Reengineering Large-Scale Polylingual Systems

Mark Grechanik, Dewayne E. Perry, and Don Batory
University of Texas at Austin
Austin, Texas 78712
{gmark|batory}@cs.utexas.edu, perry@ece.utexas.edu

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

We present a practical and effective process to reengineer large-scale polylingual systems. We offer a programming model that is based on the uniform abstraction of polylingual systems as graphs and the use of path expressions for traversing and manipulating data. This model enables us to achieve multiple benefits, including coding simplicity and uniformity (where neither was present before) that facilitate further reverse engineering. By performing control and data flow analyses of polylingual systems we infer the schemas used by all participating programs and the actions performed by each program on others. Finally, we describe a tool called FORTRESS that automates our reverse engineering process. *The contribution of this paper is a process that allows programmers to reverse engineer foreign type systems and their instances semiautomatically at the highest level of design.* We know of no other approach with comparable benefits.

1 Introduction

Building software systems from existing applications is a well-accepted practice. Applications are often written in different languages and provide data in different formats. An example is a C++ application that parses an HTML-based web page, extracts data, and passes the data to an EJB program. We can view these applications in different ways. One way is COTS integration problem where a significant amount of code is required to effect that integration. Or we can view them as instances of architectural mismatch, specifically as mismatched assumptions about data models [1]. Or we can view them, as we do in this paper, as instances of *polylingual interoperable* [2][3] applications that manipulate data in *foreign type systems (FTSs)* i.e., type systems that are different from the host language.

Consider an architecture for polylingual systems as shown in the directed graph in Figure 1. Graph nodes correspond to programs P_1, P_2, …, P_n that are written in different languages and may run on different platforms. Each edge $P_i \rightarrow P_j$ denotes the ability of program P_i to access objects of program P_j. $P_i \rightarrow P_j$ is usually implemented by a complex API that is specific to language of the calling program P_i, the platform P_i runs on, and the language and platform P_j to which it connects. (In fact, there can be several different tools and APIs that allow P_i to access objects in P_j). Note that the APIs that allow P_i to access objects in P_j may be different than the APIs that allow P_j to access objects in P_i.

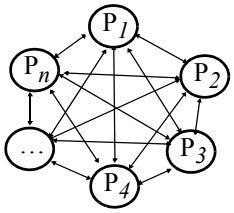

Figure 1: Architecture of Polylingual Systems

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

We propose a combined forward and reverse engineering process consisting of four steps to reverse engineer foreign type systems and their instances semiautomatically at the highest level of design. First, we offer a programming model that is based on abstracting constituents of polylingual systems as graphs of objects and providing language-neutral specifications based on path expressions, coupled with a set of basic operations, for traversing these graph to access and

manipulate their objects [4]. This step allows us to achieve multiple benefits, including coding simplicity and uniformity that facilitate further reverse engineering. Next, we perform control flow and data flow analyses of polylingual programs in order to infer schemas (e.g. a schema is a set of artifact definitions in a type system that defines the hierarchy of elements, operations, and allowable content) and actions performed by FTSs. Then we transform the schema and actions into plain English description. Finally, we implement tool called *FORTRESS (FOReign Type Reverse Engineering Semantic System)* that partially automates our reverse engineering process.

The main contribution of this paper is a combined forward and reverse engineering process based on a programming model that enables programmers to recover high-level design from complex polylingual systems with a high degree of automation.

2 The Reengineering Process

We propose a reengineering process that enables programmers to recover a high-level design of polylingual systems with a high degree of automation. The first and most important step on this process is the use of our programming model that normalizes FTS-based code into structured sequences of operations on type graph objects. These operations are provided by reification operator objects that are defined in *Reification Object-Oriented Framework (ROOF)* [4]. Program statements that contain reification operator objects are called *reification statements*. Our model reduces the complexity of polylingual programs by improving code simplicity and uniformity.

We achieve these results as naturally obtained benefits of normalized FTS-based programs. Our programming model significantly reduces the number of syntactical constructs that otherwise must be present in polylingual code, and the normalized programs have much simpler semantics.

The reverse engineering process is illustrated in Figure 2. After the code is normalized, the next step in the reverse engineering process is to use control and data flow analyses on the simplified polylingual code. *Control flow analysis (CFA)* relates the static program text to its possible execution sequences [5]. *Data flow analysis (DFA)*, on the other hand, computes relationships between data objects in programs [6]. We apply the CFA to build graphs of possible execution sequences of polylingual programs, and then we run the DFA on each execution graph to analyze FTS reification statements. The results of program analysis is used to infer schemas that describe the FTS models and operations executed against them by polylingual programs. The schemas and operations fed into high-level design description driver as shown in Figure 7 that produces plain English description of and visualizes structural and behavioral aspects of polylingual systems.

