP_Class	Class.h, specialized for C++
cpp-generator.h	CPP_Generator	Generator.h, specialized for C++
cpp-method.h	CPP_Method	Method.h, specialized for C++
cpp-oodb-generator.h	CPP_OODB_Generator	CPP_Generator, specialized for C++
cpp-parameter.h	CPP_Parameter	Parameter.h, specialized for C++
cpp-parser.y	—	Bison parser for C++
cpp-scanner.l	—	Flex scanner for C++
cpp-tt-generator.h	CPP_TT_Generator	CPP_Generator, specialized for C++
cpptest.C	—	Main program for testing C++ parser
filename.h	Filename	A file name
generator.h	Generator	Generic generator component
getoptions.h	—	Support functions for command-line options
hacks.h	—	Temporary hacks
indent.h	Indent	An output device with automatic indenting
languages.h	—	Language-specific #defines
List.h	List	Generic linked list template class
matcher.h	Matcher	Generic matcher component
matchlibrary.h	—	Library of compatibility criteria
matchresult.h	MatchResult	Output of the matcher
Method.h	Method	Language-neutral method representation
MMM.h	MMM	Missing Method Manager
ObjectTable.h	ObjectTable	Object instance names supplied by user
options.h	Options	Command-line options
Pair.h	Pair	Template for a 2-item tuple
Parameter.h	Parameter	Language-neutral parameter representation
parser.h	Parser	Generic parser component
poly.C	—	Main program, PolySPINner
primitives.h	—	Support functions for primitive types
scanner.h	Scanner	Generic scanner component
String.h	String	Strings
token.h	Token	Tokens for the Scanner
triple.h	Triple	Template for a 3-item tuple
tt-support.h	—	ToolTalk support #defines
version.h	—	Version string for PolySPINner

To delete absolutely all files generated during the compile, returning the source code to its virgin state, type:

```
make clobber
```

To check in all source files with RCS, type:

```
make ci
```

A second Makefile, called `Makefile.config`, also is found in the `src` directory. It contains global Makefile variables and targets used not only by the main Makefile but also by those of the PolySPINner Support Package. These variables are automatically imported; `Makefile.config` is not intended to be passed directly to **make** by the user.

C.4 Building the PolySPINner Support Package

This section describes how to build the PolySPINner Support Package, a body of code used by PolySPINner-generated applications. When PolySPINner generates code, that code must be compiled, and the Support Package includes header files and libraries needed for that compilation. They are kept in the `src/TT` (ToolTalk) and `src/OpenOODB` (Open OODB) directories. Selected files from these directories are copied to the `include`, `lib`, and `lisp` directories when PolySPINner is installed with `make install` (Section C.3.3).

C.4.1 ToolTalk Version

Table C.2 lists the source files that make up the ToolTalk version of the PolySPINner Support Package. They are located in the directories `TT/C++` and `TT/Lisp`. These source files produce the link library **libpolyspin.a** and the programs **ttsend**, **killservers**, and **killobject**.

Table C.2. PolySPINner Support Package source code.

Filename	Class Defined	Description
killobject	—	Script to terminate ToolTalk objects
killservers	—	Script to terminate ToolTalk servers
load-tooltalk.lisp	—	Loads ToolTalk files for Lisp in order
Name.h Name.lisp	Name	A name assignable to NameableObjects
NameServer.h NameServer.lisp	NameServer	Locator for ToolTalk objects
NameableObject.h NameableObject.lisp	NameableObject	Permits objects to have names
PolySPIN.h	—	Miscellaneous
ProgrammingLanguage.h ProgrammingLanguage.lisp	ProgrammingLanguage	Programming language names
tt-lisp-support.c	—	Additional ToolTalk functions for Lisp
tt.lisp	—	ToolTalk API for Lisp
tt_NameableObject.h tt-NameableObject.lisp	tt_NameableObject tt-NameableObject	NameableObject specialized for ToolTalk
ttsend.c	—	Program to send any ToolTalk message

To build the ToolTalk version of the PolySPINner Support Package, enter the `TT` directory and type:

```
make
```

To install the executables, header files, libraries, and so forth, type:

```
make install
```

Other **make** targets mentioned in Section C.3.3 are also valid here: `depend`, `clean`, and `ci`.

