The Dissertation Committee for Mark Grechanik

certifies that this is the approved version of the following dissertation:

Design and Analysis of Interoperating Components

Committee:

Don Batory, Supervisor

Dewayne E. Perry, Supervisor

Kathryn S. McKinley

William R. Cook

Karl J. Lieberherr, Northeastern University

David Garlan, CMU

Design and Analysis of Interoperating Components

by

Mark Grechanik, M.Sc.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2006

To my grandmother Hanna, mother Elisa, and wife Tina

Acknowledgments

First and foremost, I would like to thank my wife, Tina, for her love and support. By the time I defend this dissertation, we will celebrate our fifteenth anniversary. We went through many important personal and global events such as the collapse of the Soviet Union, our immigration to the USA, and our personal and professional growth. We have overcome our difficulties, and they made us appreciate each moment we are together.

I shall be forever grateful to my academic fathers, Don Batory and Dewayne E. Perry. There are no words that can adequately describe their impact on me during the last five years. Their valuable advice helped me to mature as a researcher and to understand intricacies of scientific work. Don and Dewayne always acted in my best interests, and I think of them more as my friends than simply my advisors. I shall always remain deeply indebted to them!

I appreciate the encouragement and support from the other members of my committee, Kathryn S. McKinley, William R. Cook, Karl J. Lieberherr, and David Garlan. They were more than just readers of this thesis – they actively participated in my research. Their insight and advice helped me to improve my work and to come up with interesting solutions for complex problems. I also want to thank Michael Dahlin, James Browne, and Prem Devanbu for their help and advice.

Special thanks go to my mentor at the University of Texas at San Antonio, Kay A. Robbins. She was my first true teacher who taught me how to do research and think critically. Her advice and support extended well beyond my years at UTSA.

I want to thank my friends, Dmitry Gokhman and Richard Mankowski, for many wonderful hours we spent together. Tina and I are especially grateful to Dmitry for his help when we first came to the United States. It is always a pleasure to listen to Richard's nonstandard analyses of historic events based on his encyclopedic knowledge. I wish I could have a father like Richard!

I also want to thank my students whom I taught at the University of Texas at San Antonio and Texas State University at San Marcos over the past two years. Not only did they actively participate in my lectures and provided valuable feedback on the ideas that I presented to them, but also they took part in different case studies that I conducted at these universities. I wish good luck to all of them in their future endeavors.

Finally, I thank all my clients and employers for whom I have built various software systems. Not only did they teach me the value of good engineering solutions, but also paid handsomely for my work. In a way, this money supported my graduate work and allowed me to purchase tools for my research.

MARK GRECHANIK

The University of Texas at Austin

December 2006

Design and Analysis of Interoperating Components

Publication No. _____

Mark Grechanik, Ph.D.

The University of Texas at Austin, 2006

Supervisors: Don Batory and Dewayne E. Perry

Components are modular units (e.g., objects, modules, or programs) that interact by exchanging data. Two or more components interoperate when they exchange information [16]. It is conservatively estimated that the cost of programming errors in component interoperability just in the capital facilities industry in the U.S. alone is $15.8 billion per year. A primary driver for this high cost is fixing flaws in incorrect data exchanges between interoperating components [17]. Interoperating components are difficult to design, build, and evolve.

We propose an approach for the design and analysis of interoperating components. The core of our approach is an abstraction in which foreign objects (i.e., objects that are not defined in a host programming language) are abstracted as graphs and abstract operations access and manipulate them. These operations navigate to data elements, read and write data, add and delete data elements, and load and save data.

We offer different uses for the proposed abstraction. We build a framework called *Reification Object-Oriented Framework (ROOF)* which uses our abstraction to reduce the

multiplicity of platform API calls to a small set of operations on foreign objects that are common to all platforms [48]. With ROOF, we hide the tremendously ugly, hard-to-learn, hard-to-maintain, and hard-to-evolve code that programmers must write or generate today, i.e., we simplified code of interoperating components, making it scalable and easier to write, maintain, and evolve.

Our abstraction makes the task of static checking of interoperating components tractable by reducing the multiplicity of platform APIs to a small set of operations on foreign objects that are common to all platforms. By introducing a simple extension to grammars of object-oriented languages, we enable the collection of information about foreign objects at compile time. Static type checking uses this information to find possible errors that could otherwise be detected only at the runtime. We designed *Foreign Object REification Language (FOREL)*, an extension for object-oriented languages that provides a general abstraction for foreign component access and manipulation based on ROOF. FOREL type checking coupled with a conservative static analysis mechanism reports potential errors when referencing foreign components.

While ROOF and FOREL offer new approaches for developing interoperating components, many components are still written using low-level platform API calls. We use our abstraction to design and build a tool called a ***V**erifier for **I**nteroperating c**O**mponents for finding **L**ogic f**A**ults (Viola)* that finds errors in components exchanging XML data. Viola creates models of the source code of components by extracting abstract operations and computing approximate specifications of the data (i.e., schemas) that these components exchange. With these extracted models, Viola's static analysis mechanism reports potential errors for a system of interoperating components.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Components are modular units (e.g., objects, modules, or programs) that interact by exchanging data. Components are hosted on a platform, which is a collection of software packages. These packages export *Application Programming Interface (API)* functions through which components invoke platform services to access and manipulate data. For example, an *eXtensible Markup Language (XML)* [4] parser is a platform for XML data; it exports API functions that different components invoke to access and manipulate XML documents.

Two or more components interoperate when they exchange information [16]. It is conservatively estimated that the cost of programming errors in component interoperability just in the capital facilities industry[1] in the U.S. alone is $15.8 billion per year. A primary driver for this high cost is fixing flaws in incorrect data exchanges between interoperating components [17].

Software companies concentrate a lot of effort into software interoperability and developing interoperable solutions for customers [3]. The problem of making software

[1] A capital facility is a structure or equipment which generally costs at least $10,000 and has a useful life of ten years or more.

interoperable is difficult and pervasive. After giving a talk at Intel, I was approached by a manager who expressed his bewilderment at the complexity of this problem. He said that his programmers managed to write software components that were correct with respect to their specifications; however, these components did not function properly when integrated into a system. With utter frustration he exclaimed: "Why do I get a chemical compound with exact properties that I predicted from its constituent chemical components, and I cannot do the same with a system of interoperating software components?"

There is no silver bullet for software interoperability. The proliferation of different programming languages, operating systems and various virtual machines (e.g., CORBA, EJB, and .Net), databases and semistructured data (e.g., HTML and XML) parsers exacerbates the problem of seamless integration of software applications. Partial advances can be achieved through frameworks and static analysis techniques for finding errors in interoperating components. This thesis is a step towards building interoperating components with higher quality and at a lower cost.

1.1 A Motivating Example

We consider a primary mode of exchanging information for interoperating components by using XML data. An example is shown in Figure 1.1 in which fragments of Java (Figure 1.1a) and C++ code (Figure 1.1d) for two respective components that interoperate using XML data (Figure 1.1b-c). Block arrows show the flow of XML data between components. Variations of these code fragments are used in many open source and commercial applications. The Java component uses `Xerces` DOM parser API to read in and modify XML data that is shown in Figure 1.1b. This XML data describes the attributes of a book that include the author and title. The Java component modifies the structure of the XML

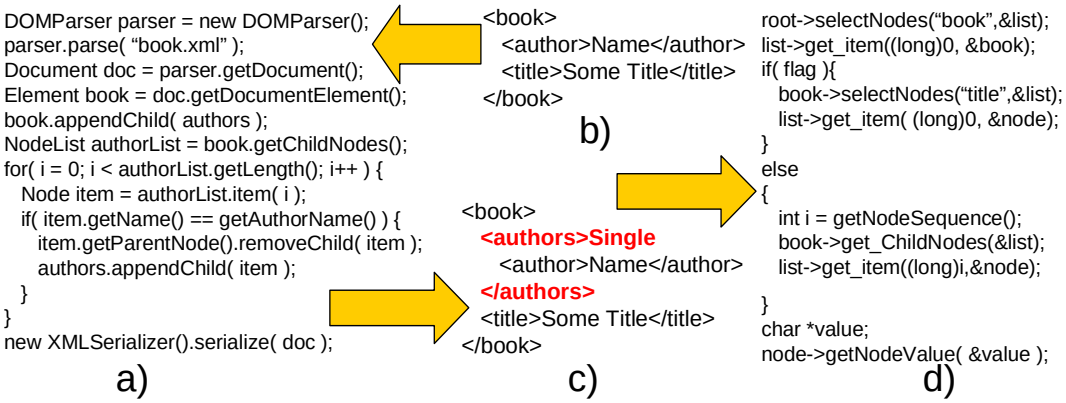


Figure 1.1: Java (a) and C++ (d) components that interoperate using XML data (b) and (c).

component by adding the tag `authors` as a child element of the root element `book` and moving the `author` element under the tag `authors`. The resulting XML data is shown in Figure 1.1c.

The C++ component shown in Figure 1.1d reads in the XML data shown in Figure 1.1b-c, and depending on the value of the boolean variable `flag`, returns the title or the name of the author of a book. The writer of this component assumes that a book has a single author, and the structure of XML data corresponds to the one shown in Figure 1.1b. When the Java component modifies this data, the C++ component code may throw a run-time exception because the element `author` is not present in the XML data under the root element `book`.

Currently, there are various projects that address this problem by making XML a first-class data type at the language level (e.g., XJ, XLinq, Xact, and Cω) [49][29]. While some success has been demonstrated, these projects have three major problems. First, they impose additional type systems and new coding practices on programmers, and these additions serve as inhibiting factors for their adoption. Second, for these languages to be sound

(i.e., to ensure the absence of bugs if the compiler reports no errors) programmers should not compute names of XML data elements at runtime. If names of data elements are not known at compile time, then type systems cannot be applied to ensure the absence of errors in interoperating components. This constraint limits programmers to a small class of applications. Third, given the large number of legacy systems that have been written using API calls exported by XML parsers, it is unlikely that these systems will be rewritten any time soon using these approaches.

In our example, schemas are not used to validate the XML data at runtime. If they were, then exceptions would be thrown during runtime validation of XML data either in the the Java component after it modified the data, or in the C++ component before it reads the data. If XML data is not validated at runtime, then exceptions will be thrown when certain API functions are called to access data elements. Either way, runtime errors occur whether XML data is validated or not.

Suppose that parser validators are not used. It is possible for an XML document to fail validation against a schema, however, components may never throw runtime exceptions. It happens when different interoperating components do not access and modify the same data elements (and do not validate XML data with its schemas). For example, the Java component modifies the element `author` by moving it as a child of the inserted element `authors`, and the path to the element `title` remains the same. Even though the XML data shown in Figure 1.1b and Figure 1.1c are different, the C++ component will not throw a runtime error when the value of its variable `flag` is `true` returning the title of the book.

Even with this simple example it takes a considerable amount of time to find errors. Several factors are involved: knowing the structure of the input data and how changes made by components affect it, using platform API calls correctly and translating API calls at com-

pile time into changes that would be made to XML data, and knowing the order in which components execute. The temporal dependency between the order of component execution and the visibility of errors makes catching errors especially difficult. If the C++ component executes before the Java component, then it would operate on the correct XML data shown in Figure 1.1b. However, if the Java component executes before the C++ component, then it would modify the data into an instance shown in Figure 1.1c, and thus make it incompatible for the C++ component. These factors add to the complexity of interoperating components, and make it difficult to catch errors at compile time.

1.2 A Model of Interoperability

We use a basic model shown in Figure 1.2 throughout this thesis. In this model, `J` and `C` are components (say a Java and C++ components respectively) that interact using XML data D_2. Component `J` reads in data D_1, modifies it, and passes it as data D_2 to the component `C`. Data is described using schemas, which are sets of artifact definitions in type systems that define the hierarchy of elements, operations, and allowable contents. Component `C` reads in the data D_2 expecting it to be an instance of some schema `S`. Since `J` outputs data D_2 before `C` accesses it, concurrency is not relevant. However, because of design or programming errors, the component `J` outputs the data D_2 as an instance of a different schema `S'`, which is not explicitly stated in any design document. Since `S'` is different from `S`, a runtime

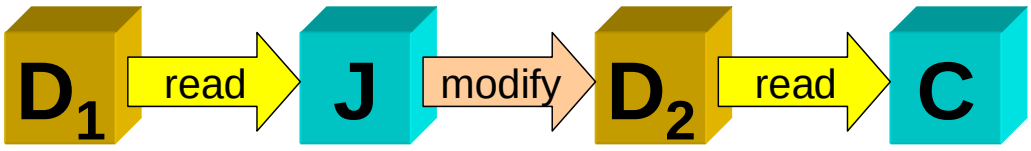


Figure 1.2: A model of component interoperability.

error may be issued when C reads in D_2.

There are different reasons why programmers make mistakes when they write the components J and C. Based on our participation in large-scale projects, we observe that programmers often make wrong assumptions about schemas. Given that many industrial schemas contain thousands of elements and types, it is easy to make mistakes about names of elements and their locations in schemas. Other source of errors lie in the complexity of platform API calls that programmers use to access and manipulate XML data. XML parsers export dozens of different API calls, and mastering them requires a steep learning curve.

Often, programmers lack the knowledge of the impact caused by changing the code of some component on other components that interoperate using XML data. This lack of knowledge is an effect of the Curtis' law that states that application and domain knowledge is thinly spread and only one or two team members may possess the full knowledge of a software system [37]. The effect of this law combined with the difficulty of comprehending large-scale XML schemas and high complexity of platform API calls result in components producing XML data that is incompatible for use by other components.

Another source of errors is the disparity in evolving XML schemas and components. Database administrators usually maintain schemas, and programmers maintain components that interoperate using XML data that should be instances of these schemas. If a database administrator modifies some schemas and does not inform all programmers whose components are affected by this change, then some components will keep modifying XML data according to the obsolete schemas.

1.3 A Big Picture

A big picture of this thesis is shown in Figure 1.3. The core of our approach is an abstraction in which foreign objects (i.e., objects that are not defined in a host programming language) are abstracted as graphs and abstract operations allow programmers to access and manipulate data. These operations are for navigating to data elements, reading and writing data, adding and deleting data elements, and loading and saving data, designated as `Navigate`, `Read`, `Write`, `Add`, `Delete`, `Load`, and `Save` respectively. We use these abstract operations as a basis for the framework and bug finding approaches proposed in this thesis.

This thesis offers two uses for the proposed abstraction. First, we build a framework called *Reification Object-Oriented Framework (ROOF)* which uses our abstraction to reduce the multiplicity of platform API calls to a small set of operations on foreign objects that are common to all platforms [48]. With ROOF, we hide some of the tremendously ugly, hard-to-learn, hard-to-maintain, and hard-to-evolve code that programmers must write or generate today, i.e., we simplify code of interoperating components, making it more scalable and easier to write, maintain, and evolve.

Second, our abstraction makes the task of static checking of interoperating components tractable by reducing the multiplicity of platform APIs to a small set of operations on foreign objects that are common to all platforms. By introducing a simple extension to grammars of object-oriented languages, we enable the collection of information about foreign objects at compile time. This information is used to perform static type checking in order to determine possible errors that could otherwise be detected only at the runtime. I designed *Foreign Object REification Language (FOREL)*, an extension for object-oriented languages, that provides a general abstraction for foreign component access and manipu-

lation based on ROOF. FOREL type checking coupled with a conservative static analysis mechanism reports potential errors when referencing foreign components.

While ROOF and FOREL offer new approaches for developing interoperating components, many components are still being written using low-level platform API calls. It is not likely that millions of lines of legacy software would be replaced in the near future using ROOF and FOREL (although we hope that it will!). In the meantime, we use our abstraction to design and build a tool called a ***V**erifier for **I**nteroperating c**O**mponents for finding **L**ogic f**A**ults (Viola)* that finds errors in components exchanging XML data. Viola creates models of the source code of components by extracting abstract operations and computing approximate specifications of the data (i.e., schemas) that these components exchange. With these extracted models, Viola's static analysis mechanism reports some potential errors for a system of interoperating components.

1.4 Scalability

Quality of the code of interoperating components and their scalability are critical for large-scale applications. Weaving interoperability into the fabric of enterprise-level architectures often decreases the scalability of the resulting system. In this section, we analyze the sources of nonscalability for systems of interoperating components.

We extend the model of component interoperability shown in Figure 1.2. Consider an architecture for systems of interoperating components as shown in the directed graph in Figure 1.4. Graph nodes correspond to components 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 the component P_i to access objects of some other component P_j. $P_i \rightarrow P_j$ is usually implemented by a complex API that is specific to the language of the calling program P_i,

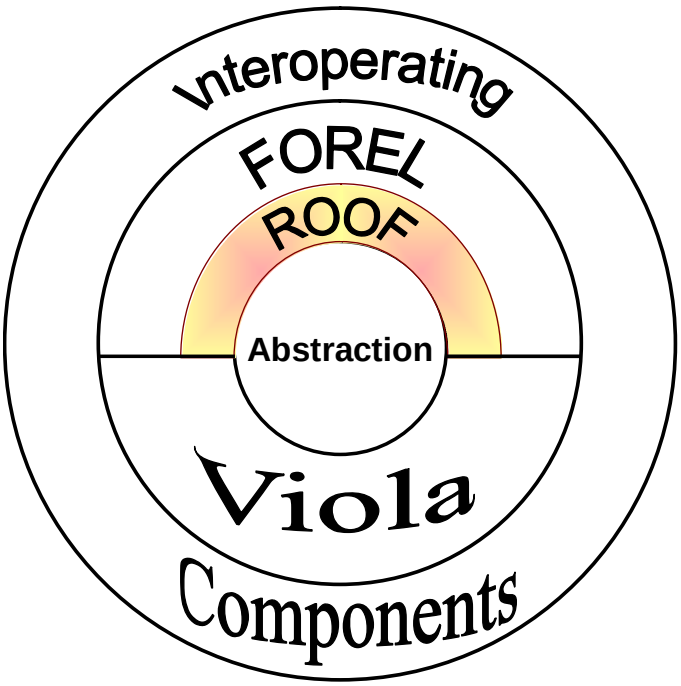


Figure 1.3: A big picture of this thesis.

the platform P_j 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.

The complexity of a system of interoperating components is approximately the number of edges in Figure 1.4 that it uses. That is, when the number of edges (i.e., platform APIs needed for interoperability) is minuscule, the complexity of a system of interoperating components 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 1.4), the complexity of a system of interoperating components 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 system of interoperating components is a system where the number of edges (API calls) is excessive. Such systems are notoriously difficult to develop, maintain, and evolve. Current approaches do not support large-scale systems of interoperating com-

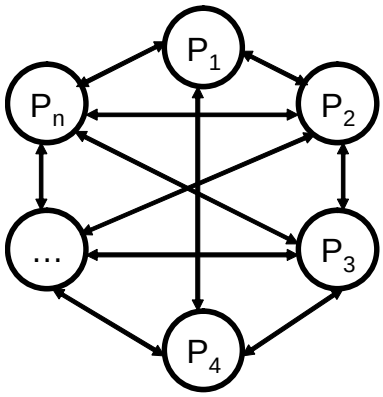


Figure 1.4: Architecture of a system of interoperating components.

ponents well. They are often limited to specific languages (e.g., typical CORBA platforms allow Java, C++, etc. programs to interoperate, but there are no facilities for accessing HTML or XML data or objects in C# programs). This leads to a proliferation in tools and their API calls, which noticeably increases the accidental complexity of the resulting code [30]. The loss of uniformity in the way programs are written renders resulting systems extremely difficult to maintain and evolve.

1.5 Safety Properties

Type checking algorithms enable proofs of the absence of certain program behavior statically [73]. However, there are many situations where the static type checking of interoperating components is not attempted, resulting in the run-time discovery of errors. For example, programmers may use platform API calls incorrectly in the component source code, and modify XML data so that it becomes incompatible for use by some other components. Currently, no tool checks interoperating components for potential flaws in their source code that lead to incorrect data exchanges and runtime errors, even when components are located within the same application.

The problem of mismatch between XML data and schemas is typically addressed by using schema validators that are parts of many XML parsers. In our model shown in Figure 1.2, an XML parser can validate that the data $\mathtt{D}_2$ is an instance of the schema `S` when `J` produces this data. If the data is not an instance of this schema, then the parser throws a runtime exception. Obviously, it is better to predict possible errors at compile time rather than to deal with them at runtime.

In reality, the situation is even more complicated. Using schemas for validating XML data is often not attempted because it degrades components' performance [74][67],

and it even leads to throwing exceptions when there may not be any runtime errors. Suppose that the component J deletes all instances of some data element thus violating the schema S that requires at least one instance of this element be present in D_2. If either of components J and C validates this incorrect data D_2 against the schema S, then a runtime error will be issued. However, when executed, the component C may never attempt to access the deleted data element, and therefore, no exception will be thrown if the validation step is bypassed. It is important to know what data elements components J and C access and modify, and if no data element accessed by C is modified by J, then components J and C may still interact safely even if the data D_2 is not an instance of the given schema S.

Although it is known in advance that components exchange data, it is not clear how to detect at compile time operations that lead to possible runtime errors. Using API calls exported by XML parsers remains the primary mode of XML access and manipulation. Various language extensions and type systems were proposed to address this problem [29][59][49]. Some of these approaches require programmers to map XML types to types from the proposed type systems, and that adds complexity to developing interoperating components. Other approaches propose type systems that are not sound or have constraints (e.g., structural modification to XML data are prohibited) that reduce their practicality. It is not clear whether these approaches are suitable for use in commercial or open source projects.

Many errors can be avoided if certain properties hold in components that interchange XML data. These properties are main and secondary safety properties. Given interoperating components J and C producing and exchanging data D_2 at runtime which is an instance of the schema S, the *main safety property (MSP)* is defined as ensuring that D_2 is an instance of S.

The *secondary safety property (SSP)* is defined as the same data elements in `S` should not be accessed by one and modified by some other interoperating components provided that specifications are not used at runtime to validate XML data. Since the MSP and SSP ensure stronger guarantees that no runtime exception will be thrown, using XML parsers to validate data against schemas is irrelevant to a problem of finding errors at compile time.

1.6 Contributions

The contributions of this dissertation are the following:

Abstraction. We propose an abstraction in which foreign objects (i.e. objects that are not defined in a host programming language) are abstracted as graphs and abstract operations allow programmers to access and manipulate data. These operations navigate to data elements, read and write data, add and delete data elements, and load and save data. These abstract operations are implemented in components using low-level platform API calls. We use these abstract operations as a basis for the framework and bug finding approaches proposed in this thesis.

Reification Object-Oriented Framework (ROOF). We define a framework based on the proposed abstraction that presents a single API for component interoperability. In ROOF we implement abstract operations by dynamically converting foreign objects into first-class host language objects so that we enable access to and manipulation of their instances. This is the concept of reification. Reification by reflection eliminates the need for generating potentially huge numbers of conversion classes, and allows us to access and manipulate semi-structured data that have no schemas.

Language Extension. We present *Foreign Object REification Language (FOREL)*, an extension for object-oriented languages that enables programmers to use abstract operations to enable interoperating components to exchange information. FOREL type checking coupled with static analysis algorithms enable developers to reason more effectively about interoperating components.

Bug Finding Tool. In order to ensure safety properties for interoperating components, we developed a ***V**erifier for **I**nteroperating c**O**mponents for finding **L**ogic f**A**ults (Viola)* that finds some errors in components exchanging XML data. Viola helps test engineers to validate reported errors. Viola is a helpful bug finding and testing tool whose static analysis mechanism reports potential errors for a system of interoperating components.

Empirical Validation. We implemented the ROOF framework, a Viola bug finding tool, and a FOREL compiler. We applied ROOF to a real-time component-based semiconductor overlay analysis and control system as well as conducted controlled experiments to evaluate how effective ROOF is. We tested Viola and FOREL on open source and commercial systems, and we detected a number of known and unknown errors in these applications with good precision thus proving the effectiveness of our approaches.

1.7 Thesis Statement

Our thesis can be stated as follows:

An effective and practical approach can be used to design, analyze, and implement systems of interoperating components, and this approach can lead to lightweight and prac-

tical frameworks and tools for creating large-scale systems of components and ensuring their safety properties.

The next chapter describes the ROOF and its implementation, and presents results of its evaluation. Chapter 3 presents FOREL, describes its type system, implementation, show the proof of the soundness of the FOREL type system, and give our type checking algorithm. Chapter 4 presents Viola, shows its architecture, and describes its implementation. Chapter 5 compares our approaches to related work, and Chapter 6 concludes this dissertation by discussing our solutions and mapping future work.

Chapter 2

Reification Object-Oriented Framework (ROOF)

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 shown in Figure 1.1a that parses XML data, and passes the data to an *Enterprise JavaBean (EJB)* program. We can view these applications in different ways. We can view them as *Components-Off-The-Shelf (COTS)* integration applications 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 [43]. Or we can view them, as we do in this thesis, as instances of interoperable [28] components that manipulate data in *foreign type systems (FTSs)* i.e., type systems that are different from the host language in which a component is written.

We showed in Section 1.4 that the complexity of a system of interoperating components with a clique-like architecture is approximately the number of edges in Figure 1.4 that

it uses. Object-oriented researchers have developed frameworks as a technique for eliminating this kind of complexity. A framework is an abstraction that underlies a number of similar programs, and is represented by a set of abstract classes. A particular implementation of this framework is a set of concrete classes that customizes the frameworks abstract classes for a designated application. The benefit of a framework is that it defines a single platform that all programmers can use; so instead of having $O(n^2)$ possible API platforms for achieving component-to-component communication, a single, standard, and clear API is used [1]. New framework implementations are easy to add, and consequently, this is a scalable approach.

In this chapter, we define a framework that presents a single API platform for the interoperability of components in a software system. Our idea is to abstract instances of an FTS as a graph of objects and to provide language-neutral specifications based on path expressions, coupled with a set of basic operations, for traversing this graph to access and manipulate its objects. We implement traversals by dynamically converting foreign types into first-class host language objects so that we enable access to and manipulation of their instances. This is the concept of *type reification*. Reification by reflection eliminates the need for generating potentially huge numbers of conversion classes, and allows us to access and manipulate semi-structured data that have no schemas. In this respect, we claim it is superior to existing approaches because it does not require programmers to generate potentially large number of corresponding types, explicitly define common interfaces using an IDL language, or use different low-level API platforms.We call our approach the *Reification Object-Oriented Framework (ROOF)*.

[1] We distinguish between platform APIs and API calls. The latter are methods exported by the former.