Finally, we describe the FORTRESS tool that enables programmers to reverse engineer polylingual systems using our process, browse high-level descriptions of FTS-based systems, visualize their impacts on schemas and their instances, and map elements of these descriptions and schema definitions to the source code.

3 ROOF Programming Model

ROOF is designed in light of principles of interoperable polylingual systems described in [2][3][4]. The goals of the ROOF programming model is to enable easily maintainable and evolvable polylingual interoperability by removing the need for elaborate name management solutions and allowing programmers to make decisions about sharing objects at the megaprogramming stage. The maintainability and evolvability of polylingual systems are achieved by using foreign objects by their names as they are defined in FTSs thereby eliminating the need for creation of isomorphic types in a host programming language and enabling programmers to share objects at the megaprogramming stage. We also provide a comprehensive mechanism for type checking that allows programmers to verify semantic validity of operations on foreign types both statically and dynamically with certain limitations.

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

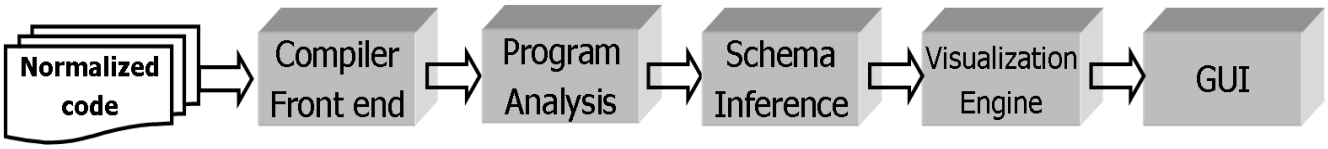

Figure 2: The illustration of steps in reengineering process.

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

`R.GetObject(`t_1`)...GetObject(`t_k`)`

`R` implements a *reification operator (RO)* that provides access to objects in a graph of foreign objects. We give all ROs the same interface (i.e., the same set of methods) so that its design is language independent; reification operators possess general functionality that can operate on type graphs of any FTS. By implementing `R` as an object-oriented framework that is extended to support different computing platforms, we allow programmers to write polylingual programs using a uniform language notation without having to bother about peculiarities of each platform. That is, for Java we have separate extensions of the framework that allows Java programs to manipulate C# objects, another extension to manipulate XML documents, etc. Similarly, for C# we have separate extensions of an equivalent framework that allows C# programs to manipulate Java objects, another extension to manipulate XML documents, etc. *Foreign Object REification Language (FOREL)* is a user interface provided by ROOF to enable programmers to write interoperable polylingual programs.

4 Program Analysis

Assuming that polylingual programs are normalized to conform to the ROOF programming model, we can perform program analysis as the next step in our reverse engineering process. Since programs that constitute polylingual systems communicate by changing each other's data and structures, program analysis allows us to recover these changes from normalized polylingual code. Thus, program analysis is an integral part of our reverse engineering process whose goal is to produce a high-level description of these changes.

We analyze programs in two steps. First, we run control flow analysis (CFA) to build execution graphs, and then we perform data flow analysis (DFA) on these graphs to compute relationships between data objects in polylingual programs [5][6].

DFA is the most significant part of a program analysis. We are interested in finding all definitions and uses of reification operator objects. There are three types of statement that can use RO objects:

- Navigation statements in which RO objects point to a certain object in the FTS type graph. For example, statement `R["CEO"]["CTO"]` denotes a collection of objects of type `CTO` contained in type `CEO`.
- Assignment statements in which values of type objects are set or retrieved. For example, the following statement `R["CEO"]["CTO"]<<5.0` sets the value of an object of type `CTO` to `5.0`.
- Structural statements that invoke operations that modify the structure of other polylingual programs. Given two RO objects `R` and `Q` the structural statement has a form of `R` $\otimes$ `Q` where $\otimes$ is a structural operation, for example, append a new object from RO `Q` to a branch in a type graph represented by RO `R`.

DFA enables us to answer the following five questions.

- What is the structure of FTSs operated upon by the analyzed program?
- How do FTSs affect the control flow of the analyzed program?
- What is the relationship between RO objects and program variables?
- What are the operations performed by the analyzed program on foreign type objects?
- What are the operations performed by the analyzed program on the structure of FTSs?

Answering these questions allows us to recover a high-level design from polylingual code. By determining the structure of FTSs we present a unified schema of the system. This operation is sound but not complete since we recover only a part of the schema that is operated upon by polylingual code. However, in the majority of cases the complete schema does not exist for a variety of reasons (e.g., it has never been created or it is rendered obsolete), and a part of a schema that describes type definitions operated on by polylingual code is a good enough approximation.