C.4.2 Open OODB Version

The Open OODB version of the PolySPINner Support Package is entirely different from the ToolTalk version, as it was developed entirely independently prior to this dissertation. The directory `OpenOODB` is a placeholder for an anticipated improved version that parallels the ToolTalk version in the `TT` directory.

The Open OODB version uses files found in the directories:

Table C.3. PolySPINner match kinds.

Match Kind	Meaning
`a`	Arbitrary
`e`	Exact
`i`	Intersection
`s`	Subset
`S`	Superset

`/ccsl/share/SunOS5/OpenOODB`

`/ccsl/share/SunOS5/exodus`

Each of these directories has a `src` subdirectory containing a Makefile.

Section C.6.2 explains how to use the Open OODB version.

C.5 Running PolySPINner

C.5.1 Command Line Syntax

PolySPINner is invoked from the UNIX command line as follows:

`poly` *[options] match_kind C++_files CLOS_file*

C++_files is the prefix of the C++ files to be read. A value of `myfile` implies that PolySPINner should read `myfile.C` and `myfile.h`. *CLOS_file* is the full name of the CLOS file to be read, i.e., `myfile.lisp`.

The value *match_kind* selects the match criterion used by PolySPINner. It may be one of those given in Table C.3. Exact match (`e`) requires the type names and structure to be identical. Subset (`s`) and superset match (`S`) require one type to have a subset/superset of the other's methods. Intersection match (`i`) requires the two types to have some but not all methods in common. Finally, arbitrary match (`a`) uses an interactive user interface to match methods and parameters.

Table C.4. PolySPINner command-line options.

Option	Meaning
`-b`	Enable Bison debug mode. May be combined with `-d` or `-D`.
`-d`	Enable debug mode, level II.
`-D`	Enable debug mode, level II. Implies `-d`.
`-g`	Insert diagnostics into generated code.
`-h`	Help mode: print a usage message and exit.
`-i`	Enable inheritance processing in the Matcher (default).
`-I`	Disable inheritance processing in the Matcher.
`-m`	Enable the Missing Method Manager.
`-p`	Disable automatic loading of primitive types.
`-o [filename]`	Read objects from object description file **filename**.

PolySPINner's command-line options are summarized in Table C.4. The simplest option is `-h`, which causes PolySPINner to print a help message, describing its usage, and then exit immediately. All other options and arguments are ignored in this case.

Four options are related to debugging PolySPINner. The `-d` option turns on level I debugging, causing PolySPINner to print diagnostic information on standard error (stderr). The `-D` option turns on level II debugging, which is a more verbose superset of the level I output. Among other things, every token read by PolySPINner's parsers will be echoed to stderr. The `-b` option turns on Bison debugging, causing the Bison-generated parsers to print information about their states and transitions.[2] The last debugging option, `-g`, causes PolySPINner to insert additional output statements into the code it generates, causing the modified classes to print diagnostic messages as they execute.

The `-o` option is for specifying an object definition file for PolySPINner to load, as described in Section 10.3.1. If no object definition file is specified, PolySPINner cannot generate main "server" programs for managing transient objects.

[2]The `-b` option has the same effect as setting Bison's internal `yydebug` variable equal to 1.

The remaining options affect the behavior of the Matcher. The `-i` and `-I` options enable and disable inheritance processing, respectively. When inheritance processing is enabled, as it is by default, PolySPINner flattens the inheritance hierarchy so that inherited methods will be considered by the Matcher. The `-m` option enables the Missing Method Manager, described in Section 7.4.1.1, to process any unmatched types. Finally, the `-p` option disables the automatic loading of primitive types. By default, PolySPINner automatically creates classes in its internal representation for all primitive types in the languages it supports, such as integers, reals, and strings. If `-p` is used, all primitives will be considered unknown types.