2.1 Principles of Interoperability

A framework for the analysis and design of interoperable systems is given in [55][80]. This framework has two objectives. First, developers should not be constrained in using the type system provided by the host programming language. For example, if a programmer wants to share a Java object, then (s)he should be able to do it directly in Java without resorting to some other type system such as an *Interface Definition Language (IDL)*. Second, the design of interoperating components should not be affected by a decision to share them. It means that the structure of classes and their interfaces should not be a function of sharing their instances.

Both these objectives define a foundational principle in the design of interoperating components, called *seamlessness* [80]. This principle states that developers of systems of interoperating components need not be aware of language differences between interoperating components. For example, interoperating components that include special platform-dependent functions that facilitate interactions between the host and foreign type systems, are not seamless. Violations of this principle of seamlessness lead to complex and nonuniform code that is difficult to understand and reason about and subsequently difficult to maintain and evolve.

When analyzing and designing interoperating components it is equally important to address three additional requirements. The first is naming. Sharing objects among FTSs often requires elaborate name management mechanism. Suppose that an XML type name (e.g., `friend` or `union`) is a keyword in a host programming language, for example, C++. Then this XML type name cannot be used directly when defining a corresponding class in C++. Even worse, if we have an XML and HTML schemas that have identical type names, then what is a naming scheme that resolves this ambiguity when defining corresponding

types to C++? Serious effort is dedicated to offering effective name management strategies, however, most carry a significant overhead.

A second requirement is the time when a decision is made to share objects. The decision time is defined by three intervals: *earliest time* when a decision to exchange information is made before any component is written, *common time* of making an information sharing decision when only a part of an interoperating component is developed, and *megaprogramming time* when information sharing decision is made after the entire system is developed. Clearly, allowing programmers to make components share information at the megaprogramming stage is both attractive and quite difficult because it requires changing existing code.

The third requirement is type checking of shared and native objects to ensure that they have compatible types. Different approaches that provide effective type checking can only be used at the earliest stage. Thus, the design and development of interoperating components is the science of trade-offs among objectives and concepts described above.

2.2 Our Solution

ROOF is designed in light of principles for systems of interoperating components described in Section 2.1. The goals of our solution is to enable easily maintainable and evolvable interoperability by removing the need for elaborate name management solutions and allowing programmers to make decisions about exchanging information at the megaprogramming stage. The maintainability and evolvability of interoperating components are achieved by using foreign components 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 exchange information 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, which is described in Chapter 3.

Our solution 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 [82], we limit the scope 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.

2.3 Type Graphs

Schemas can be represented as graphs whose nodes are composite types, leaves are primitive or simple types, and parent-child relationships between nodes or leaves defines a type containment hierarchy [21]. In order to operate on such graphs, a programmer must be able to reach nodes at arbitrary depth. This is accomplished via path expressions that are queries whose results are sets of nodes. A path expression is a sequence of variable identifiers or names of subordinate (or containment) types that define a unique traversal through a schema.

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 shown in Figure 2.1. `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`)`.

```
class ReificationOperator
{
    public:
    ReificationOperator &GetObject( string t );
    int Count( void );
};
...
ReificationOperator R;
```

Figure 2.1: Declaration and instantiation of ReificationOperator class in C++.

Consider a schema that describes the organizational structure of a company shown in Figure 2.2. It is a directed graph where each node is named after an organizational entity within a company and edges describe the subordination of one entity to the other. Each node has attributes shown as line connectors with filled circles followed by the names of the attributes. The `CEO`s subordinate is the `CTO` who in turn supervises two departments shown as `Test` and `Geeks`. An instance of this schema may be given in existing markup languages such as HTML, XML, or SGML.

We simplify the notation for the `ReificationOperator` class by introducing array access operators `[]` that replaces the `GetObject` method. For example, if `R` denotes an instance of the schema shown in Figure 2.2, we can write the C++ program that counts the number of employees in the `Geeks` department as:

```
int n = R["CEO"]["CTO"]["Geeks"].Count().
```

The method `Count` returns the number of child nodes under a given path. Such notation is useful since a single line of lucid code is used to replace a lot of hand-written or generated code. No additionally defined types and operations are required.

Path expressions symbolize simplicity and uniformity. These are the properties that we inherit from path expressions and they enable programmers to uniformly navigate

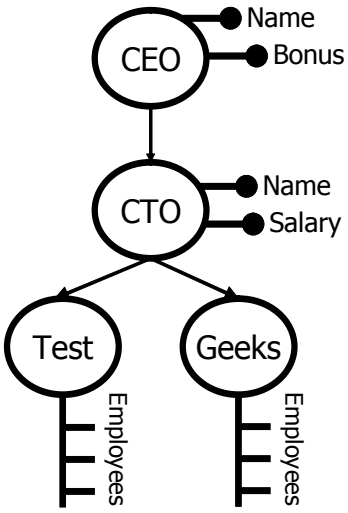


Figure 2.2: A schema of the organizational structure of a company.

through instances of foreign types. Further, since type names are used as they are defined in foreign schemas, there is no need to redefine them again in the host language. That is, we use the existing names of foreign types; we do not create corresponding types in the host language! We will show how this is accomplished shortly.

FTS-based components often change each other's structures. A common example is a C++ application that changes the structure of an XML document. These modifications are complex and require carefully crafted software called transformation engines. However, since all type systems can be represented as graphs, these modifications can be reduced to transformations on graph structures. Thus, we reduce the task of manipulating FTS structures to that of manipulating graphs. A comprehensive set of basic operations used to manipulate FTS graph structures includes copying or moving a node to its new location, appending and removing a node, and logic set and relational operations on graphs.

By implementing these operations using a standardized notation we achieve uniformity of FTS-based code. Indeed, interoperating components written in different languages

that perform the same operation on the same schema will look the same. This very important property of uniformity enables effective program evolution and maintenance of and automated reasoning about systems of interoperating components.

2.4 Reification

In this section we show how to reify types. We first explain reification concepts, then their abstraction as reification operators, and then present detailed notes on implementation.

2.4.1 Reification Concepts

Reflection. Reflection is a powerful and common mechanism in contemporary programming languages and programming infrastructures (e.g., reflection in virtual machines). Reflection exposes the type of a given object; it reveals the public data members, method names, type signatures of method parameters and results, and superclasses (if any) of an object's class. Further, reflection enables a program to invoke methods of objects whose classes were not statically known at the time the program was compiled. It also allows a program to navigate a graph of interconnected instances without statically knowing the types of these objects. All of this information and power is available to a program at runtime.

Connectors. A *reification connector (RC)* is an example of an architectural connector [72][71]; it is a communication channel between a host language application and an application with a foreign type system. At the host language end, there are one or more classes corresponding to reification classes which accepts navigation instructions starting from a given foreign object. These instructions are transmitted via the RC to one or more classes in the foreign application, which executes these instructions and returns a reference

to the resulting object. This is similar to the way methods of remote objects are executed in CORBA and the result is returned to the calling language. The difference is that there is no need to define explicit CORBA interfaces between the host (or client) application and the foreign (or server) application. Internally, we use a low-level API calls to transmit names of object attributes, names of methods, and primitive values to execute in the foreign application, and use reflection (on the foreign application side) to invoke the appropriate method call.

Combining. Using reflection and reification connectors, a host program in one type system can navigate a graph of objects in a foreign type system. Suppose we are given a foreign object `x` and a path expression `x.a.b`. That is, starting with foreign object `x`, we access its "`a`" attribute to obtain some object `y`, and then we access the "`b`" attribute of `y`, as the result of the path expression. Given `x`, we transmit "`a`" to the foreign executable. Using reflection, we can validate that "`a`" is indeed a public member of `x`, and by invoking the appropriate get method (or simply variable access), we can access the "`a`" value of `x`. We return the handle of the resulting object `y` back to the host language, and repeat the process for attribute "`b`". This is the essence of our implementation.

2.4.2 Reification Operators

We mentioned in the previous section that a host application has a set of classes that hide the details of our implementation. In fact, the handle to a foreign object is the object `R` of Figure 2.1. `R` implements a *reification operator (ROPE)* that provides access to objects in a graph of foreign objects. We give all ROPEs 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 interoperating components 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 allow 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.

We write ROPEs as $\mathtt{R}_{\mathtt{IJ}}$, where the subscript $\mathtt{I}$ denotes a component to which foreign objects are reified from the component denoted with the subscript $\mathtt{J}$. Reification operators are nonsymmetrical, that is, reifying types from $\mathtt{FTS}_{\mathtt{I}}$ to $\mathtt{FTS}_{\mathtt{J}}$ is not the same as the converse. ROPE has the transitive property, that is by applying ROPE $\mathtt{R}_{\mathtt{IJ}}$ to an instance of $\mathtt{FTS}_{\mathtt{I}}$ we reify it to the $\mathtt{FTS}_{\mathtt{J}}$. Then by applying ROPE $\mathtt{R}_{\mathtt{JK}}$ to an instance of $\mathtt{FTS}_{\mathtt{J}}$ we reify it to the $\mathtt{FTS}_{\mathtt{K}}$. The same result is achieved by applying ROPE $\mathtt{R}_{\mathtt{IK}}$ directly to an instance of $\mathtt{FTS}_{\mathtt{I}}$. This property is useful since it enables the composition of reification operators to obtain a new ROPE. Finally, an identity ROPE reifies FTS types to the same type system, and interestingly, this function is useful. Consider a Java program that needs to analyze its own structure. The identity ROPE enables such a reflective capability and extends it to all systems of interoperating components.

2.4.3 Reification of Methods

So far, we have described how host programs can navigate a graph of foreign objects. But in addition to navigation, we would like to invoke methods on foreign objects as well.

In order to reify methods, we instruct ROPE to set all parameters for a desired method and execute it in its native type system, and then reify the returned result.

We propose a reification model where operators << and >> set and get values of reified type instances, and constitute the basis for operations on reified types. In the notation shown below, we use parentheses to specify attribute name a_r of type t_k.

$R_{IJ}[t_k]...[t_k](a_r)$ << E_J

$R_{IJ}[t_k]...[t_k](a_r)$ >> E_J

The operation << takes the value of a variable E_J and instructs the ROPE to set the particular attribute of a foreign object to this value; conversely the operation >> instructs the ROPE to obtain the value of the particular attribute of a foreign object and assign it to some variable E_J.

Consider setting the `Salary` attribute of the `CTO` to the value of the integer `salary` and retrieving the value of the `Bonus` attribute of the type `CEO` into the variable `bonus` for an instance of the organizational schema shown in Figure 2.2.

```
int salary = 10000, bonus;
R["CEO"]["CTO"]("Salary") << salary;
R["CEO"]("Bonus") >> bonus;
```

Our notation can also be used to reify methods in FTSs. Since a method has a unique type determined by its name, its signature types, and its return type, we treat a method name as a type t_k and its parameters as a set of attributes a_r. We set the value for each parameter using the operation <<. Then by applying the ROPE to the typed operation t_k we execute it. The result of the execution is an instance of some type that is stored in some internal representation of the reification operator.

A fragment of C++ code that provides an example of reification of a Java method is shown in Figure 2.3. A declaration for the reified Java method is shown in Figure 2.4.

We declare ROPE `rj` that reifies Java type instances to C++. The method `mthd`

```
int j, rv;
string s;
RO_Java rj;
.....................................
rj[SomeClass][mthd](ip) << j;
rj[SomeClass][mthd](sp) << s;
rj[SomeClass][mthd] >> rv;
```

Figure 2.3: Example of a method reification.

declared as a member of Java class `SomeClass` has two parameters. We navigate to the method `mthd` and set the values of its parameters using the attribute semantics. Then we call this method, retrieve the return value and set it to the local variable `rv`.

2.4.4 Implementation Details

Here is how a reification operator `RO_Java` that reifies Java types to C++ programs can be created. A C++ program uses low-level API calls to invoke a JVM and load it in the memory as shown in Figure 2.5 with a block arrow. Then, using the *Java Native Interface (JNI)* API calls: `FindClass`, `GetStaticMethodID` and `CallStaticVoidMethod`. The JVM loads then a Java class, executes it, and returns the results of its execution to the C++ program. The interaction between the JVM and the Java class is shown in Figure 2.5 with dashed arrows.

```
class SomeClass
{
    public:
        int mthd( int ip, String sp ) {...}
}
```

Figure 2.4: Declaration of a Java class.

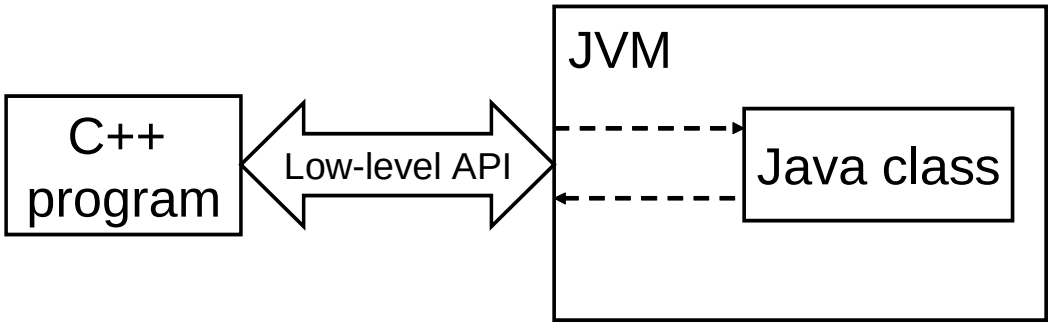


Figure 2.5: A C++ program interacting with a Java class via JVM low-level API.

An example of ROPE `RO_Java` is implemented as a C++ class declared and instantiated in Figure 2.6. For example, if we want to load a Java class called `j.class` and execute its method `A` with the parameter `In` set to integer value `5`, we write the statement below:

```
rj.Load("j").GetMethod("A").SetParam("In", 5);
```

This notation is different from the one shown in Figure 2.3 because Java does not allow programmers to overload operators, specifically, `[]` and `()`. Because of this limitation, the operator `[]` is replaced with the `GetMethod` method and the operator `()` is replaced with the `SetParam` method. Other languages (e.g., C++ and C#) allow us to overload the

```
class RO_Java
{
    public:
        RO_Java &Load( string name );
        RO_Java &GetMethod( string name );
        RO_Java &SetParam( string name, int v );
};
RO_Java rj;
```

Figure 2.6: Declaration and instantiation of ROPE `RO_Java`.

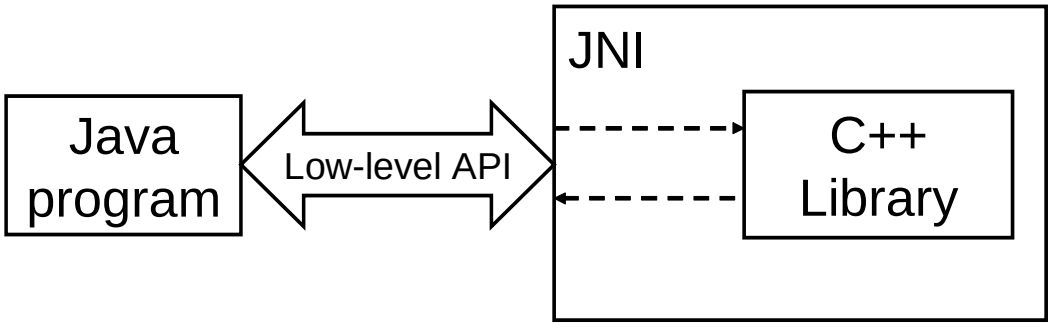


Figure 2.7: A Java program interacting with a C++ library via Java Native Interface (JNI).

bracket and parentheses operators, and thus use the default notation shown in Figure 2.3.

Now consider a ROPE `RO_CPP` that reifies C++ types to Java. It is an inverse operator to `RO_Java`. Using JNI, an instance of `RO_CPP` in a Java program invokes a C++ library and executes methods as shown in Figure 2.7. Even though the implementation of these operations is different from the ones of `RO_Java`, the class declaration is pretty much the same as shown in Figure 2.8.

The implementation of `RO_Java` is based on the concept of a shared stub [62], a native method that dispatches to other native functions and is responsible for locating and loading libraries, passing arguments, calling native functions, and returning results.

Finally, consider a ROPE `RO_XML` that reifies XML types to C++. Using a *Doc-*

```
class RO_CPP
{
    public:
        RO_CPP Load( string name );
        RO_CPP GetMethod( string name );
        RO_CPP SetParam( string name, int v );
};
```

Figure 2.8: Declaration and instantiation of ROPE `RO_CPP`.

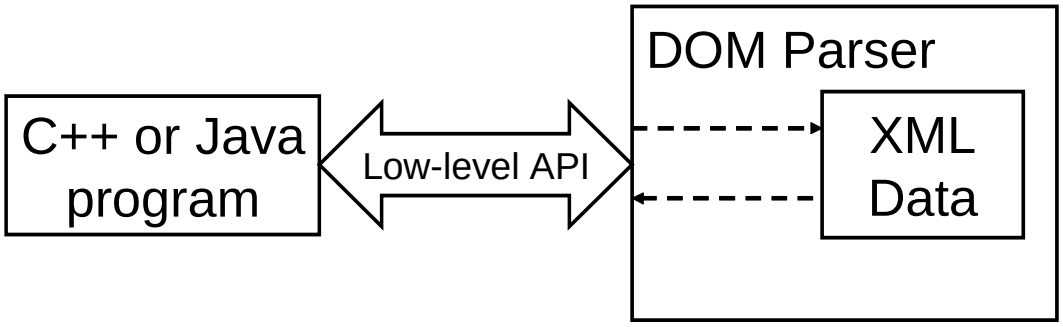


Figure 2.9: A C++ or a Java program interacting with an XML data instance using DOM.

ument Object Model (DOM) XML parser low-level API we load and parse XML data as shown in Figure 2.9. We can access any type or collection of types and change the structure of this data. We can also execute any method defined in an XLS document associated with any XML type. In this respect a declaration and implementation of `RO_XML` does not differ fundamentally from `RO_Java` or `RO_CPP` shown in Figure 2.8 and Figure 2.6 respectively.

The inverse ROPE to `RO_XML` that allows an XML instance to access C++ programs can be implemented as a C++ component that is loaded via XSL commands by a DOM parser and serves as a bridge between XML and C++ FTSs.

At this point we can introduce a generic reification operator `GenericROPE` shown in Figure 2.10 that uses `[]` and `()` operator overloading to provide uniform syntax and semantics to FTS-based programs. The operator `[]` allows programmers to access typed objects in FTS programs, for example, XML types and Java member variables and methods, and the operator `()` provides access to type attributes, for example, method parameters. Using this syntax we can rewrite the statement that loads Java class called `j.class` and execute its method `A` by setting its parameter `In` to integer value `5` as following:

```
rj[j][A](In) << 5;
```

The generality of ROPE `GenericROPE` operator enables us to introduce a plat-

form that uses this operator as an abstract parent class to implement different ROPEs derived from it. This way we provide uniformity to programmers and remove the use of low-level API calls thereby reducing complexity of systems of interoperating components.

2.4.5 Organization of ROOF

ROOF targets FTS-based applications and provides a reification model. This model is intended for programmers writing interoperating components. Unlike many existing frameworks that require a steep learning curve to understand the semantics of hundreds or even thousands of classes and collaborations among them, ROOF effectively eliminates most of the classes and collaborations that programmers would otherwise have to develop or learn.

ROOF reifies types by providing a framework that is a collection of reusable classes that implement the following functionality:

- establish a channel between FTSs. This channel may be different from interprocess communication channels because it uses a low-level API to initialize a FTS environment and establish denotation of its control and data structures;
- retrieve a collection of instances of a desired type;

```
class GenericROPE
{
    public:
        GenericROPE &operator[]( string name );
        GenericROPE &operator()( string name );
        GenericROPE &operator<<( int i );
};
GenericROPE ro;
```

Figure 2.10: Declaration and instantiation of `GenericROPE`.

- retrieve a specific instance of the given type;
- set and retrieve values from an instance of the given type;
- invoke operations on type instances, and
- implement polymorphic/overloaded operations on the structures of interoperating components.

Our goal in designing ROOF was to reduce the number of interfaces exposed to programmers to a bare minimum. A single class for each subscript `I` and `J` of reification operator $\mathtt{R}_{\mathtt{IJ}}$ implements operations on reified type instances and structures of FTS-based applications.

A layered view of the reification framework is shown in Figure 2.11. Applications `A` and `B` are based on FTSs `I` and `J` respectively. For example, `A` may be based on C# and `B` is an XML instance. The access to these applications and FTSs is provided by low-level API calls, some C# API and some XML parser API respectively, that provide both control and reflective capabilities. ROOF unifies FTSs by providing uniform polymorphic operations that are based on their low-level APIs. *Foreign Object REification Language (FOREL)* is a user interface provided by ROOF to enable programmers to write interoperable components. FOREL is presented in Chapter 3.

2.5 Evaluation of ROOF

Informally, the research question that we address is, "How does ROOF compare to lower-level API libraries?" More formally, we wish to evaluate the following hypotheses:

Cost: Using ROOF decreases the development time for interoperating components.

Quality: Using ROOF decreases the number of bugs in components;

Performance: The performance of programs written using ROOF does not differ significantly from programs written using low-level API libraries.

To test our hypotheses we designed a controlled experiment, and we applied ROOF to a commercial real-time component-based semiconductor overlay analysis and control system. A controlled experiment is described in Section 2.5.1, and the experience with using ROOF for a commercial system is described in Section 2.5.2.

2.5.1 Controlled Experiment

We conducted a controlled experiment to evaluate ROOF at the department of computer science of the Texas State University at San Marcos. As part of a graduate course on software engineering, students were asked to write C++ programs that accessed and manipulated XML data. We divided students in two groups, each consisted of five students. Students from the first group were given thirty-minute presentation on how to use ROOF, and students from the second group were given two-hour lecture on how to use Xerces API calls.

The difference in the length of instruction lectures is the result of the complexity

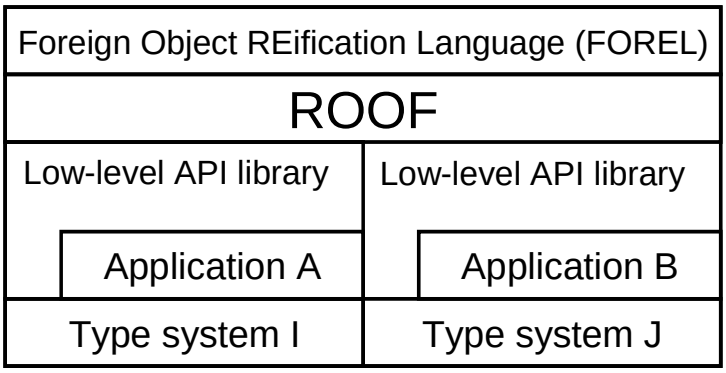


Figure 2.11: Abstraction layers for a type reification framework.

of Xerces API calls. Initially, both lectures were designed to be half-hour long. However, students asked many more questions about using Xerces API calls. These questions were about sequences in which API calls are made and parameters that should be passed to them. Some API calls return intermediate objects that should be used when making other API calls, and understanding of these dependencies required more time that learning how to use ROOF.

One student from the second group had prior programming experience working with Xerces DOM XML parser, and no students had prior exposure to ROOF. All students had basic knowledge of C++. A special lecture was given to all students participating in the experiment on XML. Response variables in this case study are times it takes participants to develop programs, execution time and memory consumption for developed programs.

Data

We used an XMark benchmark XML document [76][75] for our experiments. The schema [75] for this XML document is shown in Figure 2.12. While this schema was not used to validate the XML document, students consulted the schema to understand how to access data elements. Since the XML document was not related to any specific domain, students were not given any explanation on the semantics of data.

Program

The specification for the program consists of five items:

- Create a new XML file called "`new.xml`" with the root `sites`, and copy all data elements that have attributes from the XMark data under this root element.
- Insert the attribute "`rich`" with the value "`yes`" to all elements `person` whose in-

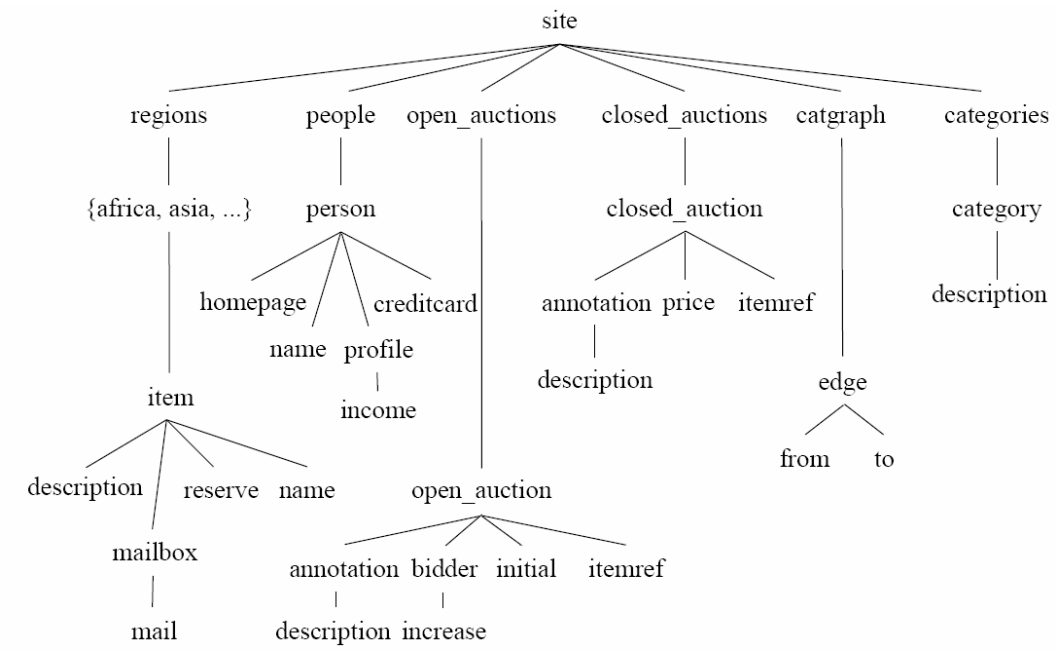


Figure 2.12: XML schema for XMark benchmark documents.

come is greater than $10,000. The path to the element `person` is `site`→`people`→`person`, and the path to the element `income` is `site`→`people`→`person`→`profile`→`income`.

- Delete closed auctions whose price is less than $3.

- Add category with value "`category10000`" under the path `site`→`categories`, and add the corresponding edge from this category to the category "`category99`' under the path `site`→`catgraph`→`edge`.

- Create new data element `EducatedPeople` under the path `site`→`people`. Move all data elements `person` whose attribute `Education` in the child element `profile` has the value "`Graduate School`" under the added element.