Knowing the structure of FTSs can help to determine how they affect the control flow of the analyzed program. Consider the following segment of FOREL code that contains a reification statement.

```
if( R["CEO"].Count() == 1 ){...}
else{...}
```

Suppose that we retrieve a schema of the FTS designated by `R`. If we establish that the number of `CEO` objects is always equal to one then we do not have to analyze the `ELSE` branch of the `IF` statement since the boolean condition is always true.

Unfortunately, we cannot limit the DFA to the analysis of RO objects. Type names and values of type objects can be assigned to program variables. Tracking uses and definitions of these variables may lead to establishing more complete schemas describing FTSs and improving the analysis of poly-

lingual programs subsequently raising the quality of the recovered high-level design.

The answers to the last two questions can be derived from the type of a reification statement. Navigation statements give us path expressions on type graphs. Assignment statements coupled with the knowledge of relationships between RO objects and program variables tell us what values we assign to or retrieve from destination type objects and what path expressions navigate to them.

Recall that structural reification statements modify the structure of FTSs, and such operations are reduced to modifications of abstract type graphs. Since the ROOF provides a unified set of operations on type graph objects we can detect them using elementary DFA algorithms.

5 Schema Inference

Schema inference is the process of deriving structure information from the query that generated data. This process is more complex than schema extraction that finds the most specific schema for a particular data instance [7]. Some languages make it easy to infer schemas just by looking at a statement that access or modifies data. Consider the following SQL statement.

```
SELECT u.Name, c.Course FROM User u,
Courses c WHERE u.ID = c.ID;
```

Just by looking at this statement we can infer the following information. There are

- two tables: `User` and `Courses`;
- attributes `Name` and `ID` in `User` table;
- attributes `Course` and `ID` in `Course` table;
- declaration of attribute `ID` in both tables is the same or compatible.

A diagram for the inferred schema is shown in Figure 3. It allows programmers to view the design of a database at a high level.

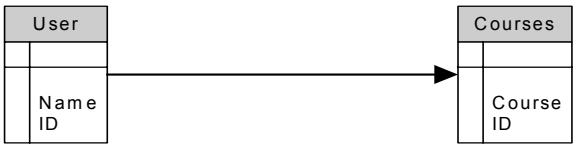

Figure 3: A schema diagram inferred from an SQL statement.

In general, polylingual code is suited very poorly for the schema inference process. Consider a fragment of C++ code shown in Figure 4.

We retrieve a handle to `CEO` node located in some XML data using the MS XML parser, COM and the Active Template Library. These tools and libraries are among the best available and are widely used today. The complexity of this fragment of code is clearly evident. Note that if we use a different XML parser (e.g., for improved performance), we must rewrite this code because their low-level APIs would be different. It is clear that automatic schema inference is very difficult and impractical for this and similar code. There should exist detailed information about the semantics of low-level APIs and their compositional semantics should also be defined. These conditions increase the complexity of program analysis algorithms significantly.

```
HRESULT hr = CoInitialize( NULL );
if( FAILED(hr) ) return( NULL );

CComPtr<IXMLDOMDocument> spDomDoc;
hr = spDomDoc.CoCreateInstance(
   __uuidof(DOMDocument40) );
if( FAILED(hr) ) return( NULL );

CComPtr<IXMLDOMNode> node;
node = spDomDoc;
BSTR b = AsciiToBSTR( "CEO" );
hr = node->selectNodes( b, &childList );
SysFreeString( b );
if( FAILED(hr) ) return( NULL );

CComPtr<IXMLDOMNode> nodeCEO;
hr = childList->get_item((long)0,&nodeCEO );
if( FAILED(hr) ) return( NULL );
```

Figure 4: A fragment of C++ code using COM and ATL.

As a result of our normalization process, the semantics of polylingual programs is much simpler than the ones of original programs. It is noteworthy that reification statements do not lead to increased complexity of program analysis algorithms that model this semantics. All high level design can be extracted directly from reification statements by analyzing their type and performing the DFA.

An additional benefit of inferring schemas from polylingual programs is the ability to compute relationships between FTSs that are manipulated by some polylingual program.