The auxiliary programs **cpptest** and **clostest** use exactly the same command-line options as in Table C.4. Otherwise, they read from standard input and write to standard output.

C.5.2 Run-time Behavior of PolySPINner

PolySPINner (the **poly** program) examines the types given as input. If the files contain more than one type, PolySPINner prompts the user to select one from each file. Then it operates on the selected types.

The **poly** program does not overwrite the files it processes. Instead, it creates new files with the prefix "`out-`" in their names. Suppose the **poly** program is invoked with the following arguments:

```
poly -o OBJECTS e phonebook addressbook.lisp
```

In this case, PolySPINner performs an exact match between the classes in the files `phonebook.C` and `phonebook.h`, and the classes in `addressbook.lisp`. The results are written to the files:

- `out-phonebook.h`: The modified version of `phonebook.h`.
- `out-phonebook.C`: The modified version of `phonebook.C`.

- `out-addressbook.lisp`: The modified version of `addressbook.lisp`.
- `out-main-phonebook.C`: Main server program that maintains C++ objects specified in the object definition file, `OBJECTS`.
- `out-main-addressbook.lisp`: Main server program that maintains CLOS objects specified in the object definition file, `OBJECTS`.

Section C.6 describes how these files are compiled.

C.5.3 Arbitrary Match

All the match criteria work automatically except for arbitrary match, which is interactive. After PolySPINner prompts the user to select two types, as explained in Section C.5.2, it guides the user through selecting methods and parameters to match. The user response is always one or two integers, identifying the types, methods, or parameters to match. For example, in Figure C.1, the user is asked for a pair of parameters to match. In the example, the response "1 2" indicates to match parameter #1 of the C++ method `Insert` (`char *name`) with parameter #2 of the CLOS method `add` (`name string`). A value of -1 (or any value less than zero) signals to PolySPINner that matching is finished. This user interface is primitive and should be replaced with a more powerful, graphical user interface.

C.6 Compiling and Running PolySPINner-Generated Code

C.6.1 ToolTalk

In order to build and run the ToolTalk-based examples, you will need:

- Liquid Common Lisp 4.2 or higher, with the packages `clos`, `c-to-ffi`, and `ansi-packages` loaded. You can create an appropriate executable image with the commands:

```
Select a pair of parameters:

   C++ method Insert:
   0.  Phonebook *this
   1.  char *name
   2.  char *number

   CLOS method add:
   0. (book address-book)
   1. (number string)
   2. (name string)

Match what with what? [ x<0 to end ]:  1 2
```

Figure C.1. PolySPINner prompts the user to select two parameters during an arbitrary match.

```
$ lisp
(load "clos")
(load "c-to-ffi")
(load "ansi-packages")
(disksave "clos" :full-gc t)
(quit)
```

This creates an executable called **clos** in the current directory. A premade executable resides in the Liquid Common Lisp directory,

```
/ccsl/share/SunOS5/liquid
```

- GNU gcc compiler[3]
- ToolTalk

[3]PolySPINner is compiled with SPARCworks CC, but the experiments of Chapter 11 were compiled with gcc.

In order to compile and run the experiments of Chapter 11, you will also need:

- CL-HTTP (`/ccsl/share/SunOS5/cl-http`)
- The GNU C++ library (installed to be accessible automatically to **gcc**).

C.6.2 Open OODB

In order to do Open OODB examples, you'll need:

- Liquid Common Lisp 4.2, as above
- SPARCworks CC 2.0 (not 4.0)[4]
- Open OODB 0.2
- The Exodus storage manager, as shown below

Support for the Open OODB is currently broken under Solaris and Open OODB 1.0 due to software environment changes, as explained in Section B.4.2.1. Here are the last known instructions that used to work, courtesy of Alan Kaplan. They work correctly under SunOS version 4 with Open OODB 0.2 on the machine "relativity."