Students were given two hours to write this program. Each student had a template of a C++ program and a makefile that was used to compile it, so that students did not have to spend

time to learn how to locate necessary include files and to what libraries to link.

Results

The results of our case study are summarized in Table 2.1. The first column specifies case study participants by assigned unique numbers and platforms they use (X stands for Xerces and R stands for ROOF). The second column shows the time it took for participants to write programs. If by the end of the experiment a participant said that the program was not ready, then we marked this program as "`unfinished`". The third column shows the number of *lines of code (LOC)* in programs that participants wrote not including LOC that are a part of the template program code. After running each submitted program, the resulting XML data was examined with respect to the expected structure defined by the specification. Each deviation from the specification in the XML data was counted as a bug, and a total number of bugs was put into the fourth column of the table. If a program crashed, we marked it with "`crash`". When running submitted programs we measured their execution times and the maximum amount of memory their occupy. We report these results in the columns five and six respectively.

The average number of LOC is 250 for programs that use Xerces and 43 for programs that use ROOF. That is, using low-level API calls requires programmers to write six times more code than when they use ROOF. ROOF-based programs have on average 0.8 bugs, and Xerces-based programs have 2.7 bugs. With respect to performance, the average execution time of ROOF-based programs is 388 seconds versus 356 seconds for Xerces-based programs, and average memory consumptions are 780 and 540Mb respectively. That is, ROOF imposes nine percent overhead in execution time and 44% in memory usage.

Participant	Time to write	No of bugs	Size, LOC	Exec time, sec	Memory, Mb
Student 1-X	96 mins	2	228	387	535
Student 2-X	120 mins	2	276	319	561
Student 3-X	unfinished	4	294	362	524
Student 4-X	unfinished	crash	323	-	-
Student 5-X	unfinished	crash	329	-	-
Student 6-R	42 mins	1	35	394	782
Student 7-R	47 mins	0	49	421	823
Student 8-R	55 mins	0	43	371	793
Student 9-R	62 mins	1	57	388	811
Student 10-R	108 mins	2	31	367	692

Table 2.1: The results of the case study.

Threats to Validity

We discuss internal and external threats to the validity of our case study, and how these threats can be minimized.

Internal Validity. Internal validity establishes that the values of the dependent variables are solely the result of the manipulations of the independent variables. Dependent variables are the response variables. A major threat to internal validity would be the stimulus variables in our experiment. Students should be given sufficient stimuli to make the best effort to write quality code, without this stimuli they are likely to walk away from the experiment when encountering any difficulties. To address this concern, we told participants that they would be graded based on the overall quality of their work.

All participants were given the same C++ compiler (gcc) and a template for C++ program in order to reduce the effect of confounding variables (whose effects on response variables cannot be distinguished from each other). They were given full

access to the Xerces documentation and code samples as well as ROOF. However, students were prevented from using any libraries or tools other than the gcc compiler, an XML viewer, and the XML schema.

External Validity. External validity establishes the extent to which the results support the claims of ROOF generalizability. A major threat to the external validity is that the results of the case study with the XMark-based programs may not be representative of large-scale commercial systems of interoperating components. Since programs written in our case study are small (less than 300 lines of code), we address this threat with an additional study described in our experience report in Section 2.5.2.

2.5.2 Experience Report

We applied our approach to a real-time component-based semiconductor overlay analysis and control system. The *Archer Analyzer (AA)* is a software package geared for Archer 10 optical overlay metrology systems manufactured by the California-based KLA-Tencor Corporation [10][9].

The purpose of optical overlay measurements is to detect and fix misalignments between layers of semiconductor chips that were put on a silicon wafer using microlithography processes. Overlay or misregistration is a vector quantity defined at every point on the wafer. Ideally, the value of overlay should be zero. When nonzero overlay is detected the tool is stopped and the error is corrected as soon as possible.

The first attempt was started by KLA-Tencor in 1997. A team of forty specialists was assembled; the original system was written in a programming language REXX in the beginning of 80s and became obsolete. The design for the first release of the product was done carefully. The management allocated more than a year to hire people, educate them

about the company and existing systems and processes, and to produce design documentation. Each group planned high-level design, functionality and interfaces in great detail. The approach to design of this system of interoperating components was IDL-based with CORBA and DCOM used as the underlying distributed object middleware platforms. The number of classes, data structures, and various methods and functions was in the thousands, and the complexity of the system grew to such a degree that communication overhead between groups became excessive. Clearly, the selected approach was not scalable. This situation was compounded by the conflicting and overlapping terminology that led to many syntactically similar types serving different purposes. When the size of code grew to two hundred thousand lines the project became unmanageable.

It took four years and more than $3 million to design and develop the software and take it to beta test at Texas Instruments Corporation. The test failed with the *Mean Time to Failure (MTF)* approximately two hours despite the initial requirement for MTF $\approx 2,000$ hours. Each time the system went down it was virtually impossible to determine the cause of breakdown since each group claimed that its components performed internal algorithms properly and the algorithms themselves were correct. The management tried to hire more people who would handle the inter-component connectivity, but it realized soon that the project would require significant additional investment as the number of failures increased the further testing went. Therefore, management decided to start from scratch.

The second attempt was started in January 2001. The author of this thesis was hired as a consultant to define the strategy for the implementation of AA. He saw that it was very difficult to extract full and correct information about all aspects of communications among different modules of the existing system. Each group member of the first implementation could clearly explain the programming logic of his/her code and had a clear picture

of a schema of data pertinent to the part of the project the s/he owned. However, the understanding of the overall structure of the project, relations among modules and data was vague. When programmers followed the process of creating structures that map FTS types, they created a system based on many wrong assumptions that was extremely difficult to trace in the resulting code. When brought together these group members could not agree upon all details of the big picture of the project.

We applied our approach to target the key problems of the project. The first was to make each member of the team think about a common schema. The second was to introduce uniformity and reduce the complexity of code. The way to do it was to eradicate the thousands of explicit mappings between FTS types.

The first problem was addressed with the type graph solution explained in Section 2.3. A single schema and its instance were created by an engineer whose job was to maintain the schema and serve as a single point of reference to define every term. The second problem was solved by enabling programmers to reference each type exactly as it was defined in the schema. This required a common platform that subsumed all other FTSs used in the project. Thus, ROOF was created. Each team member was given a thirty minute presentation of the basic structure of ROOF and its operations. Moreover, each programmer was told that if s/he found the concept and implementation of the ROOF difficult to understand and use they may go back to low-level APIs that they originally planned on using. They did not go back.

The architecture of the AA is based on many FTSs. AA is created as an open system and is integrated in the production environment and communicates with many other FTSs, such as EJBs, CORBA, and .Net assemblies. The components for AA are created using C++ and different low-level APIs were used for a variety of tasks such as parsing XML

data and forming matrices. Since we did not generate or create mapped types in the host programming language it is not difficult to see how we achieved the reduction of code close to 80% for this project. The size of the code that handled interoperability and that was created during the successful second attempt was measured with respect to the size of the corresponding code (by functionality) written during the first failed attempt.

The results exceeded expectations. The system was much easier to write and the resulting code was clear. It took over a year with a team of six programmers using our approach to deliver this project to beta test at AMD Corp. that was successfully passed in 2002. This software has since been successfully commercialized [10][9].

2.6 Summary

In today's enterprise environment it is desirable to make each program interoperate with other programs. However, existing solutions have the significant accidental complexity as there are potentially $O(n^2)$ possible APIs for achieving communication among n programs. We have solved the problem of enabling all-connected architecture graph of systems of interoperating components by providing a single framework called ROOF with a set of standard and clear API calls

ROOF is a simple and effective way to develop easily maintainable and evolvable systems of interoperating components by reifying foreign type instances and their operations into first-class language objects and enabling access to and manipulation of them. By doing so we hide the tremendously ugly, hard-to-learn, hard-to-maintain, and hard-to-evolve code that programmers must write or generate today, i.e., we simplified code of interoperating components, making it scalable and easier to write, maintain, and evolve.

The capability to write uniform and compact programs that work with applications

based on different type systems enables faster development of complex systems at a fraction of their cost. Indeed, if developers concentrate on reasoning about properties of applications without the need to master many low-level API calls that operate on type system elements then it significantly improves their productivity and the quality of the resulting system. We came to this conclusion based on using our approach for a complex commercial software system and for a case study.

Chapter 3

Type Checking and Type Inference For Interoperating Components

An abstract view of a component is a graph of interconnected objects. An XML document, when "hosted" by an XML parser, is a tree of *Document Object Model (DOM)* objects. A Java program, when "hosted" by a JVM, becomes a graph of Java objects. *Path expressions* can be used to navigate from one object to another in such graphs. The expression "`a.b`" means to start at object named `a` and traverse to `b`, where `b` is the name of the member field of `a` that references some object. More generally, methods can replace field names in a path expression enabling computations to be performed enroute.

Type checking is an important mechanism to guarantee the safety of ROOF-based programs and improve programmers productivity. When components are written in the same language, conventional type checking algorithms can be used to verify the correctness of path expressions statically. Contemporary programming languages handle these common situations with great success. However, there are many other situations where the static

type checking of path expressions is not attempted, resulting in the run-time discovery of type errors. This arises when a (*native*) component references a *foreign* component, which is written in a different language or that is hosted on a different platform. Consider the following examples.

A C program can manipulate Java objects by invoking *Java Virtual Machine (JVM)* API calls. (For example, a static method can be located in a Java object by calling the JVM API `GetStaticMethodID` with the name of the method passed as a parameter. Broadly, JVM API calls take a string of a path expression as input, and the JVM interprets that string. If a member name is incorrect, or a method call has the wrong number of parameters, then these errors are discovered only at run-time. None of these errors will be detected by the type system of a C compiler.

Another example is accessing databases via *Java DataBase Connectivity (JDBC)*. SQL statements are passed as strings to JDBC APIs. At run-time, these strings are parsed and matched to a given database schema. If the target relation does not exist or attribute names are misspelled, the error is reported at run-time. A third example is a Java program accessing data from an XML document. The program uses DOM API calls to navigate a parsed XML document, where navigation methods take string names of desired XML nodes as parameters. If a name is incorrect, the error is discovered at run-time.

In each example, there is enough information to detect these errors statically at compile-time, as the Java/C executables or database/XML schemas are readily available. The problem is that the type systems for different components (e.g., a Java program, a C program, a database, an XML document) are not the same. While there is prior work to perform such static checks on the interaction of specific pairs of components (e.g., Java and databases) [45], there is no general approach to accomplish the static type checking of

references to arbitrary foreign components. We need a compiler that is capable of checking reified objects against foreign type systems.

In this chapter, we describe and evaluate *Foreign Object REification Language (FOREL)*, an extension for object-oriented languages that provides a general abstraction for foreign component access and manipulation. Traversal strategies, borrowed from adaptive programming [63], are used in FOREL to simplify foreign component navigation and manipulation. As shown in Figure 2.11, *Foreign Object REification Language (FOREL)* is a user interface provided by ROOF to enable programmers to write interoperating components.

FOREL type checking coupled with a conservative static analysis mechanism reports potential errors when referencing foreign components. We formalize the type checking rules of FOREL, present its operational semantics, and prove the soundness of its type system. We define a static analysis of FOREL expressions that is based on the combination of algorithms defined in adaptive programming, the formal type system of FOREL, and our type inference algorithm. We tested FOREL on *Archer Analyzer*, a real-time semiconductor overlay analysis and control system geared for the optical overlay metrology tool *Archer 10* manufactured by California-based KLA-Tencor Corporation, and a number of smaller systems with interoperating components. We detected a number of known and unknown errors in these applications with high precision and a low false-positive rate thus demonstrating the effectiveness of our approach.

3.1 Language Description

In this section, we present FOREL as a type system for object-oriented programs that provides a common abstraction for interoperating components. We describe the FOREL

language through a series of examples. A more precise description of the type system is provided in Section 3.3.

3.1.1 Overview

FOREL is based on ROOF which abstracts the common functionality of different platforms by presenting a small set of operations for navigating and manipulating foreign objects [48]. These operations are navigating to foreign objects, setting and getting their values, modifying their structures, and invoking their methods. The ROOF abstraction makes the task of static checking of interoperating components tractable by reducing the multiplicity of platform API calls to a small set of operations on foreign objects that are common to all platforms. By introducing a simple extension to grammars of object-oriented languages we enable the collection of information about foreign objects at compile time. This information is used to perform static type checking in order to determine possible errors that could otherwise be detected only at the runtime.

Reification OPErator (ROPE) is a basic type in FOREL that denotes a communication channel between components through which they interact when running on different platforms. Its purpose is to make foreign objects first-class entities in native components (i.e., to reify them), and to enable programmers to perform operations on these objects the same way they work with native objects (i.e., by addressing them directly without the use of complicated platform API calls). ROPEs store the structures of foreign components in an intermediate format, and their abstraction enables programmers to concentrate on high-level operations on foreign components without getting bogged down in the low-level platform API calls.

3.1.2 A Concrete Example

ROPE objects are declared using the `ro` keyword followed by the location of the schema describing foreign components. In Figure 3.1, at line 1 ROPE `x` denotes a foreign object whose schema is given in the file `orgstr.xsd`. Line 2 navigates to the field `stock` of the foreign component `CEO`, obtains its value and stores it in the local variable `stockOptions` of type `int`. If one of the navigated objects does not exist, it is created automatically and an exception is thrown informing programmers about the situation. Line 3 sets the value of the field `salary` of the foreign object `CTO` that is contained in the foreign object `CEO` to the result of an expression that multiplies the value of the local variable `stockOptions` by real number `15.2` that is assumed to be the price of a single share.

Lines 4, 5, and 6 depict ROPEs inserting and deleting foreign objects. The `VPEng` component of the type `VPEngineering` is created and aggregated in the `CEO` object at line 4 by invoking method `InsertPart` on the ROPE `x`. In line 5 field `salary` of the type `float` is created and inserted in the object `VPEng`, and in line 6 the object `CTO` with all of its fields is deleted.

The syntax of FOREL is different from the syntax of ROOF operations described in

```
1: ro ["orgstr.xsd"] x = new ro;
2: int stockOptions=x["CEO"]["stock"];
3: x["CEO"]["CTO"]["salary"]=stockOptions*15.2;
4: x["CEO"].InsertPart("VPEngineering","VPEng");
5: x["CEO"]["CFO"].InsertField("Salary","float");
6: x["CEO"].DeletePart( "CTO" );
7: ro ["orgstr.class"] y = new ro;
8: int newShares = y["CEO"]["IncreaseStock"].
9:     Invoke(<"annual"=1><"percent"=10>);
```

Figure 3.1: A fragment of FOREL program.

Chapter 2. The latter was designed and implemented in 2001 using generics and overloading operators in languages that allow programmers to do it (e.g., C++, C#). FOREL was designed few years later by extending grammars of object-oriented languages, and its new syntax reflected our experience using ROOF for different projects.

3.1.3 Operations

FOREL operations include navigating to foreign components, reading their values into local variables, writing values from local variables into foreign components, and changing them by inserting, deleting, and copying fields. Invoking foreign methods using ROPEs and returning their results and storing them in local variables are also discussed.

Navigations

In FOREL, names of foreign components are specified as strings. Brackets `[]` and parentheses `()` are used to guard these names to prevent possible naming conflicts between names of foreign objects and names of native object and language keywords. Parentheses are used to denote attributes in XML components.

Invoking Foreign Methods

In FOREL navigating to a function in a foreign component results in its invocation and the retrieval of the resulting return value, if any. Currently, FOREL deals only with the exchange of primitive values. At line 7 ROPE `y` is declared pointing to the Java class `orgstr.class`. At lines 8 and 9 foreign method `IncreaseStock` of the component `CEO` is invoked to increase the number of shares owned by the `CEO`, and its return value specifying the resulting number of shares is put in the local variable `newShares`. The first

parameter of this foreign method is named `annual` for an annual increase in stock options, and the second parameter is `percent` specifying the percentage of the increase.

3.2 Checking FOREL Expressions

Using FOREL expressions programmers can navigate and manipulate foreign objects whose names are specified as parameters to these expressions. If these parameters are constants, then the names of foreign objects are known at compile time, and FOREL expressions can be type checked statically. Otherwise, the names of foreign objects are computed at runtime, and it is an undecidable problem in general to check the type safety of these expressions.

We use a combination of compile time type checking and type inference in order to check the correctness of FOREL expressions. In this section we use a toy example to illustrate the algorithm used to check the correctness of FOREL expressions. Recall that the input to this algorithm is a native object written using FOREL and a schema describing foreign objects with which this native object interoperates. The schema is shown in Figure 3.2 as a graph whose nodes are foreign objects and the edges are the references between them. Consider the FOREL expression `x["s"][e1][e2]["d"]`, where `x` is a ROPE object navigating from the foreign object named `"s"` to the foreign object named `"d"`. Expressions `e1` and `e2` compute values of the intermediate nodes in the traversal path at runtime. If constant strings are specified in place of `e1` and `e2`, for example, "`a`" and "`f`", then typing rules can be applied to the path expression "`s.a.f.d`" to determine its correctness. We given these typing rules in Section 3.3.5. However, when values of expressions `e1` and `e2` can be determined only at runtime, then an inference algorithm is invoked on this strategy to compute sets of values for these expressions.

The gist of the algorithm is in deciding which foreign objects should be visited from

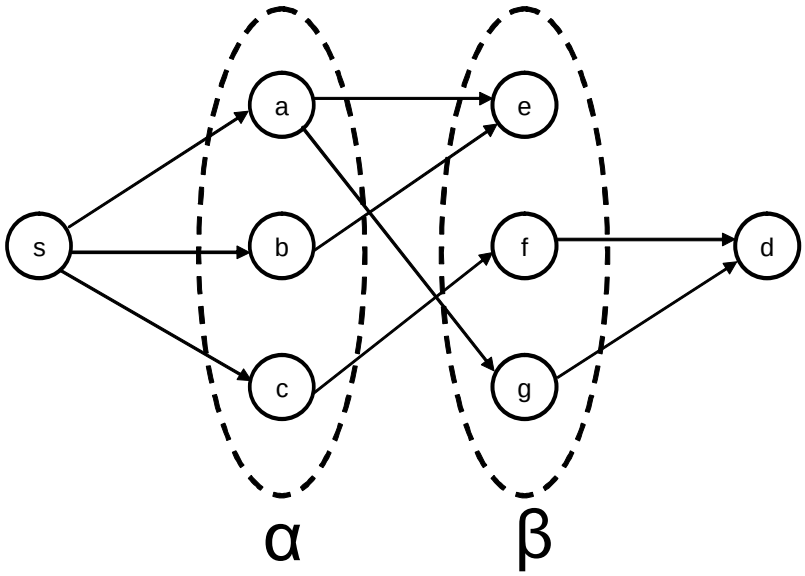


Figure 3.2: Navigation paths from the source object named `s` to the destination object named `d`.

the node `s` in the graph in order to reach the destination node `d` given a strategy `S` (i.e., a path expression). We check for possible paths from source to destination nodes, but we do not instantiate a FOREL program to check for active bindings of the expressions that compute names of foreign objects. By finding all paths leading to the destination node `d` starting with the source node `s` we can conclude whether the navigation expression is correct. If no path exists, then this expression is incorrect. Alternatively, if there are paths leading from `s` to `d` by traversing two objects, then this expression is correct provided that expressions `e1` and `e2` evaluate to the names of the nodes in the discovered paths. Our type inference algorithm infers possible names computed by expressions `e1` and `e2` at compile time. We describe this algorithm informally in this section. Its formal description will be given in Section 3.3.

Expressions `e1` and `e2` are replaced with the object name variables α and β correspondingly, and the original expression is converted into traversal strategy `S = s` $\rightarrow$

$\alpha \rightarrow \beta \rightarrow$ d. We introduce the function first(s) computing a set of edges that can be traversed from node s. These edges lead to a set of objects designated by the variable α. Function first(s) is computed using a graph reachability algorithm, and it returns edges that could lead to the target node. According to Figure 3.2, α = {a, b, c}. Then for each element of α, function first is computed. As a result we obtain β = {e, f, g}, where first(a) = {e, g}, first(b) = {e}, and first(c) = {f}. Proceeding to the next step we obtain first(e) = {$\oslash$}, first(f) = {d}, and first(g) = {d}. From the computed node values a worklist W is formed containing a set of all computed paths, W = {(s, a, e), (s, a, g, d), (s, b, e), (s, c, f, d)}. Each path is taken from W and checked to see whether it contains nodes s and d. If both nodes are present in the path as its source and target, then a valid path is obtained. If no paths exist, then an error is issued and the algorithm terminates.

An example of an incorrect FOREL expression is x["s"][e1][e2][e3]["d"]. All paths between nodes s and d have at most two objects. Therefore no matter what values are computed at runtime for expressions e1, e2, and e3 they cannot represent objects in a valid path between the source and the destination objects. The other example of an incorrect FOREL expression is x["s"]["b"][e1]["d"]. In this case there is no value for the expression e1 that makes it a valid path, and the FOREL compiler will issue a warning.

3.3 Formalization

A key property of FOREL is the compatibility of operations on foreign types. If a native program accesses objects that do not exist in foreign programs or attempts to set a value of a foreign object that is not compatible with its type, then it violates constraints imposed on interoperating components. A FOREL compiler type checks foreign systems and guar-

antees that incorrect operations cannot be executed. We now precisely define type graphs, paths, and traversals and use these definitions for the formalization of FOREL. These definitions enable us to reason about not only the operations on the types directly specified in the operations of FOREL, but also infer type information about objects that are present in some path in a foreign program and whose presence is not explicitly defined in the ROPE expressions navigating through this path.

3.3.1 Classes in FOREL

When navigating to foreign objects in FOREL we throw away inheritance and subtyping relations between classes of these objects, and we enhance the concept of class to include hierarchical containments (e.g., program scopes and system hierarchies). Classes designate namespaces and methods, and they can also define external environments for program scopes. For example, in order to access a field in a Java class one should load a JVM, locate the package that contains the Java class (i.e., to navigate through a sequence of directories), locate the file that contains the Java class, instantiate it by loading this class into the JVM using a classloader, and finally access the desired field. Examples of classes in FOREL are also markup language types (e.g., elements, tags and attributes) and relational database objects (e.g., tables and stored procedures).

3.3.2 Type Graphs, Paths, and Traversals

The basic notions of adaptive programming are class and object graphs [64]. We replace the notion of class with a more general notion of type since we apply FOREL to values that are not necessarily instances of classes. For example, we reify markup language objects that are not always instances of classes to the type systems of OO languages, and we need

a theoretical abstraction that treats all objects uniformly.

We distinguish between complex and simple types. Complex types contain fields while simple types do not. Let T be finite sets of type names and F of field names or labels, and two distinct symbols this $\in$ F and $\diamond \in$ F. Type graphs are directed graphs G = (V, E, L) such that

- V $\subseteq$ T, the nodes are type names;
- L $\subseteq$ F, edges are labeled by field names, or "$\diamond$" if fields do not have names. We call edges that are labeled by "$\diamond$" *aggregation edges*, and edges that are labeled by field names *reference edges*. The difference between aggregation and reference edges becomes clear with the following example. Fields of classes in object-oriented languages designate instances of some classes, and these fields have names that are used to reference them. It means that each field of a class is defined by its name and the name of the class (type) that this field is an instance of. The name of a field is the label of the corresponding reference edge in the type graph.

  When a class designates a directory and the other class designates a file that is contained in this directory, the type graph has two nodes, one for the directory and the other for the file it contains. The names of the directory and the file serve as their types. The file is an instance of the class that represents it, and this file is also a field of the directory class, however, this file field does not have a name. The relation between the directory and the file type is represented using the edge labeled with the "$\diamond$" in the type graph.

- E $\subseteq$ L$\times$V$\times$V, edges are cross-products of labels and nodes;
- for each v $\in$ V, the labels of all outgoing edges with the exception of "$\diamond$" are dis-

tinct;

- for each for each $\mathtt{v} \in \mathtt{V}$, where $\mathtt{v}$ represents a concrete type, $\mathtt{v} \xrightarrow{\mathtt{this}} \mathtt{v} \in \mathtt{E}$.

An object graph is a labeled directed graph `O = (V', E', L')` that is an instance of a type graph `G = (V, E, L)` under a given function `Class` that maps objects to their classes, if the following conditions are satisfied:

- for all objects $\mathtt{o} \in \mathtt{V'}$, $\mathtt{o}$ is an instance of the concrete type given by function `Class(o)`;

- for each object $\mathtt{o} \in \mathtt{V'}$, the labels of its outgoing reference edges are exactly those of the set of labels of references of `Class(o)` including edges and their labels inherited from parent classes;

- for each edge $\mathtt{o} \xrightarrow{\mathtt{e}} \mathtt{o'} \in \mathtt{E'}$, `Class(o)` has a reference edge $\mathtt{v} \xrightarrow{\mathtt{e}} \mathtt{u}$ such that $\mathtt{v}$ is a parent type of `Class(o)` and $\mathtt{u}$ is a parent type of `Class(o')`.

An object graph is a model of the objects, represented in the heap or elsewhere, and their references to each other. A collection of fields in an object graph is a set of edges labeled by field names. A collection of aggregated objects in an object graph is a set of edges labeled by "$\diamond$". A path in a type graph `G = (V, E, L)` is a sequence of nodes and labels $p_G = \langle v_0 e_1, v_1 e_2, \ldots e_n v_n \rangle$, where $\mathtt{v}_i \in \mathtt{V}$ and $v_i \xrightarrow{e_{i+1}} v_{i+1}$ for $0 \leq i \leq n$. We define a concrete path to be an alternating sequence of type names and labels designating reference edges. In general a concrete path p_c is a subset of the corresponding type path p_G, i.e. $p_c \subseteq p_G$. Since some classes are compiled away and are not present in the object graphs (e.g., namespaces in `C++` are not a part of object representations), the paths in object graphs are subsets of the corresponding type graphs.

An object graph has the special object $o_r \in V'$, o_r is a collection of root objects $o_r \subseteq V'$ in the object graph O given by function root: $O \rightarrow o_r$. This object has type Class(o_r)=ro and its relation with objects in its collection is expressed via $o_r \xrightarrow{\diamond} o' \in E'$.

3.3.3 Example of a Type Graph

An example of the type graph of the organizational structure of a company is shown in Figure 3.3. A FOREL program based on this graph is shown in Figure 3.1. CEO is a root type that has the field stock of type int and aggregates type CTO. CTO is a type that has fields salary of type Check and boss of type CEO. Type Check has in turn fields amount of type float and issuer of type CEO. Instances of this type graph can be equally implemented in XML, Java, and other languages. For example, consider a possible implementation of this graph in Java: the aggregation of objects is done by instantiating them in the scope of the object that contains them (i.e., an instance of CTO is included in the scope of the type CEO), or by creating Java packages using the corresponding type names in separate directories. We can navigate to the field salary of type CTO by executing FOREL expression x["CEO"]["CTO"]("salary"), where x is an instance of ROPE.