This concept is illustrated in Figure 5. Polylingual program `P` manipulates FTS_I and FTS_J using RO objects R_I and R_J. The interactions between program `P` and FTSs are shown with solid arrows. Suppose that R_I reads a value of some type

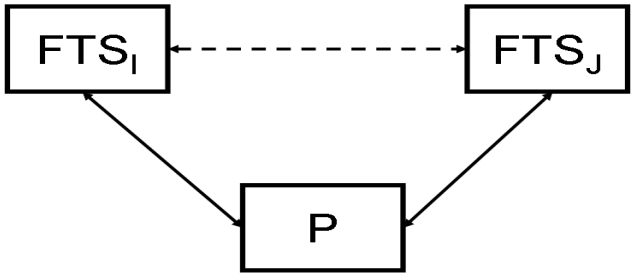

Figure 5: A computed relationship between FTSs.

```
if( R["CEO"]["CTO"]("Salary") > 100000 )
{
  if( R["CEO"]["CTO"]["Geeks"].Count() < 1 )
  {
          ............
  }

  if( R["CEO"]["CTO"]["Test"].Count() < 1 )
  {
          ............
  }
}
```

Figure 6: A fragment of FOREL code containing reification statements.

object from FTS_I and stores it in a program variable. Then, R_J assigns a value of this variable to some type object in FTS_J. By performing the DFA on program `P` we can compute the relationship between FTS_I and FTS_J as it is shown in Figure 5 with a dashed arrow.

Consider a fragment of FOREL code shown in Figure 6. This small fragment of code allows us to infer key definitions of some organizational schema. When analyzing the boolean condition of the first `IF` statement we can conclude that type `CEO` contains type `CTO` that has attribute `Salary` of primitive type `float`. When analyzing the boolean conditions of the second and third `IF` statements we infer that type `CTO` contains types `Geeks` and `Test`. We can also infer various schema constraints, for example, if a CTO makes more than $100,000 annual salary then there should be Geeks and Test departments in the company reporting directly to him.

6 FORTRESS

FORTRESS (FOReign Type Reverse Engineering Semantic System) is a framework for reengiineering and reverse engineering normalized polylingual applications. Figure 7 shows the architecture of FORTRESS. We envision FORTRESS as a framework for building a set of tools that help programmers to reengineer and reverse engineer polylingual systems and enable their effective evolution and maintenance.

The architecture reflects the steps of our reverse engineering process. Once an polylingual program is loaded into the tool we use a parser for the language in which this program is written to create its *abstract syntax tree (AST)*. Currently we support polylingual programs written in C++ and Java. We use EDG front end compilers [8] to parse source code and produce ASTs. Then we perform CFA and DFA on the AST to infer schemas of FTSs that this program accesses and manipulates. Finally, we transform operations on FTSs into plain English and present the listing of these operations in the FORTRESS GUI.

7 Related Work

We know of no existing technologies that fully address the problem of reverse engineering polylingual systems. A variety of reverse engineering tools has been reviewed in [9][10][11][12]. While they are effective in the problems that they intended to solve, none of the existing tools fully addresses the problem of reverse engineering software that consists of applications based on distinct and different type systems. Some tools are developed to reverse engineer multilanguage systems. For example, EER/GRAL approach to graph-based conceptual modeling is used to build models representing relevant aspects of single language [13]. These models are later integrated in a common conceptual model. The other paper [14] introduces GRASP - a software engineering tool designed to provide visualization of multilanguage software control structure, complexity, and architecture. Both approaches for reverse engineering multilanguage systems fall short of producing effective high-level designs by reverse engineering polylingual code.