1. Ensure that your environment variables are set as in Figure C.2.
2. Run **Format_PS**. This formats the persistent store for use by the name manager. According to Alan, "you only need do this once, or whenever you want to start fresh." If you later encounter the Lisp error "Trap: Interrupt: segmentation violation," you should have run **Format_PS** first.
3. Invoke the Exodus server. The script `/ccsl/share/SunOS4/oodb/do-oodb` accomplishes this, as shown in Figure C.3.

[4]Unknown internal changes from CC version 2.0 to 4.0 prevent CC 4.0 from working with PolySPINner-generated code.

```
O3DBROOT=/u/liberation/sdl/external/OpenOODB

O3DB_SYSDIR=/usr/local/OpenOODB

O3DB_EX_PATH=$O3DBROOT/bin\
:/usr/lang/SC2.0.1patch\
:/usr/lang/SC2.0.1patch/tools\
:/usr/lang\
:/u/liberation/sdl/external/exodus/release.2.2.ti/bin\
:/u/liberation/sdl/external/OpenOODB/src/NM/bin

PATH=${O3DB_EX_PATH}:${PATH}
```

Figure C.2. Environment variables needed for Open OODB version of PolySPINner.

4. In the `/u/liberation/sdl/external/OpenOODB/src/NM/test` directory is a test program. Briefly, it populates a binding space named `Persons` with three instance of a `Person` class (also defined in that directory). The important source files are `Person.h`, `Person.C`, `Populate.C`, and `Person_Lisp_Stub.c`. `Person_Lisp_Stub.c` defines some Lisp stubs so that `Populate` can operate without Lisp. However, polylingual calls are included in `Person.C`. To populate the store, run `Populate`.

5. In the `/u/liberation/sdl/external/OpenOODB/src/NM/lisp/person` directory is a test program for polylingual interoperability. The important source files are:

   (a) Symbolic links to the `Person` source files in the `test` directory (from step 4 above).

   (b) The file `person.lisp`, which contains a CLOS type compatible with the C++ `Person` type.

   (c) The object file `Person.o`, which will be loaded by Lisp to encapsulate the C++ `Person` type.

```
# Run the Open OODB under SunOS version 4.
# Make sure to assign SGDIR and EXDIR to existing directories
# where you have write privileges.
SGDIR=${HOME}/.lib/sgdir
EXDIR=$HOME/.lib/exodus

# More vars - you should not need to set these
PATH=${O3DBROOT}:${PATH}
OODBDIR=/usr/local/OpenOODB
EXBINDIR=/u/liberation/sdl/external/exodus/release.2.2.ti/bin
PORTNUM=8000
PORT="relativity:${PORTNUM}"

# Remove old O3DB and Exodus data files.
KILLFILES="${OODBDIR}/system_sg ${SGDIR}/system_sg ${SGDIR}/sg2 \
           ${EXDIR}/log"
for file in ${KILLFILES}
do
        if [ -f "${file}" ]
        then
                echo "Removing ${file}"
                /bin/rm -f "${file}"
        fi
done

# Create and format the Exodus log
echo "****** Formatting Exodus Log ******"
echo "$EXDIR/log" | ${EXBINDIR}/formatlog > /dev/null 2> /dev/null

# Fire up ESM
echo "****** Start Exodus Server ******"
(cd $EXBINDIR && sm_server -log ${EXDIR}/log -bufpages 192 \
     -port $PORTNUM -background yes ) &
echo "****** Sleep 100 ******"
sleep 100

# Install the O3DB data files
echo "****** Install O3DB Storage Groups ******"
install_o3db ${PORT} ${SGDIR} 2000000
create_sg    ${PORT} ${SGDIR} 25000000
```

Figure C.3. The `do-oodb` script.

(d) The files `person-defsys.lisp`, `load-person-object-code.lisp`, and `setup-person.lisp`, which are initialization files for the CLOS `Person` type.