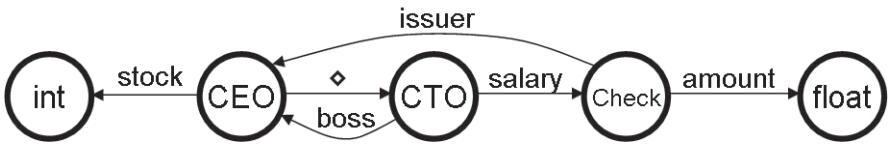


Figure 3.3: Type graph of the organizational structure of a company.

3.3.4 Formalization of a Traversal Problem

Given an object o of some type we wish to find all reachable objects that satisfy certain criteria. This task is equivalent to determining whether FOREL statements that describe navigation paths through foreign programs are correct. Navigation paths specified in FOREL ROPE statements can be thought of as specifications of constraints for the object reachability problem. Finding reachable objects is done via traversals. The traversal of an edge labeled e corresponds to retrieving the value of the e field. Every edge in the object graph is an image of a has-part edge in the type graph: there is an edge $e(o_1, o_2)$ in O only when there exist types v_1 and v_2 such that object o_1 is of type v_1, v_1 has an e-part of type v_2, and o_2 is of type v_2.

The first node of a path p is called the *source* of p and the last node is called the *target* of p. A traversal of an object graph O started with an object v_i and guided by paths from a set of paths p is done by performing depth-first search on O with p used to prune this search. The resulting traversal history is a depth-first traversal of the object graph along object paths agreeing with the given concrete path set. We formalize the problem of finding all reachable objects from a given object o that satisfy certain criteria as follows. For each pair of classes c and c′, we need to find a set of edges e by computing FIRST(c, c′) iff it is possible for an object of type c to reach an object of type c′ by a path beginning with an edge e. More precisely, FIRST(c, c′)=e∈E, such that there exists an object graph O of C and objects o and o′ such that:

1. Class(o) = c,
2. Class(o′) = c′, and
3. o e^* o′.

The last condition, o e^* o′ says that there is (∃) a path from o to o′ in the object graph, consisting of an edge labeled e, followed by any sequence of edges in the graph. Our lack of information about the actual graph is represented by the existential operator ∃.

3.3.5 FOREL

We formalize the FOREL language based on the type graph model. We present the syntax of FOREL, give its operational semantics and type checking rules, and prove the soundness of its type system.

Syntax

Figure 3.4 presents a ClassicJava-based syntax of FOREL. FOREL-based rules are represented with a **bold font**. The metavariable C ranges over class names; field and fd range over field declarations and field names respectively; t ranges over types; meth ranges over methods; ρ ranges over reification operators; ROPE ranges over methods of reification operators; T ranges over variants that are used to specify values of parameters when invoking foreign methods, and defn ranges over class definitions. A component in FOREL, P, is a pair (defn, e) of class definitions and an expression. FOREL extends ClassicJava in several ways. Reification operators that designate dynamic connectors between programs are represented by objects of type ro. To reason about structures of programs connected by ROPEs, attribute L is added that points to a schema that describes a foreign program. Therefore, a ROPE ρ maps objects and fields of foreign programs and data sources designated by their locations L to variables in native programs: objects and their fields. T ranges over navigation and access expressions in ROPEs. An error expression is also included, representing failed casts and null dereferences.

```
P       ::=  defn e
defn    ::=  class C extends C { field meth }
             | interface i extends i { meth }
field   ::=  t fd
meth    ::=  t md( arg ) { body }
arg     ::=  t var
body    ::=  e | abstract
e       ::=  new C | new ρ | new ρ constraint [L]
             | null | fd | fd = e | e.md(e)
             | view t e | let var = e in e | e N
             | valueOf(e) | ROPE
N       ::=  [e] | (e) | (T)
ROPE    ::=  e.InsertPart(e.e)
             | e.InsertField(e.e)
             | e.DeletePart(e.e)
             | e.DeleteField(e.e) | e.Store(e)
ρ       ::=  ro | [L] ro
t       ::=  C | i | ρ
T       ::=  <var = e>
var     ::=  a variable name or this
C       ::=  a class name or Object
i       ::=  interface name or Empty
fd      ::=  a field name
md      ::=  a method name
```

Figure 3.4: ClassicJava-based grammar of FOREL

Operational Semantics

A program in a system of interoperating components is a running instance of a binary executable file, which consists of a set of locations and a set of values. The state, S, of a system of interoperating components, PS, that comprises interoperating programs P_1, P_2, ..., P_n is the union of the states of these programs $S=\bigcup_n P_n$. The state of a program is obtained via mapping function ProgramState: $PS \times P \rightarrow S_P$. When we write S_P in the reduction rules, we mean it as shorthand for the application of the ProgramState function to obtain the state S of some program P. The evaluation relation, defined by the reduction rules in Figure 3.5, has the form $PS \hookrightarrow PS'$, where $PS=\langle P, e, S_P \rangle^*$, read "A collection of programs in a system of interoperating components PS, executing expression e with the initial state S_P transitions to a new set of programs, producing the new state S_P'." In these rules P is a program and S is a state. Transition $\hookrightarrow \subseteq \langle P, e, S_P \rangle \times \langle P, e, S_P \rangle$.

RO-NEW

$$\frac{\langle \emptyset, e, S_\emptyset \rangle \in PS \qquad \texttt{getObjectGraph}(\emptyset) = G \qquad \langle \emptyset, S_\emptyset \rangle \vdash \texttt{root}(G) = o_r}{\langle P, E[\texttt{new ro}], S_P \rangle, PS \hookrightarrow \langle P, E[o_P], S_P[o_P \mapsto (\texttt{ro}, \texttt{object} \mapsto o_r)] \rangle, PS}$$

RO-LOCNEW

$$\frac{\langle Q, e, S_Q \rangle \in PS \qquad \texttt{getObjectGraph}(Q) = G \qquad \langle \emptyset, S_\emptyset \rangle \vdash \texttt{root}(G) = o_r}{\langle P, E[\texttt{new [Q] ro}], S_P \rangle, PS \hookrightarrow \langle P, E[o_P], S_P[o_P \mapsto (\texttt{ro}, \texttt{object} \mapsto o_r)] \rangle, PS}$$

RO-OLGET

$$\frac{\langle Q, e, S_Q \rangle \in PS \qquad S_P(o_P) = (\texttt{ro}, \ \texttt{object} \mapsto o_Q) \qquad t \in \texttt{String} \qquad \langle Q, S_Q \rangle \vdash \texttt{parts}(o_Q, t) = o}{\langle P, E[o_P\,\texttt{[t]}], S_P \rangle, PS \hookrightarrow \langle P, E[o_P], S_P[o_P \mapsto (\texttt{ro}, \texttt{object} \mapsto o)] \rangle, PS}$$

RO-ONGET

$$\frac{\langle Q, e, S_Q \rangle \in PS \qquad S_P(o_P) = (\texttt{ro}, \ \texttt{object} \mapsto o_Q) \qquad n \in \mathbb{N} \qquad \langle Q, S_Q \rangle \vdash \texttt{part}(o_Q, n) = o}{\langle P, E[o_P\,\texttt{[n]}], S_P \rangle, PS \hookrightarrow \langle P, E[o_P], S_P[o_P \mapsto (\texttt{ro}, \texttt{object} \mapsto o)] \rangle, PS}$$

RO-FLGET

$$\frac{\langle Q, e, S_Q \rangle \in PS \qquad S_P(o_P) = (\texttt{ro}, \ \texttt{object} \mapsto o_Q) \qquad t \in \texttt{String} \qquad \langle Q, S_Q \rangle \vdash \texttt{fieldByName}(o_Q, t) = o}{\langle P, E[o_P\,\texttt{(t)}], S_P \rangle, PS \hookrightarrow \langle P, E[o_P], S_P[o_P \mapsto (\texttt{ro}, \texttt{object} \mapsto o)] \rangle, PS}$$

RO-FNGET

$$\frac{\langle Q, e, S_Q \rangle \in PS \qquad S_P(o_P) = (\texttt{ro}, \ \texttt{object} \mapsto o_Q) \qquad n \in \mathbb{N} \qquad \langle Q, S_Q \rangle \vdash \texttt{fieldBySeqNum}(o_Q, n) = o}{\langle P, E[o_P\,\texttt{(n)}], S_P \rangle, PS \hookrightarrow \langle P, E[o_P], S_P[o_P \mapsto (\texttt{ro}, \texttt{object} \mapsto o)] \rangle, PS}$$

RO-FOSET

$$\frac{S_P(o_P) = (\texttt{ro}, \ \texttt{object} \mapsto o_Q) \qquad o' = \Omega(o)}{\langle Q, e, S_Q \rangle \in PS, \langle P, E[o_P\,\texttt{.fd=o}], S_P \rangle, PS \hookrightarrow \langle P, E[o_P], S_P[o_P \mapsto (\texttt{ro}, \texttt{object} \mapsto o)] \rangle, \langle S_Q[o_Q \mapsto (\texttt{fd} \mapsto o')] \rangle, PS}$$

RO-CALL

$$\frac{\langle Q, o_Q.\texttt{m}(o_1', \ldots, o_n'), S_Q \rangle, PS \hookrightarrow^* \langle Q, o, S_Q' \rangle, PS}{\langle P, E[o_P\,\texttt{[m]}(\langle p_1 = o_1 \rangle, \ldots, \langle p_n = o_n \rangle)], S_P \rangle, \langle Q, e, S_Q \rangle, PS \hookrightarrow \langle P, E[o_P], S_P[o_P \mapsto (\texttt{ro}, \texttt{object} \mapsto o)] \rangle, \langle Q, e, S_Q' \rangle, PS}$$

RO-VALUE

$$\langle P, E[\texttt{valueOf}(o_P)], S_P \rangle, PS \hookrightarrow \langle P, E[o], S_P[o_P \mapsto (\texttt{ro}, \texttt{object} \mapsto o)] \rangle, PS$$

RO-INSPRT

$$\frac{o_Q \notin \text{dom}(S_Q) \qquad S_P(o_P) = (\texttt{ro}, \ \texttt{object} \mapsto o_Q) \qquad t, s \in \texttt{String}}{\langle P, E[o_P\,\texttt{.InsertPart(t,s)}], S_P \rangle, \langle Q, e, S_Q \rangle, PS \hookrightarrow \langle P, E[o_P], S_P[o_P \mapsto (\texttt{ro}, \texttt{object} \mapsto o_Q')] \rangle, \langle Q, e, S_Q[o_Q' \mapsto (t, F \mapsto \texttt{null})] \rangle, PS}$$

where $S[o \mapsto (C, \diamond \mapsto S \cup o_Q')]$ is applied to both RO-INSPRT and RO-INSFLD

RO-INSFLD

$$\frac{o_Q \notin \text{dom}(S_Q) \qquad S_P(o_P) = (\texttt{ro}, \ \texttt{object} \mapsto o_Q) \qquad t, s \in \texttt{String}}{\langle P, E[o_P\,\texttt{.InsertField(t,s)}], S_P \rangle, \langle Q, e, S_Q \rangle, PS \hookrightarrow \langle P, E[o_P], S_P[o_P \mapsto (\texttt{ro}, \texttt{object} \mapsto o_Q')] \rangle, \langle Q, e, S_Q[o_Q' \mapsto (t, F \mapsto \texttt{null})] \rangle, PS}$$

RO-DELPRT

$$\frac{\texttt{parts}(o_Q, t) = o_Q' \in \text{dom}(S_Q) \qquad S_P(o_P) = (\texttt{ro}, \ \texttt{object} \mapsto o_Q) \qquad t \in \texttt{String}}{\langle P, E[o_P\,\texttt{.DeletePart(t)}], S_P \rangle, \langle Q, e, S_Q \rangle, PS \hookrightarrow \langle P, E[o_P], S_P[o_P \mapsto (\texttt{ro}, \texttt{object} \mapsto o_Q)] \rangle, S[o \mapsto (C, \diamond \mapsto S \setminus o_Q')], PS}$$

RO-DELFLD

$$\frac{\texttt{fieldByName}(o_Q, t) = o_Q' \in \text{dom}(S_Q) \qquad S_P(o_P) = (\texttt{ro}, \ \texttt{object} \mapsto o_Q) \qquad t \in \texttt{String}}{\langle P, E[o_P\,\texttt{.DeleteField(t)}], S_P \rangle, \langle Q, e, S_Q \rangle, PS \hookrightarrow \langle P, E[o_P], S_P[o_P \mapsto (\texttt{ro}, \texttt{object} \mapsto o_Q)] \rangle, S[o \mapsto (C, \diamond \mapsto S \setminus o_Q')], PS}$$

Figure 3.5: Reduction rules of FOREL.

A ROPE statement executed by one program changes not only the state of this program, but also states of other programs to which this ROPE instance is connected. We write $\hookrightarrow^*$ for the reflexive, transitive closure of $\hookrightarrow$. Most of the rules are standard; the interesting features are how they manipulate the FOREL type system. The RO-NEW rule reduces a new expression into a ROPE reference to object o_r that is a collection of root objects $o_r \subseteq V'$ in an object graph O that describes a foreign program. When the RO-NEW rule is applicable the foreign program to which the connection is established is unknown and object o_r has an empty collection of root objects. When the location of a schema that describes a foreign program is known, we use the RO-LOCNEW rule and object o_r has a nonempty collection of root objects.

There are five rules for navigating objects in foreign programs. The RO-OLGET rule returns a collection of objects using helper function parts, and the names of the objects in the returned collection match the name that is supplied as a parameter to this function. The RO-ONGET rule returns an object in the collection of objects by the order sequence number using helper function part. The RO-FLGET and RO-FNGET rules produce similar effects on fields using helper functions fieldByName and fieldBySeqNum. The RO-FOSET rule is straightforward, updating the field of the foreign object with the value o written to field fd. The helper function Ω maps value o from a native program to a corresponding value o′ in a foreign program. When applying this rule the state of native program P, stays unchanged while the foreign program Q switches to a new state . The invocation rules use the helper functions to determine the object, fields, and type values and names as well as the conversion of values between foreign and native programs. The full semantics of the language include other rules including error rules representing casts that fail and null pointer dereferences. A set of congruence rules (such as if $e \rightarrow e'$ then $e.f \rightarrow e'.f$)

allows reduction to proceed in the order of evaluation defined by `ClassicJava`.

Typing Rules

Typing judgments, shown in Figure 3.7, are of the form $\Gamma \vdash$ `e:T` , read "In the type environment Γ, expression `e` has type `T`". The `T-FLGET` and `T-FNGET` rules obtain the type of a reification expression in Γ, replacing the grammar expression `e(e)` with its value of some type. The expression in the parenthesis may evaluate to a string or a number variable. Similarly, the `T-OLGET1`, `T-OLGET2` and `T-ONGET` rules looks up the type of an object in a foreign program, producing its equivalent type in the native program. There are also four typing rules for modifications of foreign objects-the insertion rules `T-IORT` and `T-IFLD`, which insert fields and parts into foreign programs, and the deletion rules `T-DPRT` and `T-DFLD`, which remove parts and fields in foreign programs. FOREL follows `ClassicJava`'s lookup rules for method types and method bodies. Type lookup rule `matchType` performs a lookup of type δ in the native program that corresponds to a type in the foreign program described by its name that is a parameter to this function.

Helper Functions

Most of the helper functions are straightforward. The `root` function returns the root object o_r in a given program as defined in Section 3.3.2. Helper functions `part`, `parts`, `fieldByName`, and `fieldBySeqNum` are shown in Figure 3.6. In a foreign program the `parts` function maps an object and the name of a child object or part to an object that represents a list of children objects or parts that have this name. The `part` function maps a list of objects of different types and a sequence number to an object or part of some type that occupies the position in the collection defined by the sequence number. Correspond-

$$\langle \mathtt{Q}, \mathtt{S_Q} \rangle, \mathtt{getObjectGraph(Q)=G} \vdash \mathtt{root(G)} \;=\; \mathtt{o_R}$$

$$\frac{\mathtt{S_Q} = (\mathtt{o_k} = (\mathtt{C}, \mathtt{F})) \qquad \mathtt{t} \in \mathtt{String}}{\langle \mathtt{Q}, \mathtt{S_Q} \rangle \vdash \mathtt{parts}(\mathtt{o_Q}, \mathtt{t}) = \mathtt{o_k}, \mathtt{Class}(\mathtt{o_k}) \;=\; \mathtt{t}}$$

$$\frac{\mathtt{S_Q} = (\mathtt{o_1} = (\mathtt{C_1}, \mathtt{F_1}), \ldots, \mathtt{o_k} = (\mathtt{C_k}, \mathtt{F_k})) \qquad 1 \leq \mathtt{n} \leq \mathtt{k} \qquad \mathtt{n}, \mathtt{k} \in \mathbf{N}}{\langle \mathtt{Q}, \mathtt{S_Q} \rangle \vdash \mathtt{part}(\mathtt{o_Q},\ \mathtt{n}) = \mathtt{o_i}}$$

$$\frac{\mathtt{S_Q} = (\mathtt{C}, \mathtt{F}\{\mathtt{f_1} \mapsto \mathtt{o_1}, \ldots, \mathtt{f_k} \mapsto \mathtt{o_k}\}) \qquad \mathtt{t} \in \mathtt{String} \qquad 1 \leq \mathtt{i} \leq \mathtt{k} \qquad \mathtt{t} = \mathtt{f_i}}{\langle \mathtt{Q}, \mathtt{S_Q} \rangle \vdash \mathtt{fieldByName}(\mathtt{o_Q}, \mathtt{t}) = \mathtt{o_i}}$$

$$\frac{\mathtt{S_Q} = (\mathtt{C}, \mathtt{F}\{\mathtt{f_1} \mapsto \mathtt{o_1}, \ldots, \mathtt{f_k} \mapsto \mathtt{o_k}\}) \qquad \mathtt{n} \in \mathbb{N} \qquad 1 \leq \mathtt{i} \leq \mathtt{k} \qquad \mathtt{n} = i}{\langle \mathtt{Q}, \mathtt{S_Q} \rangle \vdash \mathtt{fieldBySeqNum}(\mathtt{o_Q}, \mathtt{n}) = \mathtt{o_i}}$$

Figure 3.6: Helper functions used in foreign programs.

ingly, `field` lookup functions perform retrievals of fields and their types by their names or sequence numbers in the collection of fields for a given object.

Type Soundness

We can show the type soundness of FOREL through two standard theorems, preservation and progress. Type soundness implies that the language's type system is well behaved. In a type-safe language like `ClassicJava`, well-typed programs do not get stuck, that is they pass the type checking algorithm successfully or halt with errors (progress). If a well-typed expression is evaluated, then the resulting expression is also well typed (preservation).We state the progress and preservation theorems and give their proofs below.

Theorem 1 (Preservation)**.** *If* $\oslash \vdash e : T$*, and* $e \hookrightarrow e'$*, then* $\oslash \vdash e' : T$*.*

Proof. Preservation is proved by induction on the rules defining the transition system for

$$\text{T-FLGET}\quad \frac{\Gamma \vdash \mathtt{e} : \tau \qquad \mathtt{E} \vdash \tau \xrightarrow{f} \delta}{\Gamma \vdash \mathtt{e}(f) : \delta}$$

$$\text{T-FNGET}\quad \frac{\Gamma \vdash \mathtt{e} : \tau \qquad \Gamma \vdash \mathtt{n} : \mathbb{N}}{\Gamma \vdash \mathtt{e}(\mathtt{n}) : \tau}$$

$$\text{T-OLGET1}\quad \frac{\Gamma \vdash \mathtt{e} : \tau \qquad \mathtt{E} \vdash \tau \xrightarrow{\diamond} \delta \qquad \mathtt{matchType}(\mathtt{t}) = \delta}{\Gamma \vdash \mathtt{e}[\mathtt{t}] : \delta}$$

$$\text{T-OLGET2}\quad \frac{\Gamma \vdash \mathtt{e} : \tau \qquad \mathtt{E} \vdash \tau \xrightarrow{f} \delta}{\Gamma \vdash \mathtt{e}[f] : \delta}$$

$$\text{T-ONGET}\quad \frac{\Gamma \vdash \mathtt{e} : \tau \qquad \Gamma \vdash \mathtt{n} : \mathbb{N}}{\Gamma \vdash \mathtt{e}[\mathtt{n}] : \tau}$$

$$\text{T-INVK}\quad \frac{\Gamma \vdash \mathtt{e} : \sigma_1, \ldots, \sigma_n \to \tau \qquad \mathtt{for\ each\ i} \in \mathbb{N}, \Gamma \vdash \mathtt{p}_i : \sigma_i \wedge \mathtt{v}_i : \sigma_i}{\Gamma \vdash \mathtt{e}(\langle \mathtt{p}_1 = \mathtt{v}_1 \rangle \ldots \langle \mathtt{p}_n = \mathtt{v}_n \rangle) : \tau}$$

$$\text{T-IORT}\quad \frac{\mathtt{matchType}(\mathtt{d}) = \delta \qquad \Gamma \vdash \mathtt{e} : \tau \qquad \Gamma \vdash \mathtt{InsertPart} : t \times \delta \to \delta}{\Gamma \vdash \mathtt{e.InsertPart(t,d)} : \delta \qquad \mathtt{E} \vdash \tau \xrightarrow{\diamond} \delta}$$

$$\text{T-IFLD}\quad \frac{\mathtt{matchType}(\mathtt{d}) = \delta \qquad \Gamma \vdash \mathtt{e} : \tau \qquad \Gamma \vdash \mathtt{InsertField} : f \times \delta \to \delta}{\Gamma \vdash \mathtt{e.InsertField(f,d)} : \delta \qquad \mathtt{E} \vdash \tau \xrightarrow{f} \delta}$$

$$\text{T-DPRT}\quad \frac{\mathtt{matchType}(\mathtt{d}) = \delta \qquad \Gamma \vdash \mathtt{e} : \tau \qquad \Gamma \vdash \mathtt{DeletePart} : \mathtt{d} \to \tau \qquad \mathtt{E} \vdash \tau \xrightarrow{\diamond} \delta}{\Gamma \vdash \mathtt{e.DeletePart(d)} : \tau \qquad \mathtt{E} \not\vdash \tau \xrightarrow{\diamond} \delta}$$

$$\text{T-DFLD}\quad \frac{\Gamma \vdash \mathtt{e} : \tau \qquad \Gamma \vdash \mathtt{DeleteField} : \mathtt{f} \to \tau \qquad \mathtt{E} \vdash \tau \xrightarrow{f} \delta}{\Gamma \vdash \mathtt{e.DeleteField(f)} : \tau \qquad \mathtt{E} \not\vdash \tau \xrightarrow{f} \delta}$$

$$\text{T-RONEW}\quad \Gamma \vdash \mathtt{new\ ro:\ \ ro}$$

$$\text{T-ROLNEW}\quad \Gamma \vdash \mathtt{new\ [L]\ ro:\ \ ro}$$

$$\text{T-ERROR}\quad \Gamma \vdash \textbf{error} \vdash \mathtt{ERROR}$$

$$\text{T-NULL}\quad \Gamma \vdash \textbf{null} \vdash \mathtt{NULL}$$

Figure 3.7: FOREL typechecking.

step-by-step evaluation of ROPE expressions.

FLGET By induction let us assume that if $\texttt{e} : \tau$, then $\texttt{e'} : \tau$. Assume that $\texttt{e(f)} : \delta$. From the typing rule we have $\texttt{e} : \tau$. By induction $\texttt{e'} : \tau$ and $E \vdash \tau \xrightarrow{f} \delta$, so by typing rule $\texttt{T-FLGET e'(f)} : \delta$.

FNGET By induction let us assume that if $\texttt{e} : \tau$, then $\texttt{e'} : \tau$. Assume that $\texttt{e(n)} : \tau$. From the typing rule we have $\texttt{e} : \tau$. By induction $\texttt{e'} : \tau$ and $\texttt{n} \in \mathbb{N}$, so by typing rule $\texttt{T-FNGET e'(n)} : \tau$.

OLGET1 By induction let us assume that if $\texttt{e} : \tau$, then $\texttt{e'} : \tau$. Assume that $\texttt{e[t]} : \delta$. From the typing rule we have $\texttt{e} : \tau$. By induction $\texttt{e'} : \tau$ and $E \vdash \tau \xrightarrow{\diamond} \delta$, so by typing rule $\texttt{T-OLGET1 e'[t]} : \delta$.

OLGET2 By induction let us assume that if $\texttt{e} : \tau$, then $\texttt{e'} : \tau$. Assume that $\texttt{e[f]} : \delta$. From the typing rule we have $\texttt{e} : \tau$. By induction $\texttt{e'} : \tau$ and $E \vdash \tau \xrightarrow{f} \delta$, so by typing rule $\texttt{T-OLGET2 e'[f]} : \delta$.

ONGET By induction let us assume that if $\texttt{e} : \tau$, then $\texttt{e'} : \tau$. Assume that $\texttt{e[n]} : \tau$. From the typing rule we have $\texttt{e} : \tau$. By induction $\texttt{e'} : \tau$ and $\texttt{n} \in \mathbb{N}$, so by typing rule $\texttt{T-ONGET e'[n]} : \tau$.

INVK By induction let us assume that if $\texttt{e} : \sigma_1, \ldots, \sigma_n \to \tau$, then $\texttt{e'} : \sigma_1, \ldots, \sigma_n \to \tau$. Assume that $\texttt{e}\,(\langle \texttt{p}_1 = \texttt{v}_1 \rangle \ldots \langle \texttt{p}_\texttt{n} = \texttt{v}_\texttt{n} \rangle) : \tau$. From the typing rule we have $\texttt{e} : \sigma_1, \ldots, \sigma_n \to \tau$. By induction $\texttt{e'} : \sigma_1, \ldots, \sigma_n \to \tau$ and , so by typing rule $\texttt{T-INVK e'}\,(\langle \texttt{p}_1 = \texttt{v}_1 \rangle \ldots \langle \texttt{p}_\texttt{n} = \texttt{v}_\texttt{n} \rangle) : \tau$.

IORT By induction let us assume that if $\texttt{e} : \tau$, then $\texttt{e'} : \tau$. Assume that $\texttt{e.InsertPart} : t \times \delta \to \delta$. From the typing rule we have $\texttt{e} : \tau$. By induction $\texttt{e'} : \tau$ so by typing rule

T-IORT e′.InsertPart:$t \times \delta \rightarrow \delta$.