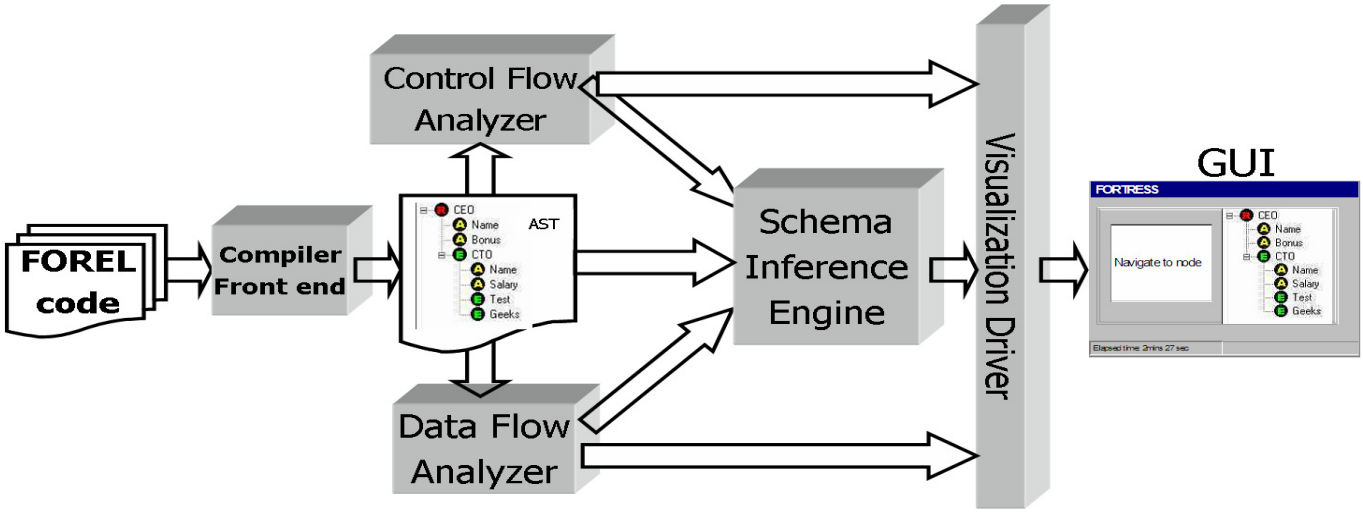

Figure 7: FORTRESS architecture.

Program comprehension techniques play important role in the normalization of source code for subsequent reverse engineering. Indeed, if a program is easy to understand by a human then it is likely to be effectively and automatically reverse engineered. Fundamental mechanisms of program comprehension are studied in [15][16].

8 Conclusions and Further Work

We have sketched a simple and effective way to reverse engineer polylingual systems. We accomplish this by introducing a semiautomatic process to reverse engineer polylingual systems and implement a tool called FORTRESS to automate this process. We offer a programming model that is based on the uniform abstraction of FTSs as graphs and the use of path expressions for traversing and manipulating data. This step allows us to achieve multiple benefits, including coding simplicity and uniformity, and to facilitate further reverse engineering. Next, we perform control flow and data flow analyses of polylingual programs in order to infer schemas describing all participating FTSs and actions performed by each FTS on others.

The contribution of this paper is a process that allows programmers to reverse engineer foreign type systems and their instances at the highest level of design automatically. We are working on a refactoring tool that would process legacy software and output the normalized polylingual code. If we are successful, we expect to make this functionality a part of FORTRESS.

We believe in practicality of our approach. The capability to write uniform and compact programs that work with applications based on different type systems enables better program comprehension and effective and automatic reverse engineering. As a result of our solution, developers concentrate on reasoning about recovered schemas that describe properties of applications without the need to understand low-level APIs and apply complex knowledge rules to recover high-level design from source code. Since the semantics of the reification languages is simple and easily parseable, it may enable various techniques and algorithms to improve reverse engineering process of ROOF-based source code. We know of no other approaches that achieve similar benefits or potential.

9 References

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

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

[3] A. Kaplan and J.Wileden, “Software interoperability: principles and practice,” *ICSE* 1999.

[4]M.Grechanik, D.Batory and D.Perry, "Design of Large-Scale Polylingual Systems," submitted to *ICSE* 2004.

[5]S.Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997.

[6]L. Moonen, “A Generic Architecture for Data Flow Analysis to Support Reverse Engineering,” *Second International Workshop on the Theory and Practice of Algebraic Specifications*, November 1997.

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

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

[9]R.Kollman, P.Selonen, E.Stroulia, T.Systä, and A.Zündorf, “A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering,“ *IEEE Ninth Working Conference on Reverse Engineering*, October-November 2002.

[10]A. van Deursen and L.Moonen, “Exploring Legacy Systems Using Types,“ *IEEE Seventh Working Conference on Reverse Engineering*, November 2000.

[11]T.Systä, “Understanding the Behavior of Java Programs,“ *IEEE Seventh Working Conference on Reverse Engineering*, November 2000.

[12]B.Bellay and H.Gall, “A Comparison of Four Reverse Engineering Tools,“ *IEEE Fourth Working Conference on Reverse Engineering*, October 1997.

[13]B.Kullbach, A.Winter, P.Dahm and J.Ebert, “Program Comprehension in Multi-Language Systems,“ *IEEE Fifth Working Conference on Reverse Engineering*, October 1998.

[14]T.Hendrix, J.Cross II, L.Barowski and K.Mathias, “Tool Support for Reverse Engineering Multi-Lingual Software,“ *IEEE Fourth Working Conference on Reverse Engineering*, October 1997.

[15]Y.Deng and S.Kothari, “Using Conceptual Roles of Data for Enhanced Program Comprehension,“ *IEEE Ninth Working Conference on Reverse Engineering*, October-November 2002.

[16]R.Clayton, S.Rugaber and L.Wills, “On the Knowledge Required to Understand a Program,“ *IEEE Fifth Working Conference on Reverse Engineering*, October 1998.