6. Invoke the commands:

```
cd /u/liberation/sdl/external/OpenOODB/src/NM/lisp/person
clos
(load "person-defsys.lisp")
(load-gbb-module :person)
```

7. Look in the file `dbpoly.lisp`. Execute each command in the file within Lisp. Briefly, the polylingual program creates a CLOS person and then accesses several `Person` objects polylingually.

C.7 Extending PolySPINner

C.7.1 Restrictions

When modifying any of PolySPINner's Generator types, do not cause them to generate any identifer beginning with the string `__POLYSPIN`. PolySPINner reserves this prefix for internal use.

C.7.2 Creating a Match Criterion

Match criteria are stored in the file `matchlibrary.C` and their signatures are specified in `matchlibrary.h`. A match criterion is a C++ function of the form:

```
MatchResult *MyMatch(Class *, Class *)
```

Functions defined in `matchlibrary.C` that you may find useful are found in Figure C.4. The function `Match_Method_Iterator` iterates through the method lists of two classes, applying the specified function to corresponding pairs of methods (one

```
// Iterate through all methods of a pair of classes.
MatchResult *
Match_Method_Iterator(
  Class *c1,            // First class
  Class *c2,            // Second class
  int NumMethods,       // Both classes must have same # of methods
  MethodMapping *(*mf)(Method *, int, Method *, int))
                        // Function to apply to each pair of methods


// Iterate through all parameters of a pair of methods.
MethodMapping *
Match_Parameter_Iterator(
  Method *m1,           // First method
  int posm1,            // First method's position in the class
  Method *m2,           // Second method
  int posm2,            // Second method's position in the class
  int NumParams,        // Both methods must have same # of params
  ParamMapping *(*mf)(Parameter *, int, Parameter *, int))
                        // Function to apply to each pair of params
```

Figure C.4. Support functions from `matchlibrary.C` for creating match criteria.

from each class) in sequence. `Match_Parameter_Iterator` does the same for parameters in two methods. For example, in Figure B.10, these iterators are applied to two classes to determine if they are structurally equivalent. Other useful functions are also defined in `matchlibrary.C` for checking name equivalence, structural equivlance, and so forth.

In general, however, one may construct any C++ function with the above signature and use it as a match criterion, provided a valid `MatchResult` object is returned. The primary methods of `MatchResult` are:

- `AddMatch`: Record a successful match between methods.
- `AddMismatch`: Record an unsuccessful match and a textual reason for the mismatch.

- `SetFail`: Indicate a failed match. (Automatically set by `AddMismatch`.)

See the files `matchresult.[Ch]` for more details.

C.7.3 Creating a Language Binding

Given a language L:

1. Create match rules for primitive types of L to those of all other supported languages. Currently this is not implemented in a robust manner in PolySPINner, so it is found in `hacks.C`.

2. Define subtypes of `Class`, `Method`, and `Parameter`, if necessary, to support language features not covered by these types.

3. Create a parser for L that translates the language into the intermediate representation. Example parsers are found in `cpp-parser.y` and `clos-parser.y`, with their respective scanners in `cpp-scanner.l` and `clos-scanner.l`. The parser must fill a List (`List.h`) with pointers to `Class` objects representing the classes found in the input file(s). See the lists `TheCPPClasses` and `TheCLOSClasses` in the parser files `cpp-parser.y` and `clos-parser.y`.

4. In the PolySPINner Support Package, create the equivalent types in your language for `Name`, `NameableObject`, `NameServer`, and `ProgrammingLanguage`.

5. Create a subtype of `Generator` appropriate for your language. See the files `cpp-tt-generator.[Ch]` and `clos-tt-generator.[Ch]` for examples.

6. Update the Missing Method Manager to add support for your language. See the files `MMM.[Ch]`.

BIBLIOGRAPHY

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

[2] Andrew C. Myers, Joseph A. Bank, and Liskov, Barbara H. Parameterized types for Java. In *Proceedings of 24th ACM Symposium on Principles of Programming Languages* (Jan. 1997).