IFLD By induction let us assume that if e:τ, then e′:τ. Assume that e.InsertField:$f \times \delta \rightarrow \delta$. From the typing rule we have e:τ. By induction e′:τ so by typing rule T-IFLD e′.InsertField:$f \times \delta \rightarrow \delta$.

DPRT By induction let us assume that if e:τ, then e′:τ. Assume that e.DeletePart:$d \rightarrow \tau$. From the typing rule we have e:τ. By induction e′:τ so by typing rule T-DPRT e′.DeletePart:$d \rightarrow \tau$.

DFLD By induction let us assume that if e:τ, then e′:τ. Assume that e.DeleteField:$f \rightarrow \tau$. From the typing rule we have e:τ. By induction e′:τ so by typing rule T-DFLD e′.DeleteField:$f \rightarrow \tau$.

□

Theorem 2 (Progress)**.** *If* $\Gamma \vdash e : T$*, then either* e *is an irreducible value, contains an error subexpression, or else* $\exists\ e'$ *such that* $e \hookrightarrow e'$.

Proof. The proof is by induction on the rules of the type checking. We consider the following cases.

T-FLGET Let e′=e(f) and assume that e′:τ. Since e′ is not a value, we must show that $\exists$e′′ such that e′ $\hookrightarrow$ e′′. By induction we have that either e is a value or $\exists\varepsilon$ such that e$\hookrightarrow \varepsilon$. In the latter case it follows that e(f)$\hookrightarrow \varepsilon$(f). In the former we have v(f)$\hookrightarrow$v.

T-FNGET Let e′=e(n) and assume that e′:τ. Since e′ is not a value, we must show that $\exists$e′′ such that e′ $\hookrightarrow$ e′′. By induction we have that either e is a value or $\exists\varepsilon$ such that e$\hookrightarrow \varepsilon$. In the latter case it follows that e(n)$\hookrightarrow \varepsilon$(n). In the former we have v(n)$\hookrightarrow$v.

T-OLGET Let $e' = e[t]$ and assume that $e' : \delta$. Since e' is not a value, we must show that $\exists e''$ such that $e' \hookrightarrow e''$. By induction we have that either e is a value or $\exists \varepsilon$ such that $e \hookrightarrow \varepsilon$. In the latter case it follows that $e[t] \hookrightarrow \varepsilon[t]$. In the former we have $v[t] \hookrightarrow v$.

T-ONGET Let $e' = e[n]$ and assume that $e' : \tau$. Since e' is not a value, we must show that $\exists e''$ such that $e' \hookrightarrow e''$. By induction we have that either e is a value or $\exists \varepsilon$ such that $e \hookrightarrow \varepsilon$. In the latter case it follows that $e[n] \hookrightarrow \varepsilon[n]$. In the former we have $v[n] \hookrightarrow v$.

T-INVK Let $e' = e\langle(p_1 = v_1 \ldots p_n = v_n)\rangle$ and assume that $e' : \sigma_1, \ldots, \sigma_n \rightarrow \tau$. Since e' is not a value, we must show that $\exists e''$ such that $e' \hookrightarrow e''$. By induction we have that either e is a value or $\exists \varepsilon$ such that $e \hookrightarrow \varepsilon$. In the latter case it follows that $e\langle(p_1 = v_1 \ldots p_n = v_n)\rangle \hookrightarrow \varepsilon\langle(p_1 = v_1 \ldots p_n = v_n)\rangle$. In the former we have $v\langle(p_1 = v_1 \ldots p_n = v_n)\rangle \hookrightarrow v\langle(p_1 = v_1 \ldots p_n = v_n)\rangle$.

T-IORT Let $e' = e.\texttt{InsertPart}(t, d)$ and assume that $e' : \delta$. Since e' is not a value, we must show that $\exists e''$ such that $e' \hookrightarrow e''$. By induction we have that either e is a value or $\exists \varepsilon$ such that $e \hookrightarrow \varepsilon$. In the latter case it follows that $e.\texttt{InsertPart}(t, d) \hookrightarrow \varepsilon.\texttt{InsertPart}(t, d)$.

In the former we have $v.\texttt{InsertPart}(t, d) \hookrightarrow v$.

T-IFLD Let $e' = e.\texttt{InsertField}(f, d)$ and assume that $e' : \delta$. Since e' is not a value, we must show that $\exists e''$ such that $e' \hookrightarrow e''$. By induction we have that either e is a value or $\exists \varepsilon$ such that $e \hookrightarrow \varepsilon$. In the latter case it follows that $e.\texttt{InsertField}(f, d) \hookrightarrow \varepsilon.\texttt{InsertField}(f, d)$.

In the former we have $v.\texttt{InsertField}(f, d) \hookrightarrow v$.

T-DPRT Let $e' = e.\texttt{DeletePart}(d)$ and assume that $e' : \tau$. Since e' is not a value, we must show that $\exists e''$ such that $e' \hookrightarrow e''$. By induction we have that either e is a value

or $\exists\varepsilon$ such that $e \hookrightarrow \varepsilon$. In the latter case it follows that

`e.DeletePart(d)` $\hookrightarrow$ ε`.DeletePart(d)`.

In the former we have `v.DeletePart(d)`$\hookrightarrow$`v`.

T-DFLD Let `e'=e.DeleteField(f)` and assume that `e'`$:\tau$. Since `e'` is not a value, we must show that $\exists$`e''` such that `e'` $\hookrightarrow$ `e''`. By induction we have that either `e` is a value or $\exists\varepsilon$ such that `e`$\hookrightarrow \varepsilon$. In the latter case it follows that `e.DeleteField(f)`$\hookrightarrow$ ε`.DeleteField(f)`.

In the former we have `v.DeleteField(f)`$\hookrightarrow$`v`. □

3.4 Type Inference

The task of static checking of FOREL programs is greatly simplified when the names of foreign components names are defined as string constants. In this case type checking rules shown in Figure 3.7 are applied directly to ROPEs to validate their correctness. However, if the names of some foreign components are specified using string expressions, then the values of these expressions may or may not be determined at compile time. A string analysis framework (e.g., Soot) may be used to recover the names of foreign components using an algorithm for string expression analysis, and if this attempt is successful, then the type system is used to perform type checking on the recovered names. However, if the names of foreign components are not known at compile time, then type graphs are used to perform the last step of the analysis to infer types of expressions or variables that hold the names of foreign components. This analysis is based on the *Traversal Graph Analysis (TGA)* defined in adaptive programming [64].

3.4.1 Strategies

An adaptive strategy S=(R, π, δ) represents paths in an object graph, where R={s, d}, where s and d are the source and target objects (components) of a path in an object graph, and R $\subseteq$ O, where O is the set of objects in a type graph, π = {e, α}, where e is a set of fields and α is a set of variables that designate a set of some edges $\alpha \subseteq$ e, and $\delta = \{\rightharpoonup, \rightharpoondown\}$ is a set of transition edges representing objects and attributes respectively. Each element in a strategy S is either the name of some foreign object or a variable designating some foreign object or attributes.

We write π(o, o′) to designate a set of objects {o′}, such that each object o′ of this set is a part of the object o expressed by some edge e $\in \pi$ such that e(o, o′). The basic idea of transforming reification statements into strategies is in defining strategy graph edges a$\rightharpoonup$b and a$\rightharpoondown$b for reification statements x["a"]["b"] and x["a"].attribute("b") respectively. Thus, a strategy is an abstraction of reification statements, and it is also an abstraction of a set of paths in the type graphs.

For example, strategy CEO $\rightharpoonup \alpha_1 \rightharpoonup \alpha_2 \rightharpoondown$ amount for reification expression x["CEO"][strexp1][strexp2].attribute("amount") for the type graph shown in Figure 3.3 designates strategy S, where s=CEO, d=amount, α_1 is a variable designating objects computed via string expression strexp1, and α_2 is a variable designating attribute object computed via string expression strexp2. Computing π(CEO, o′) we obtain {CTO}, and computing π(CTO, o′) we obtain {CEO, check}.

Each node in a strategy is assigned a distinct sequence number, and nodes are expressed as pairs (i, π). We introduce functions $\Delta_i : \mathbb{N} \times \mathbb{N} \rightarrow \delta$ and $\Delta_\pi : \pi \times \pi \rightarrow \delta$. Given two sequential natural numbers k and k+1, the function Δ_i computes the transition edge between nodes that are assigned these numbers in S, or $\oslash$ if there is no transition edge.

Correspondingly, given two nodes π_q and π_r in some type graph, function Δ_π computes the transition edge between nodes, or $\oslash$ if there is no transition edge.

3.4.2 The Algorithm

When the values of string expressions in reification statements cannot be computed at compile time, they can be inferred using the TGA-based algorithm `BuildPathTree`. It takes a set of reification statements and type graphs as its inputs and transforms each reification statement into an adaptive strategy with variables replacing string expressions. `BuildPathTree` computes possible values for each variable and generates traversal paths for each strategy. If no path exists between the source and the destination objects, then a type error is reported. If at least one path is generated, then the FOREL compiler issues warnings, since values of expressions that compute names of foreign objects may not be in the computed paths.

The TGA-based algorithm `BuildPathTree` for computing valid paths for reification expressions and statements is shown in Algorithm 1. The basic idea of this algorithm is to compute the set of edges `e` for each pair of classes `c` and `c'`, by computing `FIRST(c, c')` iff it is possible for an object of type `c` to reach an object of type `c'` by a path beginning with an edge `e`. Recall from Section 3.3.4 that `FIRST(c, c')`$=e \in E$, such that there exists an object graph `O` of `C` and objects `o` and `o'` such that:

1. `Class(o) = c`,
2. `Class(o') = c'`, and
3. `o` e^* `o'`.

The last condition, `o` e^* `o'` says that there is ($\exists$) a path from `o` to `o'` in the

object graph, consisting of an edge labeled `e`, followed by any sequence of edges in the graph. The method `FIRST` is implemented using two procedures: `BuildPathTree` and `ComputePath`.

Procedure `BuildPathTree` takes the set `R` of source and target components in `S` and set π as input parameters. The output of this procedure is a tree of valid paths in a type graph that satisfy a given strategy. Some of the input components may not make it into the path tree because they do not start any valid paths.

Procedure `BuildPathTree` calls procedure `ComputePath`, which in turn recursively calls itself. `ComputePath` takes three parameters: a component `o` that is a potential current node in the path, sequence number `i` of the node in the strategy `S`, and the transition edge δ between nodes in `S` that are assigned two sequential natural numbers `i` and `i+1`. The goal of this procedure is to color the potential current node `o` in the path as either `red` or `blue`. When colored `red` object `o` is considered a dead end on the path in the type graph that does not lead to the designated target nodes. Otherwise, it is colored `blue` and this color is propagated up to the source nodes which are subsequently included in the path tree.

The termination condition for procedure `ComputePath` is defined as the sequence number `i` being equal to or greater of the number of nodes in the strategy, $|\pi|$, or if there is no transition edge from the current node. When reaching the termination condition we color the current node `blue` and return from the procedure. In the calling procedure we check the color of the node, and if it is `blue`, then we attach this node to its parent node in the path tree.

Algorithm 2 ComputePath procedure

BuildPathTree(R $\in$ S, $\pi \in$ S)
for all s $\in$ R **do**
  ComputePath(s, 0, $\Delta_i(0,1)$)
  if color(s) = red **then**
    remove s from R
  end if
end for

ComputePath(o $\in$ O, $i \in \mathbb{N}$, $\partial \in \delta$)
if $i \geq |\pi|$ or $\partial = \oslash$ **then**
  color(o) $\mapsto$ blue
else
  for all o' $\in \pi_i$(o, o') **do**
    if Δ_π(o,o') = ∂ **then**
      ComputePath(o', $i+1$, $\Delta_i(i, i+1)$)
      if color(o') = blue **then**
        AddChildToTree(o, o')
      end if
    end if
  end for
  if children(o) = $\oslash$ **then**
    color(o) $\mapsto$ red
  else
    color(o) $\mapsto$ blue
  end if
end if

3.4.3 Pruning and Generating Paths

The algorithm `BuildPathTree` shown in Algorithm 1 computes the set of edges `e` for each pair of classes `c` and `c'`, if it is possible for an object of type `c` to reach an object of type `c'` by a path beginning with an edge `e`. This algorithm is applied individually to each ROPE expression in which foreign objects are specified using string expressions whose values are not known at compile time. This algorithm infers possible names of foreign

objects that string expressions make compute at runtime.

As it often happens, the same string expressions are used in different ROPE expressions in the same program scope. Clearly, the same expressions compute the same values if they are located in the same scope provided that the values of the variables used in these expressions are not changed. Using program analysis techniques it is possible to detect expressions at compile time whose variables are not changed at runtime. After running the algorithm `BuildPathTree`, it computes possible names of foreign objects that string expressions can take at ROPE statements. Given the same expression used in different ROPE statements in the same program scope, and provided that the values of the variables used in these expressions are not changed by other expressions executed between these ROPEs, it is possible to give more precise set of names of foreign objects computed by these string expressions. This more precise set is obtained by taking the intersection of the sets of names computed by the algorithm `BuildPathTree`. We explain this process below.

Consider the strategy graph S_1 `CEO` $\rightharpoonup \alpha_1 \rightharpoonup \alpha_2 \rightharpoondown$ `amount` for reification expression `x["CEO"][strexp1][strexp2].attribute("amount")` for the type graph shown in Figure 3.3. By applying our algorithm we compute values for type scheme variables $\alpha_1 = \{$`CTO`$\}$ and $\alpha_2 = \{$`boss, salary`$\}$. Suppose we have different strategy graph S_2, `Programmer` $\rightharpoonup \alpha_2 \rightharpoondown$ `bonus` for reification expression `y["Programmer"][strexp2].attribute("bonus")` for some other type graph. String expression variable `strexp2` is the same in both reification statements, and because of that it is designated by the same type scheme variables in the strategy graphs. Suppose that by applying `BuildPathTree` algorithm values for type scheme variable α_2 = $\{$`salary`$\}$ are computed. In order to determine the value of variable α_2 that satisfies both S_1 and S_2 we take the intersection of the sets of values of α_2 computed for these two

strategies. The resulting set $\alpha_2 = \{\texttt{salary}\}$ is the result of pruning the navigation paths.

This example illustrates the idea of pruning navigation paths using context-sensitive dataflow analysis. By determining definitions and uses of a variable that designate names of foreign objects in a given scope, sets of values are computed for each reification statements in which this variable is used. Then the intersection of these sets is taken to determine common values that this variable can take in the scope considered.

3.4.4 Communication Integrity

Communication integrity is an important criterion for architectural conformance [65]. In the context of interoperating components it specifies that each component in the implementation may only communicate directly with the programs to which it is connected in the architecture of a system of interoperating components. Compositions of ROPEs should not violate communication integrity.

Our solution ensures the communication integrity of interoperating components by analyzing compositions of ROPEs to build the transitive relations between programs in polylingual systems. For example, reification statement in program `P`, `x["y"]["z"]` navigates to the field `z` of foreign object `y` in program `Q` denoted by ROPE `x`. However, object `y` is an instance of a ROPE defined in program `Q` that denotes some foreign object in program `R` whose field `z` is accessed. Thus, we may violate the communication integrity by implicitly interoperating programs `P` and `R` via program `Q` even though this communication may be prohibited by the constraints of a given architecture.

We encode architectural constraints when defining instances of ROPEs in FOREL using the keyword `constraints` as part of the ROPE expressions that instantiate reification operators. These constraints define applications with which a given program can

interoperate. An example is a statement that specifies a constraint is `ro ["P"] x = new ro constraints ["Q"]`. This constraint effectively prohibits the program `P` to communicate with other programs but `Q`, explicitly or implicitly. Our static analysis algorithm ensures that such constraints hold by keeping a table of constraints imposed on interoperating programs and issuing errors when these constraints are violated.

3.4.5 Computational Complexity

The time complexity of `BuildPathTree` algorithm is exponential to the size of the type graph for each reification statement in a FOREL program. Since the algorithm involves the search of all nodes and edges in the type graph that contains cycles for each node in the strategy, its complexity is $O((\mathtt{V}+\mathtt{E})^{\mathtt{max}(|\pi|)})$ where `V` is the number of nodes, `E` is the number of edges in the type graph, and $\mathtt{max}(|\pi|)$ is the maximum number of nodes in strategies. The operations of putting successors in the table of variables take $O(1)$.

In general, the number of nodes $\mathtt{max}(|\pi|)$ in strategies is much smaller than the number of nodes in type graphs. It is also rare that all graph nodes have to be explored for each node in a strategy. The theoretical limit on computational complexity of `BuildPathTree` algorithm is exponential. However, our experimental evaluation showed that in practice the running time of the algorithm is small and does not exceed one minute for large schemas because typically path expressions are short.

3.5 The Prototype Implementation

Our prototype implementation included the FOREL compiler and our static checking algorithms. We wrote the FOREL compiler in C++. We extended the C++ and Java grammars with FOREL syntax with the ProGrammar visual environment for building parsers that

are platform-independent, programming language-independent and reusable [12]. Our implementation contains less than 3,000 lines of code. Its analysis routines detect errors in foreign components using type checking rules and static analysis algorithms as specified in Section 3.3.5 and Section 3.4.

Since it puts additional burden on programmers to create formal descriptions of foreign components, we automated this process by extracting type graphs and XML schemas from programs automatically using different tools [70][22]. These tools accept instances of foreign components and output type graphs or XML schemas that are used by the FOREL compiler. The latter interfaced with these tools and used the extracted type graphs and schemas to performed its functions.

Navigation through an object structure can be done with and without meta information (schema) about the structure. With the schema known, we get the benefits of type checking but also the benefit of faster execution of the navigation expressions.

3.6 Experimental Evaluation

Our goal in evaluating FOREL is to determine how effective FOREL type checking is. We applied FOREL to a real-world commercial project, to a program written by a student, and to two commercial programs that used large-scale schemas in two different domains. We report the results of these evaluations in this section.

3.6.1 Archer Analyzer

We applied our approach to the *Archer Analyzer (AA)*, a software package geared for the `Archer 10` optical overlay metrology systems manufactured by California-based KLA-Tencor Corporation [10][9]. The purpose of optical overlay measurements is to detect and

fix misalignments between layers of semiconductor chips that were put on a silicon wafer using microlithography processes.

AA is created as an open system and its interoperating components are hosted by such platforms as EJB, CORBA, and .Net assemblies. The components for AA are created using C++ and different low-level APIs were used for parsing XML and HTML data, invoking Java methods using *Java Native Interface (JNI)*, and interoperating with CORBA and .Net components.

The first release of AA occurred in June, 2001, and its testing continued through the September of 2001. FOREL compiler was not created at that time, and ROPEs were implemented as a library using C++ templates. Bugs were detected during the testing phase manually by a group of test engineers. Found bugs for AA representative programs are shown in Table 3.1. For example, `DbDataAdapter.cpp` is a program that contains `2,151` lines of code. It interoperates via three ROPEs with other programs and data whose schemas collectively contain `826` types. `DbDataAdapter.cpp` contains `295` ROPE statements that reference `147` foreign objects.

Manual testing of the first release uncovered fourteen bugs in this program that could be grouped in seven categories:

- Wrong names of referenced foreign component;
- Wrong operations on foreign components;
- Wrong assumptions about types of foreign components;
- References to wrong parts of schemas;
- Violations of communication integrity;

C++ Program (A) Archer Analyzer (S) Student Work (R) Reengineered	Size					Analysis Time, sec		Bugs Detected		
	Program, LOC	Schema, No. of Types	No. of ROPE Stmts	No. of Refd Objs	Foreign Progs	Generating, Type Gr.	Semant. Analysis	Testing Phase	With FOREL	Confirmed Errors
DbDataAdapter (A)	2151	826	295	147	3	3.6	2.8	14	53	26
FindRecipeView (A)	754	49	32	50	5	0.2	2.3	5	12	11
RecipeManager (A)	913	826	114	68	17	3.6	5.2	1	9	6
CDBQueueSettings (A)	224	16	35	24	2	0.03	0.8	12	12	12
CXMLOverlayLotData (A)	453	669	72	181	8	3.1	3.7	3	18	9
DataTestGenerator (S)	338	1653	31	115	12	11.2	1.2	8	13	8
papiNet (R)	7938	1653	313	853	1	11.2	7.8	-	49	31
MetaLex (R)	13128	66	25	19	1	0.3	2.2	-	5	2

Table 3.1: Experimental results

- Violations of schema constraints.

Last violation requires an additional explanation. Suppose that a schema specifies that some XML object may not contain more than certain number of children of some type. We found that this constraint was frequently violated, and these violations led to dangerous consequences. Instead of programs failing right away, they continued to run and produce incorrect data leading to failures in different components that used this data. This separation of cause and effect both temporally and spatially made it very difficult to localize these bugs and fix it.

Two and half years later the FOREL compiler was completed and applied to the first release of AA. Since we already knew what bugs were discovered during the testing, we were interested to see how our FOREL compiler performs with respect to human effort. While approximately three months of manual regression testing of `DbDataAdapter.cpp` revealed fourteen bugs, FOREL compiler in less than seven seconds discovered `53` bugs, `26` of which were confirmed including those found through testing. Similar results were obtained for other programs thus confirming the viability of our approach.

3.6.2 papiNet and Metalex

While the results of the evaluation of our approach proved successful for AA, we wanted to evaluate FOREL on systems from different domains, in order to answer the following experimental questions:

- Is the FOREL typechecking practical on industrial large schemas that contain over 1,000 different types?
- Does the FOREL abstraction make it easier to reengineer existing applications?

Methodology. We performed a case study during which we reengineered existing applications for legal and paper supply chain domains. The former application was written for a legal office, and it used `Metalex` schema [51]. `MetaLex` is an open XML standard for the markup of legal sources. The latter application was written by a now defunct startup company for papiNet, a transaction standard for the paper and forest supply chain [52]. The combined source code of both applications was about `30,000` lines of C++ code.

The intention of this study is to manually reengineer legacy systems with interoperating components to evaluate the effort and to see whether our FOREL compiler can find bugs in the reengineered code that were not found in the original legacy code.

Results. The study took about fifty hours for the author of this thesis to reengineer the source code to use FOREL. The process involved locating fragments of code that used MSXML parser and replace it with FOREL ROPE statements. The size of the code was reduced by 30% simply by replacing repetitive use of `MSXML` API with concise ROPE expressions and statements. More complex systems would probably require more time for reengineering with unknown reduction of source code in size, if any.

Type graph generation took a little over eleven seconds for a schema that contains `1,653` types and elements. FOREL type checking algorithm took `7.8` and `2.2` seconds for `papiNet` and `Metalex` applications respectively. For the papiNet application `49` bugs were detected, `31` of which were confirmed through manual code inspection, and for the `Metalex` application five bugs were detected two of which were confirmed later.

3.7 Summary

The contributions of this chapter are the following:

- a type system that enables the verification of foreign objects via encapsulated strategies in FOREL;
- an implementation in C++ that uses static analysis and algorithms from adaptive programming;
- a formalization of our FOREL type system using a formal model of ClassicJava and a proof of its soundness;
- a novel algorithm for inferring types of foreign objects for FOREL encapsulation strategies; and
- as a result of an empirical evaluation of FOREL we find bugs in a real-world commercial program.

Our experience suggests that FOREL is practical, and its type checking algorithm is efficient.

Chapter 4

Finding Errors In Interoperating Components

While ROOF and FOREL offer new approaches for developing interoperating components, many components are still written using low-level platform API calls. It is not likely that millions of lines of legacy software would be replaced in the near future using ROOF and FOREL (although we hope that it will!). Currently, there is no approach that can detect a situation at compile time when one component modifies XML data so that it becomes incompatible for use by other components, delaying discovery of errors to runtime.

Our solution is a ***V**erifier for **I**nteroperating c**O**mponents for finding **L**ogic f**A**ults (Viola)* that finds errors in components exchanging XML data and helps test engineers to validate reported errors. Viola creates models of the source code of components and computes approximate specifications of the data (i.e., schemas) that these components exchange. The input to Viola is the component's source code, schemas for the XML data used by these components, and *Finite State Automata (FSAs)* that model abstract opera-

tions on data with low-level platform API calls. Abstract operations include navigating to data elements, reading and writing them, adding and deleting data elements, and loading and saving XML data. These FSAs are created by expert programmers who understand how to use platform API calls to access and manipulate XML data.

Viola uses control and data flow analyses along with the provided FSAs to extract abstract operations from the component source code. Next, these operations are symbolically executed to compute approximate schemas of the data that would be output by these components. That is, given the schema of the input data, Viola reengineers the approximate schema of the data that would be output by some component from its source code.

The reengineered and expected schemas are compared to determine if they match each other. If a mismatch between them is found, it means that some component modifies the data incorrectly so that runtime exceptions may be thrown by other components using this incorrect data. To confirm this, Viola analyzes paths to data elements accessed and modified by these components to determine whether the schema mismatch results in actual errors. Sequences of operations leading to some potential errors are reported to help test engineers validate and reproduce errors.

Viola is a helpful bug finding tool whose static analysis mechanism reports some potential errors for a system of interoperating components. We tested Viola on open source and commercial systems, and detected a number of known and unknown errors in these applications with good precision thus showing the potential of this approach.

4.1 The Problem Statement

Our goal is create a tool for finding errors in interoperating components that exchange XML data. This tool should report some potential errors when evidence of violating some prop-

erties is found. A sound approach ensures the absence of errors in components if it reports that no errors exist, and a complete approach reports no errors for correct components. Our approach is neither sound nor complete, that is, it may miss some errors or report errors that do not exist in components (i.e., false positives). However, the precision of error reporting should be sufficient to find actual errors in interoperating components in practical settings.

There are different reasons why programmers make mistakes when they write interoperating components. Based on our participation in large-scale projects, we observe that programmers often make wrong assumptions about schemas. Given that many industrial schemas contain thousands of elements and types, it is easy to make mistakes about names of elements and their locations in schemas. The other source of errors lies in the complexity of platform API calls that programmers use to access and manipulate XML data. XML parsers export dozens of different API calls, and mastering them requires a steep learning curve.

Often, programmers lack the knowledge of the impact caused by changing the code of some component on other components that interoperate using XML data. This lack of knowledge is an effect of the Curtis' law that states that application and domain knowledge is thinly spread and only one or two team members may possess the full knowledge of a software system [37]. The effect of this law combined with the difficulty of comprehending large-scale XML schemas and high complexity of platform API calls result in components producing XML data that is incompatible for use by other components.