[3] Atkinson, Malcolm, Bancilhon, Francçis, DeWitt, David, Dittrich, Klaus, Maier, David, and Zdonik, Stanley. The object-oriented database system manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases* (Kyoto, Japan, Dec. 1989), pp. 1–18.

[4] Auerbach, Joshua, and Chu-Carroll, Mark C. The Mockingbird system: A compiler-based approach to maximally interoperable distributed programming. Research Report RC 20718, IBM, T.J. Watson Research Center, Yorktown Heights, New York, Feb. 1997.

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

[6] Bach, Maurice. *The Design of the UNIX Operating System*. Prentice-Hall, 1986, ch. 5.12, pp. 111–119.

[7] Baker, Sean. *CORBA Distributed Objects : Using Orbix*. Addison-Wesley Publishing Company, Inc., 1997.

[8] Barrett, Daniel J., Clarke, Lori A., Tarr, Peri L., and Wise, Alexander E. A framework for event-based software integration. *ACM Transactions on Software Engineering and Methodology 5*, 4 (Oct. 1996), 378–421.

[9] Barrett, Daniel J., Kaplan, Alan, and Wileden, Jack C. Automated support for seamless interoperability in polylingual software systems. In *SIGSOFT '96: Proceedings of the Fourth ACM Conference on the Foundations of Software Engineering* (San Francisco, Oct. 1996), David Garlan, Ed., pp. 147–155. Published as *SIGSOFT Notes 21*, 6 (Nov. 1996).

[10] Bershad, Brian N., Ching, Dennis T., Lazowska, Edward D., Sanislo, Jan, and Schwartz, Michael. A remote procedure call facility for interconnecting heterogeneous computer systems. *IEEE Transactions on Software Engineering SE-13*, 8 (Aug. 1987), 880–894.

[11] Birman, Kenneth P. The process group approach to reliable distributed computing. *Communications of The ACM 36*, 12 (Dec. 1993), 37–53.

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

[13] Blickstein, David. Personal communication, Apr. 1996. Employee, Digital Equipment Corporation.

[14] Boehm, B. W., and Scherlis, W. L. Megaprogramming. In *Proceedings of the DARPA Software Technology Conference* (Los Angeles, CA, Apr. 1992), pp. 63–82.

[15] Brockschmidt, Kraig. *Inside OLE*. Microsoft Press, 1995.

[16] Bruce, Kim M. Typing in object-oriented languages: Achieving expressiveness and safety. Technical report, Department of Computer Science, Williams College, Williamstown, MA, Sept. 1996.

[17] Buschmann, Frank, Meunier, Regine, Rohnert, Hans, Sommerlad, Peter, and Stal, Michael. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[18] Church, A. *The Calculi of Lambda Conversion*. No. 6 in Annals of Mathematical Studies. Princeton University Press, Princeton, NJ, 1941.

[19] Dahl, Ole-Johan, and Nygaard, Kristen. Simula—an Algol-based simulation language. *Communications of the ACM 9*, 9 (Sept. 1966), 671–678.

[20] Danforth, Scott, and Tomlinson, Chris. Type theories and object-oriented programming. *ACM Computing Surveys 20*, 1 (Mar. 1988), 29–72.

[21] Date, C. J. *An Introduction to Database Systems*, fifth ed. Systems Programming Series. Addison-Wesley Publishing Company, Feb. 1991.

[22] Davies, Byron, and Bryan Davies, Victoria. Patching onto the Web: Common Lisp hypermedia for the intranet. *Communications of the ACM 40*, 5 (May 1997), 66–69.

[23] Detmold, Henry, and Oudshoorn, Michael J. Responsibilities: A safe and flexible remote communication construct. Technical Report 94-12, Department of Computer Science, University of Adelaide, June 1994.

[24] Gerety, Colin. HP SoftBench: A new generation of software development tools. *Hewlett-Packard Journal 41*, 3 (June 1990), 48–59.

[25] Goldberg, Adele, and Robson, David. *Smalltalk-80—The Language and its Implementation*. Addison-Wesley, 1983.

[26] Graham, Paul. *ANSI Common Lisp*. Prentice Hall Series in Artificial Intelligence. Prentice Hall, Englewood Cliffs, NJ, 1996.

[27] Guttag, J. V., Horning, J. J., and Wing, J. M. The Larch family of specification languages. *IEEE Software 2*, 5 (1985).

[28] Halstead, R. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems 7*, 4 (Oct. 1985).

[29] Hannah, Michael J. *HTML Reference Manual*. Sandia National Laboratories, Sept. 1995.

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

[31] Heimbigner, Dennis, and McLeod, D. A federated architecture for information management. *ACM Transactions on Office Information Systems 3*, 3 (July 1985), 253–278.

[32] Hopcroft, John E., and Ullman, Jeffrey D. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company, Inc., Reading, MA, 1979.

[33] Howell, David C. *Fundamental Statistics for the Behavioral Sciences*, third ed. Duxbury Press, Belmont, CA, Jan. 1995.

[34] Insight. Insight Business Edition. Product catalog 40, Insight Direct, Tempe, Arizona, Aug. 1997.

[35] Jackson, Daniel. Structuring Z specifications with views. *ACM Transactions on Software Engineering Methodology 4*, 4 (Oct. 1995), 365–389.

[36] Janssen, Bill, Severson, Denis, and Spreitzer, Mike. *ILU Reference Manual*. Xerox Corporation, Mar. 1995. Documents ILU version 1.8.

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

[38] Jones, Michael B., Rashid, Richard F., and Thompson, Mary R. Matchmaker: An interface specification language for distributed processing. In *12th ACM Symposium on Principles of Programming Languages* (Jan. 1985).

[39] Julienne, Astrid, and Russell, Larry. Why you need ToolTalk. *SunEXPERT Magazine* (Mar. 1993), 51–58.

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

[41] Kaplan, Alan, and Wileden, Jack. Toward painless polylingual persistence. In *Proceedings of the Seventh International Workshop on Persistent Object Systems* (Cape May, NJ, May 1996).

[42] Kaplan, Alan, and Wileden, Jack C. PolySPIN: Support for polylingual persistence, interoperability and naming in object-oriented databases. Technical Report TR-96-4, University of Massachusetts, Computer Science Department, Jan. 1996.

[43] Konstantas, Dimitri. Object oriented interoperability. In *ECOOP '93—Object-Oriented Programming, 7th European Conference* (Kaiserslautern, Germany, July 1993), Oscar M. Nierstrasz, Ed., no. 707 in Lecture Notes in Computer Science, Springer-Verlag, pp. 80–102.

[44] Lea, Doug. *User's Guide to the GNU C++ Class Library*, 2.0 ed. Free Software Foundation, Inc., Cambridge, MA, Apr. 1992.

[45] Lerner, Barbara Staudt. TESS: Automated support for the evolution of persistent types. In *Proceedings of the 12th Automated Software Engineering Conference* (Lake Tahoe, Nevada, November 1997).

[46] Lippman, Stanley B. *C++ Primer*. Addison-Wesley Publishing Company, 1990.

[47] Liskov, Barbara, Bloom, Toby, Gifford, David, Scheifler, Robert, and Weihl, William. Communication in the Mercury system. Tech. Rep. 59–1, MIT Programming Methodology Group, 1988.

[48] Liskov, Barbara, and Shrira, Liuba. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the 1988 Conference on Programming Language Design and Implementation (SIGPLAN)* (Atlanta, Georgia, June 1988), pp. 260–267.

[49] Liskov, Barbara H., and Wing, Jeannette M. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems 16*, 6 (Nov. 1994), 1811–1841.

[50] MacBride, Andrew, and Susser, Joshua. *Byte Guide to OpenDoc*. Osborne (McGraw-Hill), Berkeley, CA, 1996.

[51] Mallery, John C. A Common Lisp hypermedia server. In *Proceedings of The First International Conference on The World-Wide Web* (Geneva, Switzerland, May 1994), CERN.