The other source of errors is the disparity in evolving XML schemas and components. Database administrators usually maintain schemas, and programmers maintain components that interoperate using XML data that should be instances of these schemas. If a database administrator modifies some schemas and does not inform all programmers

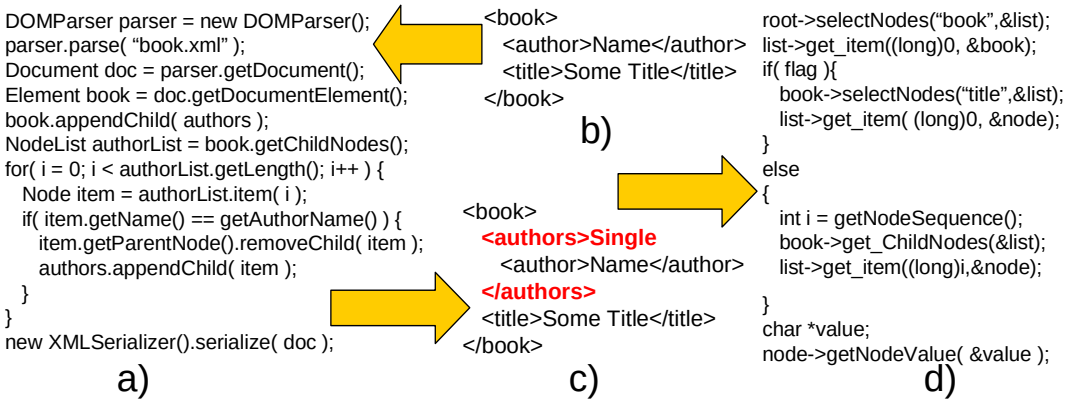


Figure 4.1: Java (a) and C++ (d) components that interoperate using XML data (b) and (c).

whose components are affected by this change, then some components will keep modifying XML data according to the obsolete schemas.

Our goal is to design a tool that ensures that certain properties hold in components interacting using XML data. These properties are the main and secondary safety properties. Recall that the *main safety property (MSP)* is defined as ensuring $\mathtt{D}_2$ conforms to $\mathtt{S}$. The *secondary safety property (SSP)* is defined as the same data elements in $\mathtt{S}$ should not be accessed by one and modified by some other interoperating components provided that specifications are not used at runtime to validate XML data.

Since the MSP and SSP ensure stronger guarantees that no runtime will be thrown, using XML parsers to validate data against schemas is irrelevant to our problem. The problem is to find and report some situations at compile time in which interoperating components violate both safety properties. Currently, no tool checks interoperating components for violating these properties, even when components are located within the same application. Viola should output descriptions of execution scenarios that lead to potential errors, and test engineers should be able to follow these scenarios to validate the reported errors.

4.2 Errors

We classify errors that Viola catches in interoperating components into the following general categories:

- *Path-Path (P2)* errors occur when a component accesses elements that may be deleted by some other components.

  P2 errors occur in components that access data elements that are deleted by some other components (P2-1) and by components that read or write wrong elements (P2-2). P2-1 errors are execution-order-dependent and therefore are difficult to find using testing or manual code inspection. If some component deletes data elements after some other component accesses these elements, then the execution proceeds correctly. However, if the order of the execution is reversed, then an exception will be thrown by a component that accesses a previously deleted element.

  P2-2 errors occur when one component navigates to a wrong data element and reads its value by using sequence numbers of elements for navigating rather than their names. Consider a component that reads the value of the first element located under the root "`book`" in the XML data shown in Figure 4.1b[1]. The read element is "`author`" and the obtained value is "`Name`." However, if the component `J` modified this data as shown in Figure 4.1c, then the read element would be "`authors`" and the obtained value is "`Single`". Thus, if the component `J` inserts a data element into the path to some elements accessed by the component `C`, then the result of interference of these operations is that the component `C` accesses and reads values of different data elements from what was intended when it uses sequence numbers of

[1] An example is shown in Figure 1.1 is replicated here in Figure 4.1 for the convenience of the reader.

elements rather than their names.

- *Path-Schema (PS)* errors occur when components attempt to access, delete, or add elements that do not exist in the schemas for the data (PS-2), or when components violate bounds set by schemas on data elements as a result of executing operations on data (PS-1).

  PS-1 errors occurs when components violate constraint bounds set by schemas. Suppose that a schema defines the value of the `minOccurs` attribute for a data element to be equal to one, however, a component deletes all instances of this element. Some other component may execute code that was written based on the assumption that at least one instance of this data element should be present in the XML data. This situation may also lead to execution-order-dependent runtime errors.

- *API errors* that result from incorrect uses of API calls.

  Mastering APIs for accessing and manipulating data often requires programmers to spend long periods of time learning dependencies between APIs and objects that are created as results of their calls [66][77]. One of common mistakes is that programmers use incorrect APIs in the sequences of calls designed to perform operations on data. Given that the knowledge of how to use APIs correctly is encapsulated in the descriptions of sequences of API calls that expert programmers build for abstract operations, Viola can flag sequences of API calls that do not match any abstract operations as potentially erroneous at compile time. It may also be that the flagged sequence of API calls is correct, and no FSA was provided to Viola to validate this sequence. In this case experts will add an FSA to the Viola FSA database, and these sequences will be accepted from that moment on.

Sometimes programmers forget to make components save their changes to data (e.g., a `return` statement may be executed before the `Save` operation in some execution path). Technically, it is not an incorrect use of API, but rather omission of a crucial operation that makes changes to data persistent. The data remains consistent after operations are executed; however, changes made by the component that does not save the data will be lost. Viola reports these situations to programmers at compile time helping them to find and debug potential logic faults.

Below are examples of warnings that Viola issues to programmers after it analyzes interoperating components:

P2-1: At line `23` component `C` accesses element ⟨`book`, `author`⟩ that may be deleted by the component `J` at line `122`.

P2-2: At line `23` component `C` may read a wrong element located under path ⟨`book`⟩ because component `J` modifies elements under this path at line `122`.

PS-1: At line `23` component `C` may delete all instances of the element ⟨`book`, `author`⟩, however, at least one instance of this element is required by the schema `S`.

PS-2: At line `23` component `C` accesses element ⟨`book`, `royalties`⟩, however, this element is not defined by the schema `S`.

4.3 The Architecture of Viola

Viola's architecture and process are shown in Figure 4.2. The steps of the Viola process are presented with numbers in circles. The names of components and schemas are taken from the model shown in Figure 1.2. The input to the architecture is the `J`'s and `C`'s components

source code (1). The EDG C++ and Java front ends [13] parse the source code of the components and output *Abstract Syntax Trees (ASTs)*(2). The *Analysis Routines (ARs)* perform control and data flow analyses on the ASTs in order to determine sequences of API calls that can be replaced with abstract operations. ARs also input FSAs that model abstract operations on XML data (3), and check to see if sequences of API calls retrieved from the source code are accepted by these FSAs. If a sequence of API calls is not recognized, or some abnormalities in using these API calls are detected, then API errors are reported to programmers (4).

Running ARs result in abstract programs for C and J components (5). Abstract programs represent sequences of abstract operations on the XML data. The *Symbolic Executor (SE)* executes the abstract program for the component J on the schema S_1 of the XML data D_1 (6) and outputs the schema S′ (7) and *Symbolic Execution Trees (SETs)* (8). SETs are graphs characterizing the execution paths followed during the symbolic executions of a program. Nodes in these graphs correspond to executed statements, and edges correspond to transitions between statements.

Schema S′ is the approximate specification of data that would be output by the component J if it is executed on the input data D_1. This schema can be viewed as reengi-

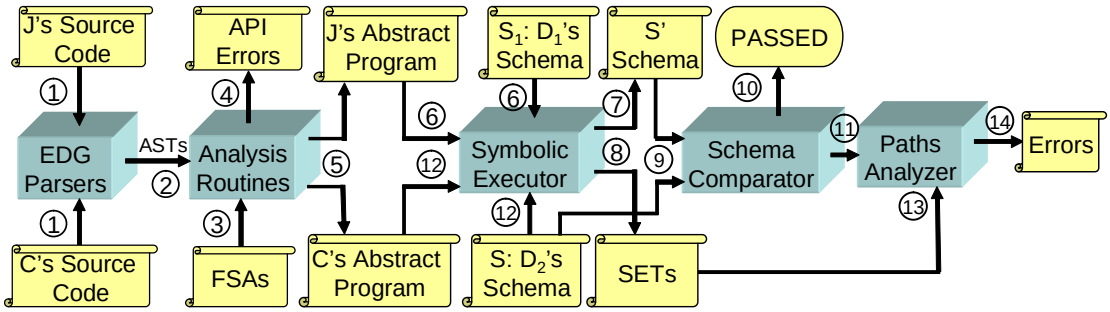


Figure 4.2: Viola's architecture and process.

neered from the component J when its abstract program is symbolically executed on the schema S_1 of the XML data D_1. This reengineered schema S′ represents the approximate view of the XML data held by a programmer who wrote the component J. Comparing the reengineered schema S′ with the schema S establishes if the MSP is violated, and consequently if the component J may perform some incorrect manipulation on the input data.

The *Schema Comparator (SC)* compares the reengineered schema S′ with the D_2's schema S (9), and reports success if the schemas are the same (10). If this step fails, then the MSP is violated. In the next step (11), Viola checks for violations of the SSP by analyzing if the component C accesses data elements that are modified by the component J.

To check for the violations of the SSP property, SE executes the abstract program of the component C on the schema S of the data D_2 (12). The purpose of this step is to obtain information about data elements that the component C accesses in the data D_2 provided that it is an instance of the schema S. Then SE executes the abstract program of the component C on the schema S′ which is reengineered from the component J during the previous steps. The purpose of this step is to obtain information about data elements that the component C would access in the data D_2 that is not an instance of the schema S_1, but rather of the reengineered schema S′. This information is stored in the SET resulting from this execution, and this SET is added to the set of SETs (8).

The *Paths Analyzer (PA)* analyzes the paths computed by components to accessed and modified data elements (13), and reports the discovered errors to programmers (14). By comparing the paths to elements that the component C may access in the data D_2 that is an instance of the schema S versus the paths to elements in the data that is an instance of the schema S′, PA reports different situations that may lead to P2 errors.

Using Viola is described in the procedure StartViola that is shown in Algo-

rithm 3. The input to this procedure is the set of components C. For each pair of distinct components c_i and c_j from the set C, it is determined whether these components interact via the set of the XML data {D}. This set is obtained by using the function GetData: C→{D} that maps the component C to the set of data {D} that it uses. This function is applied to the components c_i and c_j to obtain the sets of data $\{D\}_i$ and $\{D\}_j$ respectively, and the intersections of these sets of data is computed to determine the common data used by both components c_i and c_j. Then for each element of the computed set of common data, the procedure CatchErrors is called. This procedure executes the steps of 1-14 the Viola architecture and issues a list of potential errors.

Algorithm 3 StartViola procedure.

```
StartViola( C )
for all c_i ∈ C do
  for all c_j ∈ C ∧ c_i ≠ c_j do
    GetData(c_i) ↦ {D}_i
    GetData(c_j) ↦ {D}_j
    D = {D}_i ∩ {D}_j
    if D ≠ ∅ then
      for all d ∈ D do
        CatchErrors(c_i, c_j, d)
      end for
    end if
  end for
end for
```

4.4 The Models

Viola builds a model of a program by abstracting its operations on data, and then symbolically executes these operations on schemas. We describe program abstractions and schemas, and specify the formal framework for building these abstractions.

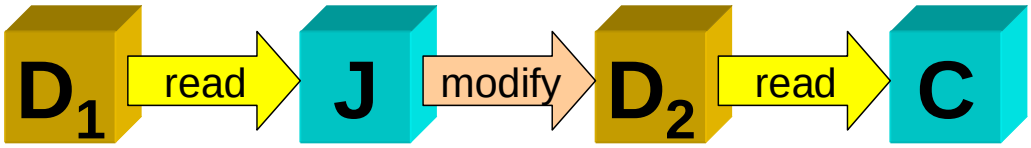


Figure 4.3: A model of component interoperability.

4.4.1 Program Models

A basic programming model for interoperating components is shown in Figure 4.3[2]. Our goal is to verify at compile time the MSP (the data D_2 should conform to the schema `S`), and the SSP (the same data elements in `S` should not be accessed by one and modified by some other interoperating components). The problem is to determine at compile time how operations that access and modify data violate these properties. These operations are navigating to data elements, reading and writing data, adding and deleting data elements, and loading and saving data, designated as `Navigate`, `Read`, `Write`, `Add`, `Delete`, `Load`, and `Save` respectively.

These abstract operations are implemented in components using low-level platform API calls. In general, API calls are complex and the client code that uses API calls is difficult and tedious to write. It is rare that a one-to-one correspondence exists between abstract operations and API calls. Programmers have to execute sequences of API calls in order to accomplish each of these abstract operations [66][77]. For example, the `getDocumentElement` API call returns a node in the internal representation of XML by *Document Object Model (DOM)* parsers. This internal node is used when calling other API calls as shown in Figure 1.1a.

Program abstractions represent source code of interoperating components using abstract operations. We observe that a majority of statements and operations in applications

[2] An example is shown in Figure 1.2 is replicated here in Figure 4.3 for the convenience of the reader.

are irrelevant to accessing and modifying XML data. Viola abstracts away specifics of API calls and program variables that do not affect XML data, and transforms applications into sequences of abstract operations. This technique decreases the number of states of components by focusing on a subset of their variables related to the given specification and eliminating the remaining variables.

Data abstractions represent actual data with smaller generalized specifications. Data abstractions can be achieved with XML schemas [14]. We consider XML schemas in this thesis because XML is the lingua franca of data exchange, and because XML schemas are also used to model non-XML data (e.g., relational databases [61] and PDF files [1]).

Program abstractions are obtained from source code by analyzing it and mapping sequences of platform API calls to the abstract operations. Examples of program abstractions for the Java and C++ components from Figure 1.1a and Figure 1.1d are shown in Figure 4.4a and Figure 4.4b respectively. In general, names of data elements (e.g., "`title`") are not constants; they are expressions whose values are computed at runtime. In program abstractions, these names are replaced with symbolic variables (e.g., `root`, `child`, `e1`, and `e2`) as it is shown in Figure 4.4. With program abstractions we lose precision, however, as the number of program states is significantly reduced.

Branching statements include boolean predicates specifying the condition under which certain execution paths will be taken. In general, we lack the knowledge to determine the exact predicate. For example, the condition `i<authorList.getLength()` of the `for` loop for the Java component shown in Figure 1.1a means that the iterator variable `i` is incremented until its value reaches the number of items in the `authorList` object. However, this condition could be determined only at runtime. In Viola a general approach is taken to abstract away predicate conditions of branching statements and loops.

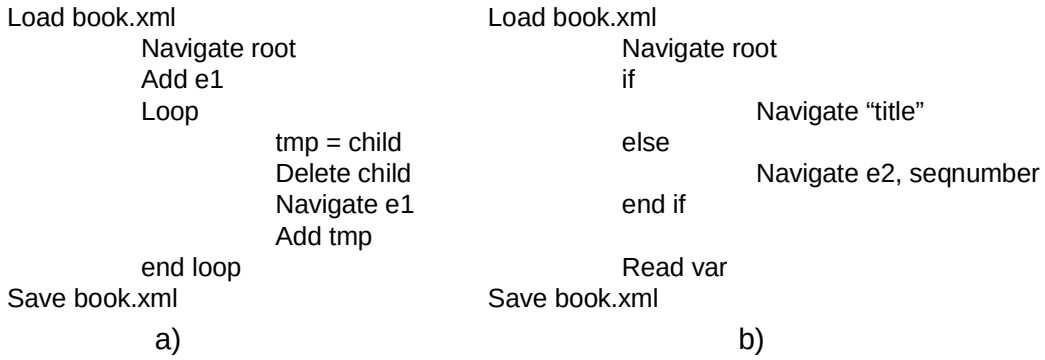


Figure 4.4: Program abstractions for Java a) and C++ b) components shown in Figure 1.1a and Figure 1.1d respectively.

The keywords `loop` and `if-then` are used in abstract programs to indicate branching control statements in the corresponding components. Recursive functions are abstracted as called inside loops with the recursive calls removed.

4.4.2 Schemas

XML schemas are recorded in the XML format [14] and each schema has the root specified with the `<schema>` element. Data elements are specified with `<element>` and `<attribute>` tags. Each data element is defined by its name and its type. Elements can be either of simple or complex types. Complex element types support nested elements while simple types are attributes and elements of basic types (e.g., `integer`, `string`, or `float`).

Elements may have two kinds of constraints. First, the values of elements may be constrained through enumerating or specifying low and upper bounds for numerical types. The second constraint specifies bounds on the number of times that a specific element may occur as a child of some element. These bounds are specified with the `minOccurs` and

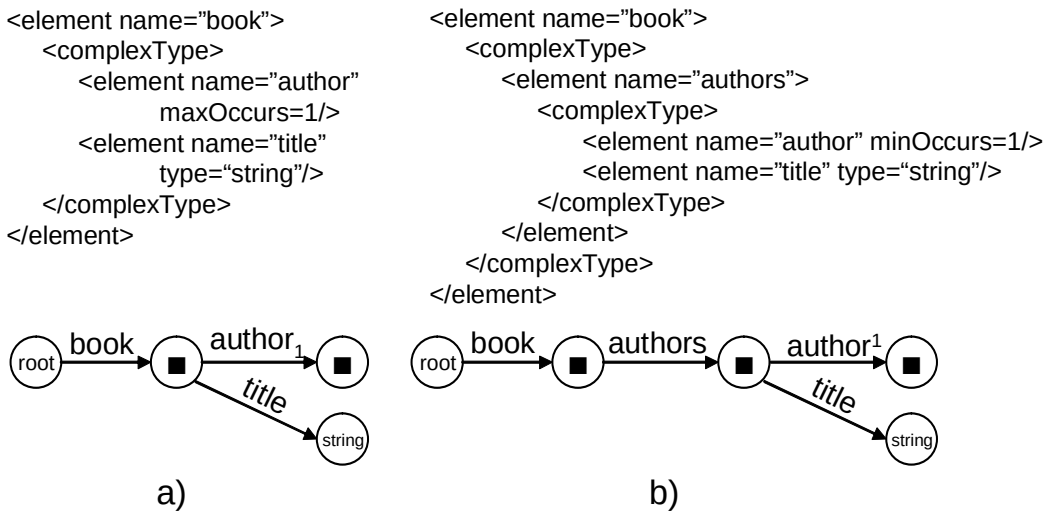


Figure 4.5: Examples of graphs for two XML schemas.

`maxOccurs` attributes of the ⟨`element`⟩ tag.

Elements can be grouped in a sequence if they are children of the same parent element. Attributes of the same element can also be grouped in a sequence. Each element or attribute in a sequence is assigned a unique positive integer sequence number. This number can be used to access elements or attributes instead of using their names.

We represent schemas using graphs, and we use this formalism for comparing different schemas in order to detect discrepancies that lead to runtime errors. Examples of graphs for two different schemas are shown in Figure 4.5. These schemas describe instances of XML data shown in our motivating example in Figure 1.1b and Figure 1.1c.

Let `T` be finite sets of type names and `F` of element and attribute names (labels), and distinct symbols $\diamond \in$ `F` and ■ $\in$ `T`. Schemas graphs are directed graphs `G = (V, E, L)` such that

- `V`$\subseteq$`T`, the nodes are type names or ■ if the type name of data is not known;
- `L`$\subseteq$`F`, edges are labeled by element or attribute names or $\diamond$ if the name is not known;

- $\texttt{E} \subseteq \texttt{L} \times \texttt{V} \times \texttt{V}$, edges are cross-products of labels and nodes. If $\langle l, v_k, v_m \rangle \in \texttt{E}$, we write $v_k \xrightarrow{l} v_m$. Nodes v_m are called children of the node v_k. If an element has no children, then its corresponding node in a schema graph has an empty collection of children nodes;

- Bounds for elements are specified with subscripts and superscripts to labels designating these elements. Subscripts are used to specify bounds defined by the `minOccurs` attribute, and superscripts designate the bounds specified by the `maxOccurs` attribute;

- Each graph has the special node labeled $\texttt{root} \in \texttt{V}$, where `root` represents a collection of the root elements. An empty schema has a single root node and no edges;

- The XML tag `<complexType>` specifies that an element is a complex type, and it is not represented in the graph.

A path in a schema graph `G = (V, E, L)` is defined as a sequence of labels $\texttt{p}_G = \langle l_1, l_2, \ldots, l_n \rangle$, where $v_k \xrightarrow{l_u} v_m$ for $v_k, v_m \in \texttt{V}$ and $1 \leq u \leq n$. The symbol $\diamond$ may be used instead of a label in a path if an element is navigated by its sequence number. For example, the symbol $\diamond$ in the path $\langle \texttt{book}, \diamond \rangle$ for the schema graph shown in Figure 4.5a stands for any child element of the element `book`, and this path may be expanded into two paths: $\langle \texttt{book}, \texttt{author} \rangle$ and $\langle \texttt{book}, \texttt{title} \rangle$.

Path $\texttt{p}_i$ is a subpath of some other path, $\texttt{p}_j$, $\texttt{p}_i \subseteq \texttt{p}_j$ if and only if all labels of the path $\texttt{p}_i$ are also labels of the path $\texttt{p}_j$, and the order in which they appear and how they relate to each other in $\texttt{p}_j$ is the same as they appear and relate to each other in the path $\texttt{p}_i$. Function $\texttt{type} : v \rightarrow \texttt{s}$ returns the type $\texttt{s} \in \texttt{T}$ of the node $v \in \texttt{V}$. Function $\texttt{max} : \texttt{label}_l^u \rightarrow u$ returns the upper bound u, or ∞ if the upper bound is not specified, and function $\texttt{min} : \texttt{label}_l^u \rightarrow l$

returns the lower bound `l`, or zero if the lower bound is not specified.

4.4.3 The Formal Framework

In order to build program abstractions we need to locate sequences of API calls that correspond to abstract operations. If all sequences of platform API calls for the abstract operations were known in advance, then these sequences can be searched for and replaced with abstract operations. Unfortunately, the multiplicity of data hosting platforms and API calls for accessing and manipulating data as well as their constant evolution makes it difficult to hardcode all sequences of API calls for abstract operations once and for all. We use a formal framework to define these sequences and extract them from source code in a uniform way.

Let Σ be the set of API calls that access and manipulate data, and let Γ be the set of abstract operations, $\Gamma = \{$`Navigate`, `Read`, `Write`, `Add`, `Delete`, `Load`, `Save`$\}$. Let $\alpha \subseteq \Sigma^*$ be sequences of API calls that access and manipulate data. Partial function $\varphi : \alpha \rightarrow \gamma$ maps a sequence of API calls α to an abstract operation $\gamma \in \Gamma$.

We assume that α, the set of sequences of API calls, is a regular language. Since for each regular language there exists an FSA that accepts this language, FSAs are provided for each γ, such that $\varphi^{-1}(\gamma) = \alpha$. If an FSA for an abstract operation γ accepts sequences of API calls α, then these API calls can be replaced with this abstract operation in the abstract program.

Different API platforms offer API calls that are organized in different sequences that are mapped to abstract operations. It is impractical to constantly evolve Viola to recognize new sequences of API calls. By describing sequences of API calls uniformly, Viola can perform its analysis in a platform-independent way.

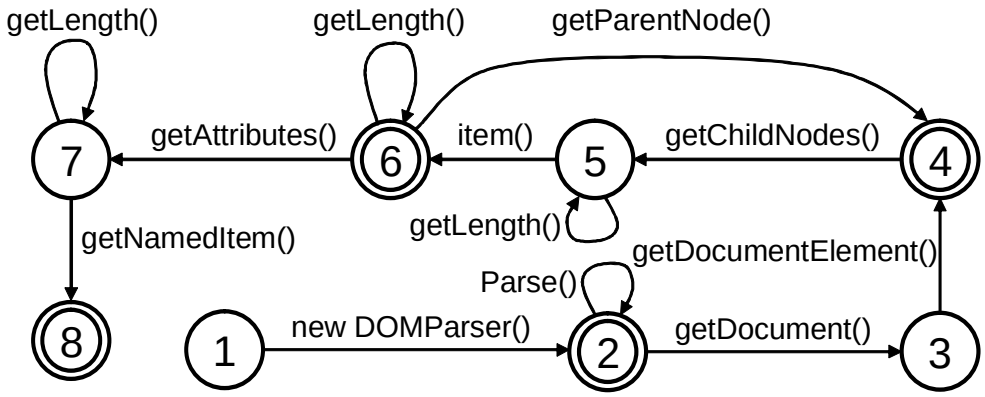


Figure 4.6: Example of an FSM for accepting XML DOM API calls for the `Navigate` abstract operation.

A trace $\tau \subseteq \Sigma^*$ is a sequence of operations that can be executed by a path in the program. Let τ^* be all possible traces that result from all possible execution paths of the program. If α is a subtrace of τ^* and $\varphi(\alpha) = \gamma$, then a subtrace of τ^* can be replaced with the corresponding abstract operation γ. In general, τ^* is not a regular language because it contains traces of function calls. When a function is called, a stack is used to store local variables and the return address. Languages that use stacks are context-free rather than regular. If τ^* is modeled as a context-free language, then there exists a *pushdown automaton (PDA)* that accepts this language. A PDA configuration is described by its current state and all the symbols that the PDA contains in the given state. It was shown that reachable configurations of a PDA form a regular language, and therefore can be represented by some FSA [33][41].

An example of an FSA for the `Load` and `Navigate` abstract operations for Xerces XML parser API calls is shown in Figure 4.6. Circles designate states with state numbers inside the circles, and transitions are labeled with API calls. Double concentric circles indicate final states, and circles with no edges incident on them are the start states. This

FSA contains four joined FSAs, one for modeling the `Load` at the final state `1`, and three for `Navigate` abstract operations. Experts who understand how to use API calls to implement abstract operations construct these FSAs. Once built, these FSAs can be used for building abstract programs from the source code of components.

In our framework we perform limited analysis of data flow. Specifically, we are only interested in producing separate traces of operations invoked on different objects that represent different data. For example, given two `DOMParser` objects that designate different data, we want to obtain separate traces of methods invoked on these objects. No dataflow analysis is performed on the parameters to these methods. This decision results in better performance of Viola, but the side-effect is more false-positives produced by the compiler. However, we show in our experimental evaluation, the number of false positives is acceptable in practice.

4.5 Building Program Abstractions

Here we explain how abstract programs are obtained using components source code and FSAs that model abstract operations. We give a grammar for abstract programs and show an example of obtaining an abstract program from a fragment of Java code.

4.5.1 Extracting Abstract Programs

Program abstractions are obtained from Java and C++ programs using FSAs that map low-level API calls to abstract operations. EDG parser front ends for C++ and Java build *Control Flow Graph (CFG)* from the source code of components. CFGs are graphs whose nodes represent *basic blocks (BBs)* with a statement containing a relevant API call from the set α, and edges between BBs that represent the flow between program statements. Compo-

nents are partitioned into BBs in the following manner. Statements are read and put into the same BB until either a relevant API call or a branching statement (e.g., `if-then`, `switch-case`, `do-while`) is encountered. Then, this BB is added to the CFG. If the program consists of multiple source files, then the CFGs produced for each of these source files are merged into a single CFG.

Program abstractions are computed for each path in the CFG, where a path is a set of nodes from the CFG connected by edges. For a selected path in the CFG stacks for API calls are created. For each BB containing an API call in the given path, this API call is extracted and put on the stack. Every time an API call is added, the sequence of API calls on the stack is checked to see if it is accepted by any FSA for abstract operations. If such an FSA is found, then this sequence is modeled by the corresponding abstract operation for this FSA, and this operation is put into the abstract program.

Each FSA has a uniquely labeled edge incident on the start node. The label of this edge is an API call that is not encountered anywhere else in the FSAs except for the edges incident on the start nodes. When such an API call is encountered in a CFG, a new stack is created. Several stacks can exist at any given time, and API calls are put on these stacks until they satisfy some FSAs and can be replaced with the corresponding abstract operations. For example, if there are three FSAs are stacked, then the next API call may be added to all three stacks. If no stack can be replaced with an abstract operation, then an API type of error is reported to programmers.

4.5.2 Limitations

In general, it is an undecidable problem to determine all API calls relevant for a given stack from an arbitrary program. If a function pointer is used to reference an API call, then Viola

is unable to resolve it without the guidance from users. To understand the limitations of Viola consider a code fragment where a reference to an object that denotes a data element is put, among other references, into a hash table object, which is passed as a parameter to some function. Inside this function, references to objects are retrieved from the hash table. In a general case it is impossible to determine what exact objects in the calling function the references denote in the hash table inside the called function. To resolve this situation, Viola requires guidance from programmers to resolve object references. This guidance takes a form of a prompt with a list of objects from the calling context that match the object in question from the called context.

4.5.3 Example of Extracting Abstract Program

We demonstrate how to extract an abstract program from a fragment of Java code using the FSA shown in Figure 4.6. A fragment of Java code is shown in Figure 4.7 to the left of the block arrow, and the extracted abstract program is shown to the right of the block arrow.

When extracted from the Java source code, API calls are put on the stack. Each time the stack configuration is updated, it is checked to see if API calls from the current stack configurations are accepted by some FSA that models some abstract operation. When API

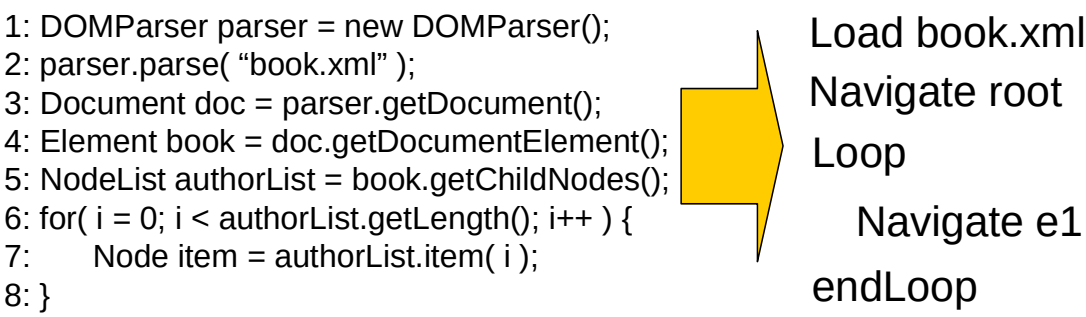


Figure 4.7: Example of the abstract program extracted from a fragment of Java code.

calls from a stack configuration are accepted by some FSA, this configuration is reduced to the abstract operation which this FSA models. If several stacks exist, then only these stack configurations are updated with API calls if these calls are accepted by corresponding FSAs.

When processing the fragment of Java code shown in Figure 4.7 to the left of the block arrow, the API call `new DOMParser()` is retrieved from the line `1`. The FSA, shown in Figure 4.6, accepts this call switching to the state `2` from the initial state `1`. The next API call `parse()` is retrieved from the line `2`, and FSA accepts this call in the state `2`. Incidentally, this sequence of API calls corresponds to the abstract operation `Load`, and it is modeled by the FSA starting at the state `1` and finishing at the state `2`. The configuration of the stack is reduced to the abstract operation `Load book.xml` which is put into the abstract program shown in Figure 4.7 to the right of the block arrow.

Next two API calls `getDocument()` and `getDocumentElement()` are retrieved from lines `3` and `4` respectively. These calls are put on the stack and accepted by the FSA, switching it to states `3` and `4`. This FSA models the abstract operation `Navigate` whose parameter is the symbolic variable `root` denoting that the program navigates to the root element of the XML document. Thus, the stack configuration is reduced to the abstract operation `Navigate root` which is put into the abstract program.

After retrieving the API call `getChildNodes()` from the line 5 and switching the FSA to the state `5`, the keyword `for` is encountered, denoting the loop control structure. In general, it is not possible to determine how many times this loop will execute, if it executes at all. This lack of knowledge is reflected by abstracting away conditional predicates of loops, leaving only information about the loop itself in abstract programs. Thus, the `Loop` and `endLoop` keywords are put into the abstract program. Continuing to re-

trieve API calls from lines 6 and 7, `getLength()` and `item()` are put on the stack and accepted by the FSA, switching it to the state 6. Then, the stack configuration is reduced to the abstract operation `Navigate e1` which is put into the abstract program inside the loop.

The lack of knowledge about the navigated data element is reflected in the symbolic variable `e1`, the parameter to the abstract operation `Navigate`. The values of this variable are the names of the navigated data elements. In general, it is an undecidable problem to retrieve names of data elements using static program analysis. However, when abstract operations are executed on schemas, symbolic variables take specific values as parameters to these operations. We show this process in Section 4.6.

4.5.4 The Grammar of Abstract Programs

A grammar for abstract programs is shown in Figure 4.8. An abstract program is a sequence of operations and command included within `Load` and `Save` operations. The metavariable `aoper` ranges over abstract operations; `id` ranges over symbolic identifiers (variables); `const` ranges over string and integer constants; `command` ranges over the `state` and `jump` commands as well as assignment expressions, and `body` ranges over program definitions. An abstract program in Viola, `Program`, is a triple `(load, body, save)` of `Load` and `Save` abstract operations and the program body. Assignment expression is also included, allowing us to create global identifiers and assign values to them.

4.5.5 Description of Abstract Operations

The summary of abstract operations is given in Table 4.1. The operations `Load` and `Save` take the identifiers of data as their arguments. These operations mark the beginning and

```
Program  ::=  load body save
load     ::=  Load var
save     ::=  Save var
body     ::=  Loop body endLoop |
              If body endIf |
              If body else body endIf |
              aoper | command
aoper    ::=  Navigate var |
              Navigate var, seqnumber |
              Navigate var, attribute |
              Navigate var, parent |
              Read var | Write var |
              Add var | Delete var
command  ::=  state var | jump var | id = var
var      ::=  id | const
id       ::=  an identifier
const    ::=  n | s
n        ::=  numerical value
s        ::=  string value
```

Figure 4.8: The grammar for abstract programs.

the end of abstract programs. The mandatory parameter of the operation `Navigate` is the name of a data element or attribute or a symbolic variable if the name is not known, or the order sequence number of the data element in the collection of elements that are children of the currently referenced element. The `seqnumber` optional parameter specifies that the sequence number of the element is used to access it rather than its name. The optional parameter `attribute` specifies whether an attribute or an element is navigated to, and the optional parameter `parent` gives the direction of the navigation to parent rather than to children elements.

The operation `Add` takes names of data elements and adds them under the currently navigated elements. Finally, the `Delete` operation takes names of data elements as its parameter and deletes them from the collection of children of the currently referenced ele-

OPERATION	DESCRIPTION
`Load identifier`	Start interacting with the XML data referenced by its identifier
`Save identifier`	Update the XML data referenced by its identifier
`Navigate element` [,`seqnumber`]	Navigate to the element whose name is specified by the symbolic variable, the optional flag `seqnumber` specifies that the element is denoted by its sequence number rather than its name,
[,`attribute`]	the optional flag `attribute` specifies that the element is an attribute,
[,`parent`]	and the optional flag `parent` specifies that it is navigated to a parent element
`Add element`	Inserts elements specified by the symbolic variable `element` under the currently navigated elements
`Delete element`	Deletes elements specified by the symbolic variable `element` that are children elements of the currently navigated element

Table 4.1: Abstract operations and their descriptions.

ments.

4.6 Symbolic Execution

Symbolic execution is a path-oriented evaluation method that describes data dependencies for a path [57][58][35]. Program variables are represented using symbolic expressions that serve as abstractions for concrete instances of data that these variables may hold. The state of a symbolically executed program includes values of symbolic variables. When a program is executed symbolically its state is changed by evaluating its statements in the sequential order.

4.6.1 Background

Historically, symbolic evaluation is used for analyzing and testing programs that perform numerical computations. We illustrate it on a simple example. Consider two consecutive statements `x=2*y` and `y=y+x` in a program. Initially, variables `x` and `y` are assigned symbolic values `X` and `Y` respectively. After symbolically executing the first statement, `x` has the value `2*Y`, and after executing the second statement the value of `y` is `Y+2*Y`. When symbolically executing numerical programs, variables obtain symbolic values of polynomial expressions.

Recall that symbolic execution trees (SETs) are graphs characterizing the execution paths followed during the symbolic executions of a program. Nodes in these graphs correspond to executed statements, and edges correspond to transitions between statements. Each node in the SET describes the current state of execution that includes values of symbolic variables and the statement counter. Nodes for branching statements (e.g., `if` or `while` statements) have two edges that connect to nodes with different condition predicates.

4.6.2 Symbolic Variables

In Viola, symbolic variables contain paths to data elements. In many cases, names of data elements are not specified as constants in programs, but rather computed at runtime. These elements are assigned unique symbolic variables in abstract programs.

Each abstract program contains four symbolic state variables: `root`, `current`, `children`, and `rw`. The variable `root` contains the names of the root elements of the data. A set of tuples $\langle$`current`, `children`$\rangle$ describes elements referenced by a component in a given state and their children elements. This information will be used in the path analysis routines of Viola to get extended diagnostic information about found bugs.

In every tuple the variable `current` contains a path to the element that can be referenced in the given state, and the variable `child` contains paths to the elements that are children of the element referenced by the variable `current`. For example, for the data shown in Figure 1.1b in a state in which the root element is navigated, the set of values for the variables are `root` = {`book`}, `current` = {⟨`book`⟩}, and `children` = {⟨`book`, `author`⟩, ⟨`book`, `title`⟩}. Finally, the `rw` variable keeps the list of elements and attributes whose values are read or written in a given state.

The state of an abstract program is the union of values of the state variables. Abstract operations modify the state of a program by changing values of the state variables. Bookmarking commands "`state <name>`" and "`jump <stateName>`" mark certain states in abstract programs in order to enable execution to return to them from any point when executing these programs. The command "`state <name>`" represents an intermediate object created by platform API calls in the component. For example, operation `doc.getDocumentElement()` creates the transient data element `book` of type `Element` in the Java component shown in Figure 4.1a. This object represents the root of the data, and it is used in other API calls that navigate down to data elements. We assign some unique name to the state in which this intermediate object is created.

The command "`jump <stateName>`" directs execution of the abstract program to abandon its current state, and this command swaps in the context associated with some named state and continues to execute the program in the switched state.

Three global symbolic path variables are associated with each SET. The *access path variable* Θ keeps paths to navigated elements and attributes, the *delete path variable* Δ keeps paths to deleted elements and attributes, and the *add path variable* Ω keeps paths to added elements and attributes. Each path is associated with a node in the SET which

represents executing some operation from an abstract program. For each component C_i and data D_j there is a triple $\langle C_i, D_j, \langle \Theta, \Delta, \Omega \rangle \rangle$ that maps this component and data to access, delete, and add paths of the data elements accessed and modified by this component. This information is used to verify if the SSP holds.

4.6.3 Semantics of Abstract Operations

Abstract programs are executed symbolically on schemas producing SETs. Recall that nodes in SETs contain the values of symbolic variables and (modified) schemas. Each abstract operation creates a new node in the SET and updates the content of the symbolic state variables. We define the operational semantics of abstract operations in terms of changes made to the state variables and schemas after these operations are executed.

In general it is an undecidable problem to determine automatically how many times to execute symbolically abstract operations located within the body of a loop. We decided to make it a configurable parameter in Viola based on the *small scope hypothesis* [53] stating that in practice, many bugs can be detected in small scopes. By default, the number of times Viola executes the bodies of loops is one.

The operation `Load` marks the beginning of abstract programs and instructs Viola to create a new SET, initialize the state variables, and read in the schema. Viola processes abstract operations until the `Save` operation is encountered, which marks the end of the abstract program.

The `Navigate` operation, just like the `Load` and `Save` operations does not change the schema, but it modifies the content of the state variables to reflect navigated elements and attributes and their child elements. Other operations that do not modify schemas are the operations `Read` and `Write`. When they are executed, new nodes are created in the SET,

and the value of the `rw` state variable is updated with paths to data elements whose values are read or updated.

Operations `Add` and `Delete` add elements to and delete elements from schemas and update values of the state variables to reflect the changes. Recall that elements in schemas have bounds that are specified with attributes `minOccurs` and `maxOccurs`. If the `Add` operation is executed unconditionally (i.e., outside a loop of a branching statement), then it gives us assurances that there should be at least one instance of the added element in the data. Correspondingly, the value of the `minOccurs` attribute is set to one if it was previously set to zero, or left unchanged if it was greater or equal to one. Unconditional execution of the `Delete` operation deletes all instances of elements with given names. Correspondingly, the value of the `minOccurs` attribute is set to zero to reflect the possibility that all instances of a given element will be deleted.

Conditional executions of the `Add` and `Delete` operations occur when these operations are located within the bodies of conditional statements (e.g., `if-then`) or loops. In case of loops, we often lack the knowledge of the upper bound on the number of iterations through the loop, and we assume that it is infinite. A conservative guess of the lower bound on the number of iterations through the loop is zero, meaning that it is never executed. If the `Add` operation is executed within the body of a conditional statement or a loop, then the entry for the added element is inserted into the schema with the attribute `minOccurs` set to zero, and the attribute `maxOccurs` is not present reflecting the absence of the upper bound on the number of instances of this element. If the name of the added data element is not known at compile time, then all edges with the label $\diamond$ incident on the current element are added to the schema graph.

Executing the `Delete` operation within the body of a conditional statement means

that a single instance of some element may be deleted from data. In general, the name of a deleted element may not be known, and Viola makes a conservative guess about what elements may be affected by these operations. If the `Delete` operation is conditionally executed within the body of a loop on the set of `n` elements in the schema and it is not known what elements are deleted, then it is assumed that any subset of the set of these elements can be deleted from the data at runtime. Formally, this is expressed with the powerset operator $\mathcal{P}$ which transforms the set of `n` elements to 2^n subsets of these elements, which may remain after executing the `Delete` operation.

Executing the `Add` operation inserts an element whose name is specified by the parameter under the currently navigated elements. If the element being added already exists in the schema, then the schema is not changed. Otherwise, the schema is modified by adding an entry that describes the added data elements as children to the currently navigated elements. The `children` symbolic variable is updated with the path to the newly added element.

When the parameter to abstract operations `Navigate`, `Add`, and `Delete` is the actual name of a data element, the symbolic executor expects this element to exist as a child of the element referenced by the variable `current`. If the referenced data element does not exist, then Viola issues PS-2 error stating that a component attempts to access, delete, or add elements that do not exist in the schema. After issuing this error Viola continues to symbolically execute the abstract program.

4.6.4 Example of Symbolic Execution

We demonstrate an example of symbolic execution of abstract programs on schemas. We use names and terminology defined in the basic model shown in Figure 1.2 and from the

motivating example described Figure 1.1. Abstract programs for the components J and C are shown in Figure 4.10a and Figure 4.10b[3] respectively. These abstract programs are obtained from the source code for the Java and C++ components shown in Figure 1.1a and Figure 1.1d respectively. XML schema shown in Figure 4.5a serves both as the schema for input data D_1 and the expected schema S.

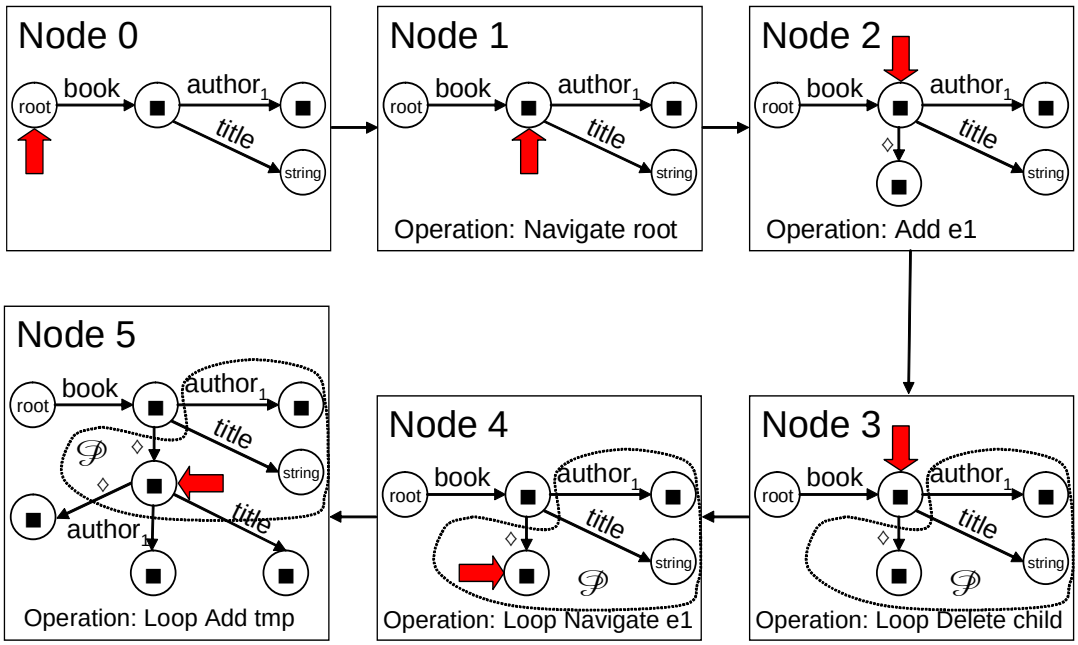
First, we execute the abstract program for the component J shown in Figure 4.10a symbolically on the XML schema for input data D_1 shown in Figure 4.5a. The result of this execution is the SET shown in Figure 4.9(a). The content of nodes in the SETs includes schemas and symbolic state variables. Due to lack of space, only schemas are shown in the nodes of the trees. Solid block arrows point to nodes denoted by the state variable current. Values for the tuple ⟨current, child⟩ for the SET nodes are shown in Tables 4.2. Values for the path variables are summarized in the Table 4.4.

Node	**Operation**	⟨**current, child**⟩
Node 1	Navigate root	⟨{⟨book⟩, {⟨book, author⟩, ⟨book, title⟩}}⟩
Node 2	Add e1	⟨{⟨book⟩, {⟨book, author⟩, ⟨book, title⟩,⟨book, ◇⟩}}⟩
Node 3	Loop Delete child	⟨{⟨book⟩, {⟨book, author⟩, ⟨book, title⟩,⟨book, ◇⟩}$_{\mathcal{P}}$}⟩
Node 4	Loop Navigate e1	⟨{⟨book, ◇⟩, ⟨⟩}⟩
Node 5	Loop Add tmp	⟨{⟨book, ◇⟩, {⟨book, ◇, author⟩, ⟨book, ◇, title⟩,⟨book, ◇, ◇⟩}$_{\mathcal{P}}$}⟩

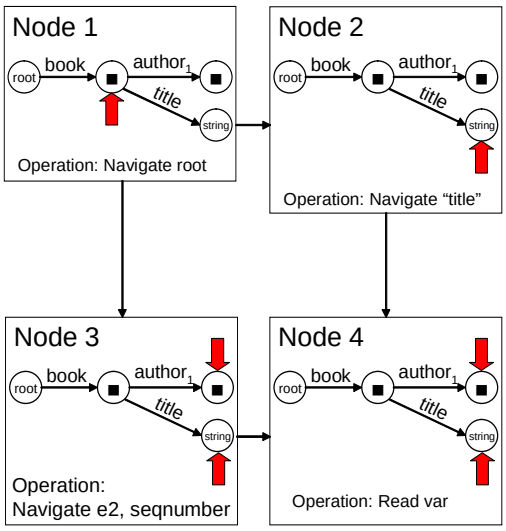
Table 4.2: Values of the state variables for the SET shown in Figure 4.9(a).

The Node 0 shows the initial schema for input data D_1. After executing the abstract operation Navigate root, the node Node 1 is created, and the schema remains unchanged because this operation does not modify the schema. Access path ⟨book⟩ is

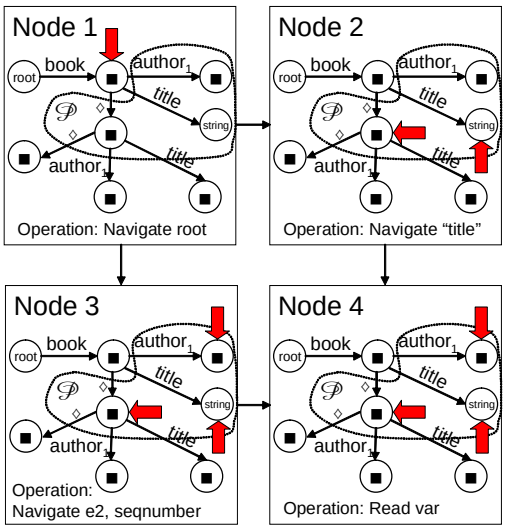
[3]Abstract programs shown in Figure 4.4 are replicated here in Figure 4.10 for the convenience of the reader.



(a) A SET for the execution of the abstract program for the component J shown in Figure 4.4a on the schema shown in Figure 4.5a.



(b) A SET for the execution of the abstract program shown in Figure 4.4b on the schema shown in Figure 4.5a.



(c) A SET for the execution of the abstract program shown in Figure 4.4b on the schema obtained as a result of the symbolic execution shown in Figure 4.9(a).

Figure 4.9: SETs for the symbolic executions of the abstract programs shown in Figure 4.4 on schemas.

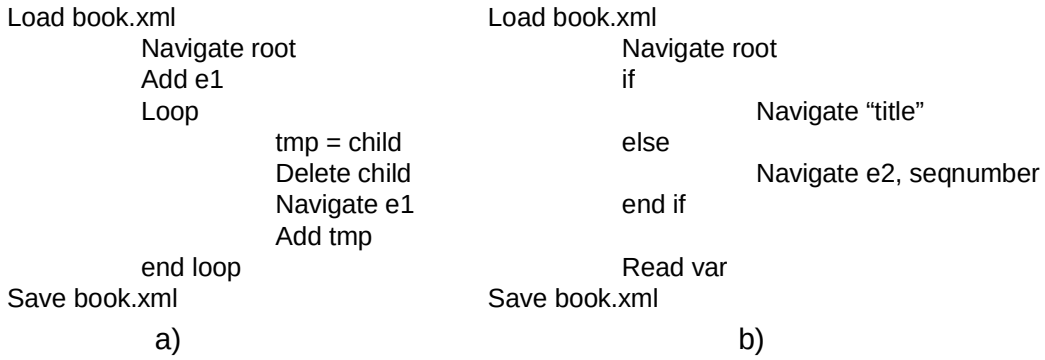


Figure 4.10: Program abstractions for Java a) and C++ b) components shown in Figure 1.1a and Figure 1.1d respectively.

added to the access path variable Θ as shown in the Table 4.4 in the column for the Java component paths.

The next operation `Add e1` adds an element denoted by the symbolic variable `e1` as a child to the currently navigated element `book`. Correspondingly, the path $\langle$`book`, $\diamond\rangle$ is added to the add path variable Ω. Since the value for the variable `e1` is not known at compile time, an edge labeled $\diamond$ is added to the schema graph. Node `Node 2` is created, and the value of the variable `current` remains unchanged, but the content of the variable `child` is updated with the path to the newly added element `e1`. We do not show the operation `tmp = child` that copies values of the variable `child` to the symbolic variable `tmp`.

Next, the operation `Delete child` is executed within a loop, and it deletes the children of the element `book`. Since the condition of the loop is not known in the abstract program, it is difficult to predict what children will be deleted, if any at all. Therefore it is assumed that any subset of children of the element `book` can be deleted. That is, after executing this operation in the loop an element of the powerset of the set of children elements of the element `book` will be left. The node `Node 3` is created and the $\mathcal{P}$ flag is set

Node	**Operation**	⟨**current, child**⟩
Node 1	Navigate root	⟨{⟨book⟩, {⟨book, author⟩, ⟨book, title⟩}}⟩
Node 2	Navigate “title”	⟨{⟨book, title⟩,⟨⟩}⟩
Node 3	Navigate e2, seqnumber	⟨{{⟨book, author⟩,⟨⟩}, {⟨book, title⟩,⟨⟩}, ⟨book, ⋄⟩, ⟨book, ⋄, ⋄⟩, {⟨book, ⋄, author⟩, ⟨book, ⋄, title⟩}}⟩

Table 4.3: Values of the state variables for the symbolic execution tree shown in Figure 4.9(c).

as indicated by applying $\mathcal{P}$ subscript to the variable `child`. That is, the children elements are tagged as potentially deleted. Correspondingly, the paths ⟨`book`, `author`⟩, ⟨`book`, `title`⟩, and ⟨`book`, ⋄⟩ are added to the delete path variable Δ as shown in Table 4.4.

Within the same loop the operation `Navigate e1` is executed as shown in the `Node 4`. Access path ⟨`book`, ⋄⟩ is added to the access path variable Θ. Finally, the operation `Add tmp` appends children of the element `book` to the currently navigated element `e1`, updating the add path variable Θ with the paths ⟨`book`, ⋄⟩, ⟨`book`, ⋄, `author`⟩, ⟨`book`, ⋄, `title`⟩, and ⟨`book`, ⋄, ⋄⟩. We designate the resulting schema shown in the `Node 5` as `S'`.