[52] Maybee, M. J., Heimbigner, D. M., and Osterweil, L. J. Multilanguage interoperability in distributed systems: Experience report. In *Proceedings of the 18th International Conference on Software Engineering* (Berlin, Mar. 1996).

[53] Milner, Robin. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17*, 3 (Dec. 1978), 348–375.

[54] Nuseibeh, Bashar, and Finkelstein, Anthony. ViewPoints: A vehicle for method and tool integration. In *Proceedings of the 1992 IEEE International Workshop on Computer Aided Software Engineering* (Montreal, Canada, July 1992), pp. 50–60.

[55] Object Management Group. *The Common Object Request Broker: Architecture and Specification.* Object Management Group and X/Open, Framingham, MA, 1996. Revision 2.0, July 1995, updated July 1996.

[56] Object Management Group. *CORBAfacilities.* Object Management Group, Framingham, MA, 1997.

[57] Object Management Group. *CORBAservices: Common Object Services Specification.* Object Management Group, Framingham, MA, Mar. 1997.

[58] Palsberg, Jens, and Schwartzbach, Michael I. *Object-Oriented Type Systems.* John Wiley & Sons, 1994.

[59] Perrochon, Louis, Wiederhold, Gio, and Burback, Ron. A compiler for composition: CHAIMS. In *Proceedings of the Fifth International Symposium on Assessment of Software Tools and Technologies* (Pittsburgh, PA, June 1997).

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

[61] Reiss, Steven P. Connecting tools using message passing in the FIELD environment. *IEEE Software 7*, 4 (July 1990), 57–67.

[62] Ridgway, John V. E., Thrall, Craig, and Wileden, Jack C. Toward assessing approaches to persistence for Java. In *Proceedings of the Second International Conference on Persistence in Java* (Half Moon Bay, CA, Aug. 1997).

[63] Rundensteiner, Elke A. MultiView: A methodology for supporting multiple views in object-oriented databases. In *Proceedings of the 18th Conference on Very Large Databases* (Vancouver, Canada, Aug. 1992).

[64] Sánchez-Ruíz, Arturo J. Type models: Toward unifying concepts for language interoperability. In *Proceedings of the 22nd Latin-American Conference on Informatics (PANEL)* (Bogotá, Colombia, June 1996), pp. 541–552.

[65] Sánchez-Ruíz, Arturo J., and Glinert, Ephraim P. Multi-language programming: An automatic-type-mapping approach. In *Proceedings of the 16th IEEE Conference on Computer Software and Applications* (Chicago, IL, Sept. 1992), pp. 57–62.

[66] Scott, D. S., and Strachey, C. Towards a mathematical semantics for computer languages. In *Symposium on Computers and Automata* (New York, NY, 1971), Polytechnic Press, pp. 19–46.

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

[68] Silberschatz, Abraham, Peterson, James L., and Galvin, Peter B. *Operating Systems Concepts*, third ed. Addison-Wesley Publishing Company, Sept. 1991, ch. 3, pp. 66–67.

[69] Stallman, Richard. The GNU manifesto. Online documentation provided with the GNU Emacs editor, 1993. To display, type ctrl-h ctrl-p.

[70] Sun Microsystems. RPC: Remote procedure call protocol specification. Tech. Rep. RFC-1057, Sun Microsystems, Inc., June 1988.

[71] Ullman, Jeffrey. *Principles of Database Systems*. Computer Science Press, 1982.

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

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

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

[75] Zaremski, Amy Moormann, and Wing, Jeannette M. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology 4*, 2 (Apr. 1995), 146–170.

[76] Zaremski, Amy Moormann, and Wing, Jeannette M. Specification matching of software components. In *Proceedings of the Third ACM SIGSOFT Symposium on Foundations of Software Engineering* (Oct. 1995), Gail Kaiser, Ed., pp. 6–17. Also appeared as Carnegie-Mellon technical report CMU-CS-95-127.