After obtaining the schema `S'`, Viola executes the abstract program for the component `C` shown in Figure 4.4b symbolically on the reengineered XML schema `S'`. The result of this execution is the SET shown in Figure 4.9(b), and the values for the tuple ⟨`current`, `child`⟩ for the SET nodes are shown in Table 4.3.

Finally, the abstract program for the component `C` shown in Figure 4.4b is executed

symbolically on the XML schemas for input data D_2 shown in Figure 4.5a. The result of this execution is the SET shown in Figure 4.9(b).

Path vars	**Java component paths**	**C++ component paths**	
		Schema S	**Schema S'**
Access paths Θ	⟨book⟩, ⟨book, ◇⟩	⟨book⟩, ⟨book, author⟩, ⟨book, title⟩	⟨book⟩, ⟨book, author⟩ ⟨book, title⟩, ⟨book, ◇⟩
Delete paths Δ	⟨book, author⟩, ⟨book, title⟩, ⟨book, ◇⟩		
Add paths Ω	⟨book, ◇⟩, ⟨book, ◇, author⟩, ⟨book, ◇ title⟩, ⟨book, ◇, ◇⟩		

Table 4.4: Values of access, delete, and add path variables after symbolically executing abstract programs for Java and C++ components shown in Figure 4.4 on the reengineered schema S′ and the expected schema S.

4.7 Finding Errors

The categories of errors, their descriptions, and where they are detected in Viola analysis are shown in Table 4.5. API and PS-2 errors are detected during extracting abstract programs (APs) and symbolic execution respectively. Errors of type PS-1 are detected when comparing the reengineered schema S′ and the expected schema S, and errors of type P2 are caught during the path analysis stage of Viola. We describe algorithms for comparing schemas and path analysis, and we explain how errors are mapped to the source code of the components.

4.7.1 Comparing Schemas

Consider the model of interoperating components shown in Figure 1.2. After executing an abstract program of the component J symbolically on the schema describing the XML

Errors	Description	When Detected
PS-1	Components violate bounds set by schemas on data elements	Comparing Schemas
PS-2	Components attempt to access, delete, or add elements that do not exist in the schemas	Symbolic Execution
P2-1	Components access data elements that are deleted by some other components	Path Analysis
P2-2	Components read or write wrong elements	Path Analysis
API	Incorrect uses of API calls	Extracting APs

Table 4.5: Categories of errors detected by Viola and steps of the analysis at which these errors are detected.

data $\mathtt{D}_1$, a schema $\mathtt{S'}$ is obtained that approximates the data $\mathtt{D}_2$ that would be output by this component. Recall the MSP that is defined as the XML data $\mathtt{D}_2$ should be an instance of the schema $\mathtt{S}$. This property is validated by comparing the schema $\mathtt{S'}$ with the schema $\mathtt{S}$. If these schemas are equal, then the data instance that conforms to one schema also equally conforms to the other schema, and the components that use this data are compatible. Otherwise, components are not compatible with respect to the data and may throw runtime errors. We describe a *bisimulation* algorithm that is used to compare schemas, and we give an example of its use.

Bisimulation

Formalization of schemas as graphs is described in Section 3.3. An efficient method to determine if two graphs are equal is bisimulation [22]. We define *bisimulation* as a binary relation between the nodes of two graphs $g_1, g_2 \in \mathtt{G}$, written as $x \sim y$, $x, y \in \mathtt{V}$, satisfying the following properties:

Property 1 if x is the root of g_1 and y is the root of g_2, then $x \sim y$;

Property 2 if $x \sim y$ and one of x or y is the root node in its graph, then the other node is the root node as well;

Property 3 if $x \sim y$ and `type(`x`)=type(`y`)`, and nodes x, y are not tagged as potentially deleted, and $x \xrightarrow{r_p^q} x'$ in g_1, then there exists an edge $y \xrightarrow{s_k^m} y'$ in g_2, with the same labels `r=s`, `r`$\neq \diamond$ and `s`$\neq \diamond$, and `max(r)=max(s)` and `min(r)=min(s)`, and `type(`x'`)=type(`y'`)`, such that $x' \sim y'$;

Property 4 conversely, if $x \sim y$ and `type(`x`)=type(`y`)`, and nodes x, y are not tagged as potentially deleted, and $y \xrightarrow{l_k^m} y'$ in g_2, then there exists an edge $x \xrightarrow{l_p^q} x'$ in g_1, with the same label `l`, `l`$\neq \diamond$, and `p=k` and `m=q`, and `type(`x'`)=type(`y'`)`, such that $x' \sim y'$.

Two finite graphs $g_1, g_2 \in$ `G` are equal is there exists a bisimulation from g_1 to g_2. A graph is always bisimilar to its infinite unfolding. Computing bisimulation of two graphs starts with selecting the root nodes and applying the above properties. When a relation `(`x`,`y`)` between nodes x and y in a graph is found that fails to satisfy the above properties, then the graphs are determined not equal and the bisimulation stops. This relation is called *offending*.

Example of Bisimulation

We demonstrate how to apply the bisimulation algorithm to show whether two schemas shown in Figure 4.11a and Figure 4.11b are equivalent. The schema G_a shown in Figure 4.11a describes XML data that the component C expects in the basic model shown in Figure 1.2, and the schema G_b shown in Figure 4.11b is reengineered from the component J as described in Section 4.6.4. If the schema G_b is equivalent to the schema G_a, then we can declare both components compatible with respect to the data.

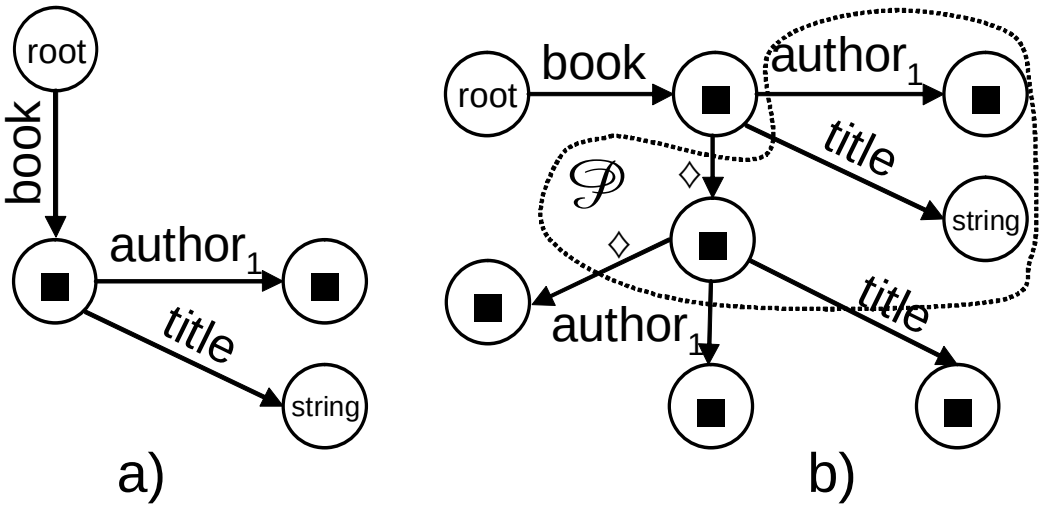


Figure 4.11: Graphs for two XML schemas describing the data shown in Figure 1.1b and Figure 1.1c.

First, we select the root nodes in both schemas which satisfy Property 1 and Property 2. Next, we select relation $\texttt{root} \xrightarrow{\texttt{book}} \blacksquare$ from the schema G_a and check to see that the Property 3 holds for the relation root $\texttt{root} \xrightarrow{\texttt{book}} \blacksquare$ from the schema G_b. Since it does, we determine that the Property 4 holds for both relations, and we proceed to the relation $\blacksquare \xrightarrow{\texttt{author}_1} \blacksquare$ for the schema G_a and the relation $\blacksquare \xrightarrow{\texttt{author}_1} \blacksquare$ for the schema G_b. We determine that Property 3 and Property 4 are violated because the offending relation `author` is tagged as potentially deleted in the schema G_b. Thus, these graphs are not equal, and the verification step fails.

4.7.2 Analyzing Paths

Consider the basic model shown in Figure 1.2 when component `J` adds or deletes some elements from, and component `C` accesses some other elements in the data $\texttt{D}_2$. When symbolically executed on the schema of the data $\texttt{D}_2$, operations of the component `J` modify it

Algorithm 4 Procedures for catching P2 errors

```
Catch-P2-1-Errors( Θ, Δ )
for all p_Θ ∈ Θ do
  for all p_Δ ∈ Δ do
    if p_Δ ⊆ p_Θ then
      print error P2-1: accessing deleted data elements
    end if
  end for
end for

Catch-P2-2-Errors( Θ, Θ' )
if Θ' ⊈ Θ then
  print error P2-2: accessing wrong data elements
  for all p ∈ Θ' − Θ do
    print path p
  end for
end if
```

incorrectly so that the bisimulation algorithm will find offending relations and determine that the schemas `S` and `S'` do not match. However, in general, component `J` may modify elements that lie on different paths from the elements accessed by the component `C`. While the mismatch between schemas may lead to errors, no errors will result from the execution of components in this specific case.

Procedures `Catch-P2-1-Errors` and `Catch-P2-2-Errors` for catching P2 errors are shown in Algorithm 4. The procedure `Catch-P2-1-Errors` takes two parameters: the sets of paths Θ to elements accessed by the component `C` in schemas `S` and `S'`, and the set of paths Δ to elements deleted by the component `J`. This procedure checks to see if each delete path from Δ is the subpath of any access path in Θ. If a delete path p_Δ is a subpath of some access path p_Θ, $p_\Delta \subseteq p_\Theta$, then a P2-1 error is issued that a component may access a deleted data element.

The procedure `Catch-P2-2-Errors` takes two parameters: the set of paths Θ

to elements accessed by the component C in schema S, and the set of paths Θ' to elements accessed by the component C in schema S′. This procedure checks to see if set of paths Θ' is the subset of the set of paths Θ. If $\Theta' \not\subseteq \Theta$, then a P2-2 error is issued that a component may access and potentially read or write values of wrong data elements.

Consider applying these procedures for analyzing paths for the motivating example whose symbolic execution is described in Section 4.6.4. Recall that lists of accessed and modified paths are created when executing the component C on schemas S and S′ symbolically. These lists are shown in Table 4.4. By analyzing these paths some P2 errors can be caught.

The procedure Catch-P2-1-Errors is applied to access paths obtained by executing the C++ component on the schema S′ and the delete paths obtained by executing the Java component. This procedure issues a warning that the elements {book, author} and {book, title} as well as the added element {book, ◇} may be deleted, since the delete paths from the Table 4.4 are subpaths of these access paths. When the C++ component accesses elements {book, author} and {book, title} in the XML data, it may throw a runtime error because the Java component deletes these elements. However, the added element {book, authors} is not deleted, and this specific error is a false positive.

Similarly, the procedure Catch-P2-2-Errors checks to see if the set of access paths obtained by executing the C++ component on the schema S′ is the subset of the set of access paths obtained by executing the C++ component on the schema S. It turns out that the difference between these sets yields the path {book, ◇}. Closer analysis confirms that when accessing a child of the root element book by its sequence number, the C++ component may read the value of the added element authors instead of the intended element author. Thus, in this case Viola issues a correct P2-2 error.

4.7.3 Mapping Errors to Source Code

When an error is found, it is important to map it to the component source code in order to improve the quality of diagnostics. This mapping is accomplished in a series of steps. A table (`C`, `SL`, `EL`, `FSA`) links components to abstract operations identified by the FSAs, where `C` is the component identifier, `SL` and `EL` are the `Starting Line` and the `Ending Line` of the component's source code that contain a sequence of API calls that is accepted by the FSA. Another table (`C`, `SL`, `EL`, `AP`) links a component `C` to some abstract programs `AP`. A pair (`AP`, `SET`) maps abstract programs `AP` to the SETs of these programs. A triple (`SET`, `SETNode`, `FSA`) links the node `SETNode` in a SET to the abstract operations identified by the FSA. This information is collected and stored when running Viola.

When errors are found, the reverse process is applied to use the stored information to locate lines in the source code of components that lead to potential errors. First, paths to data elements are associated with nodes in SETs. Each SET is mapped to an abstract program, and nodes of the tree are mapped to specific abstract operations in this program. From the triple (`SET`, `SETNode`, `FSA`), the abstract operations identified by the FSA are obtained. From the pair (`AP`, `SET`) it is determine the abstract programs `AP` whose `SET` is analyzed. From the table (`C`, `SL`, `EL`, `AP`) the component `C` and the its scope are determined that match the abstract programs `AP`. Finally, from the table (`C`, `SL`, `EL`, `FSA`) that links components to abstract operations identified by the FSAs, exact scope of the component source code is determined that leads to the found error.

Subject Program	Size						Analysis Time, sec		Memory, Mb	Bugs		
	Programs, LOC	Schema, No. of Types	CFG No. of Nodes	APs, LOC	APIs, No. of calls	No. of Components	Generating APs	Error Detection		Expected	Detected	False Positives
Book	109	16	682	16	27	3	22.7	3.2	281	10	16	6
Employees	638	11	2486	30	43	8	90.5	5.1	652	15	28	13
ProbeMsg	921	8	7301	57	82	11	183.6	4.7	794	15	32	14
Homeowners	147	12	662	32	61	4	28.3	1.8	335	15	27	12
Happycoding	372	34	924	47	58	3	50.4	9.2	487	15	36	21
papiNet	11048	1653	46934	916	1281	17	1958.3	28.3	1396	30	103	58
MetaLex	18479	66	63937	835	1526	12	2739.0	42.9	1621	15	59	42

Table 4.6: Experimental results of testing Viola on commercial and open source projects.

4.8 Prototype Implementation

A prototype implementation of the Viola architecture shown in Figure 4.2 is based on the EDG Java front end C++ and Java parsers [13] and an MS XML parser. FSAs, abstract programs, schemas, SETs, and even output errors are provided in the XML format, and we use the ROOF framework [48] to process XML documents. Our prototype implementation included the analysis routines, symbolic executor, schema comparator (bisimulator), and path analyzer with our error checking algorithms. We wrote these components of Viola in Java and interfaced them with EDG front end parsers written in C++ using the Java Native Interface. Our implementation contains 9,000 lines of Java and C++ code.

4.9 Experimental Evaluation

In this section we describe the methodology and provide the results of experimental evaluation of the Viola on subject programs.

4.9.1 Subject Programs

We applied Viola to two commercial programs for legal and paper supply chain domains (the latter uses a large XML schema with over 1,600 types), and to open source programs. One commercial application was written for a legal office, and it used the Metalex schema. MetaLex is an open XML standard for the markup of legal sources [51]. The other commercial application was written for papiNet, a transaction standard for the paper and forest supply chain [52]. The combined source code of both commercial applications was about 30,000 lines of C++ and Java code.

Open source programs are taken from different XML projects posted on the Internet. The Book and Employees projects contain programs that generate, access, and manipulate XML data that describe books and employees respectively [2]. ProbeMsg is an application that creates XML data and sends it as a stream to a different application [11]. HomeOwners applications illustrates the use of the Xerces DOM parser to access and manipulate XML data that contains information on homeowners includes their names, addresses, and closing dates of house purchases [6]. Finally, the Happycoding website contains different applications that exchange XML data [5]. Open source programs are small, ranging from less than a hundred to less than a thousand lines of code.

4.9.2 Methodology

To evaluate Viola, we carried out two experiments that explore how effectively it catches errors in the existing interoperating components, and how the precision of the data flow analysis affects the number of false positives. We inject different bugs in the subject programs and test how Viola catches them. We carried out our experiments using Windows XP that ran on Intel Pentium 4 3.2GHz dual CPUs and 2Gb of RAM.

Injecting Bugs

We wish to evaluate how effective Viola is in catching bugs from all categories of errors which are described in Section 4.2. Even though the first attempt to run Viola on subject programs yielded few faults, they were only API and P2-2 type faults with one real P2-2 type bug which we confirmed. To test Viola on other types of bugs, we asked several graduate computer science students, each with at least two years of C++ and Java programming experience to insert bugs into subject programs. Initially, these students were not aware about our study and only one student knew how to use Xerces XML DOM parser API calls. After giving instructions on using Xerces and MSXML DOM parser API calls and explaining the goals of this project, each student was asked to inject certain types of bugs in the source code of components. Students were asked to make these bugs as realistic as possible based on their experience and the knowledge of the subject programs. It was up to students to select locations in the source code for the injected bugs and create test cases to ensure that these bugs behave as intended.

When delivered, the subject programs were tested without the injected bugs (the code that students wrote to inject bugs was initially commented out), and then with injected bugs. We excluded injected bugs that introduced compounded bugs in the logic of the existing code accessing and manipulating XML data. That is, if the code for an injected bug led to one or more other bugs, then the bugs related to the injected bug were excluded. The number of bugs remaining in the components source code after exclusions is reported in Table 4.6.

Experiments

We ran two experiments with Viola. In the first experiment, we run Viola on subject programs with injected bugs. The goal of this experiment is to determine how effective Viola is at catching bugs. We report the sizes of the original programs in *lines of code (LOC)*, number of nodes in the corresponding CFGs, and the number of platform APIs used by programs to access and manipulate XML data. We measure times taken by Viola to produce abstract programs and to report bugs. The time it takes to catch API errors is included in the time of producing abstract programs because these errors are a part of building program abstraction routines. We also measure how much memory Viola consumes.

We inspect the source code of the subject programs before running Viola, and we modify source code to inject bugs. Thus, we expect Viola to catch a certain number of bugs. Viola may catch more bugs for two reasons. It is possible that we missed some existing bugs during the code inspection, and Viola can report false positives. We report the number of expected bugs, detected bugs, and false positives.

We are also interested in the breakdown of bugs based on their types as described in Section 4.2. We report the number of expected and found bugs for each type of errors. This information helps us to identify the effectiveness and precision of Viola for different types of problems. These results show for what classes of bugs Viola is most effective, and we can use these results to improve the effectiveness of Viola.

The goal of the second experiment is to evaluate the effect of having precise names of data objects versus symbolic variables in abstract programs on the Viola's rate of false positives. Since program abstractions are approximations of actual programs, symbolic variables are often used in place of names of data elements. For example, if a program has a variable whose value is a name of a data object and the value of this variable is computed

at runtime, then Viola uses a symbolic variable with an undefined value to denote this data element.

Consider a fragment of C++ code shown in Figure 4.12. The value of the variable `element` is computed at runtime and it is the name of the XML data elements accessed by the `selectNodes` and `get_item` API calls. In general, it is an undecidable problem to determine the values of string expressions at compile time. When extracting an abstract program the variable `element` is replaced with some symbolic variable whose value is undetermined. However, in the C++ code fragment, API calls are executed when the condition `element == "authors"` is met. It means that in the corresponding abstract program the value `authors` can be used in place of the symbolic variable.

```
string element;
int index = 0;
..............
if( element == "authors" ) {
    bookNode->selectNodes(element, &list);
    list->get_item((long)index, &node );
}
```

Figure 4.12: A fragment of C++ code accessing a data element using DOM API calls.

In order to compute precise values for abstract programs we need to use more sophisticated analysis that takes more time and resources. In order to know that this additional effort is justified, we manually replace symbolic variables in abstract programs with precise names of data objects, and we measure the number of false positives. To determine the names of data elements we inspected the source code manually and ran programs collecting values for variables. After determining names for data elements, we went to abstract programs generated by Viola and manually replaced the corresponding symbolic variables

with names of data elements. We symbolically execute abstract programs before and after we replace symbolic variables with data element names, and we measure the number of false positives detected by Viola.

The goal of this experiment is to plot the number of false positives as a function of the percentage of precise names of data element versus symbolic variables denoting these elements in abstract programs. If the number of false positives does not decrease, then improving data flow analysis to detect precise names of data elements will not lead to increased effectiveness of Viola. On the contrary, if Viola reports fewer false positives, the increased precision and subsequently the cost of data flow analysis is justified to improve our approach.

Threats to Validity

Our subject programs are of small to moderate size because it is surprisingly difficult to find a large-scale open source project whose components exchange XML data while there are plenty of such commercial projects. We chose small open source programs because we did not have access to large-scale application with components interacting using XML data. Our explanation is that open source projects are mostly applications with specific user-defined purposes. We have not found an open source project that integrated two or more open source applications that exchange XML data. In commercial application development it is important to make applications exchange information in a timely manner in order to achieve competitive advantage. Therefore there are various large-scale commercial projects whose goal is to write interoperating components that exchange XML data with high quality while reducing development costs.

Large applications that interact using XML data might have different characteristics

compared to our small to medium size subject programs. Increasing the size of applications to millions of lines of code may lead to a nonlinear increase in the analysis time and space demand for Viola. If this happens, then more time should be spent on making Viola scalable.

4.9.3 Results

Experimental results with testing Viola on subject programs are shown in Table 4.6. This table is divided into five main columns. The first column gives the name of the subject project. Next column, `Size`, contains five subcolumns reporting sizes of subject programs, number of types in XML schemas, number of nodes in CFGs, LOCs of abstract programs, and the numbers of API calls in the subject programs respectively. The third column reports the analysis time in seconds and contains two subcolumns for the time it takes to generate abstract programs and the time to catch errors. The following column shows the maximum memory consumption when running Viola. Finally, the `Bugs` column reports the number of bugs. Its first subcolumn shows the number of expected bugs for each subject project (and we know the number of the expected bugs since we injected them), the subcolumn `Detected` gives the number of bugs caught by Viola, and the subcolumn `False Positives` shows the number of false positives. For example, the `ProbeMsg` subject program was expected to have `15` bugs, but after running Viola `32` bugs were detected, `14` of which were confirmed through manual code inspection as false positives. It means that three more bugs were missed during the initial code inspection.

A breakdown of expected and detected errors by their types for each subject program is shown in Table 4.7. The difference between the number of detected and expected errors is the number of false positives. A graphic distribution of the number of detected and expected errors by their types is shown in Figure 4.13. Almost a half of all false positives

Project	API Errors		P2-1		P2-2		PS-1		PS-2	
	Expected	Found	Expected	Found	Expected	Found	Expected	Found	Expected	Found
Book	2	2	2	5	2	2	2	3	2	4
Employees	3	5	3	6	3	7	3	4	3	6
ProbeMsg	3	4	3	12	3	6	3	3	3	7
Homeowners	3	3	3	8	3	9	3	3	3	4
Happycoding	3	6	3	11	3	10	3	5	3	4
papiNet	6	17	6	25	6	31	6	13	6	17
MetaLex	1	5	4	21	1	13	2	5	7	15

Table 4.7: A breakdown of real and detected errors by error types.

are generated by the P2 type of errors. Recall that P2 errors occur in components that access data elements that are deleted by some other components (P2-1) and by components that read or write wrong elements (P2-2). The reason for the false positives is that Viola approximates paths through the data during symbolic execution. This approximation results in many spurious paths that are not accessed or modified when components interoperate at runtime. However, based on our experience it is easy to verify whether the code that produces a path has a bug in it by running the application and observing its behavior.

The results of evaluating the effect of having precise names of data objects versus symbolic variables in abstract programs on the Viola's rate of false positives are shown in Figure 4.14. The horizontal axis shows the percentage of symbolic variables that we replaced in abstract programs with actual data object names, and the vertical axis shows the number of false positives. We observe that when close to 30% of symbolic variables are replaced with actual object names, the number of false positives decreases approximately three times. Such a significant drop in false positives justifies the use of elaborate data flow analyses that help to improve the precision of the generated abstract programs.

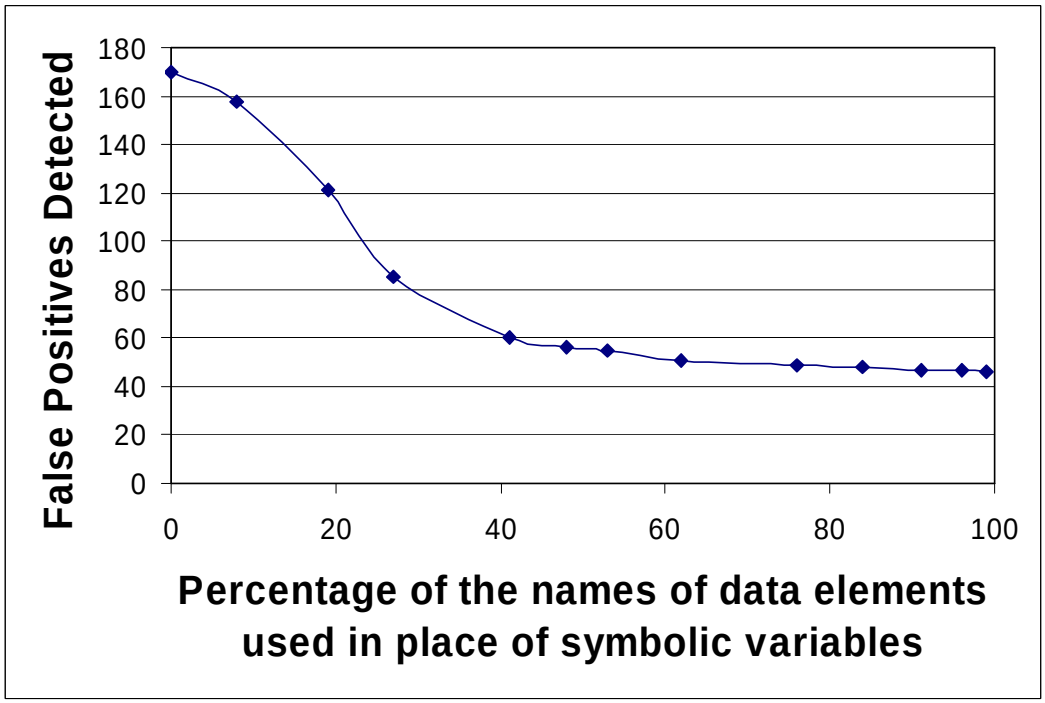


Figure 4.13: Dependency of false positives issued by Viola from the percentage of precise names of data elements versus symbolic variables used in abstract programs.

4.9.4 Recommendation

As our experiments show, Viola takes around 45 minutes to run on a system with over 20,000 lines of code. Since programmers routinely run compilers on the code they write many times a day, using Viola this way is prohibitive since it takes too much time to check the code. We suggest that Viola should be used after "freezing" a release and before testing starts. This way test engineers obtain warnings about potential bugs, and they can define test plans to validate these warnings.

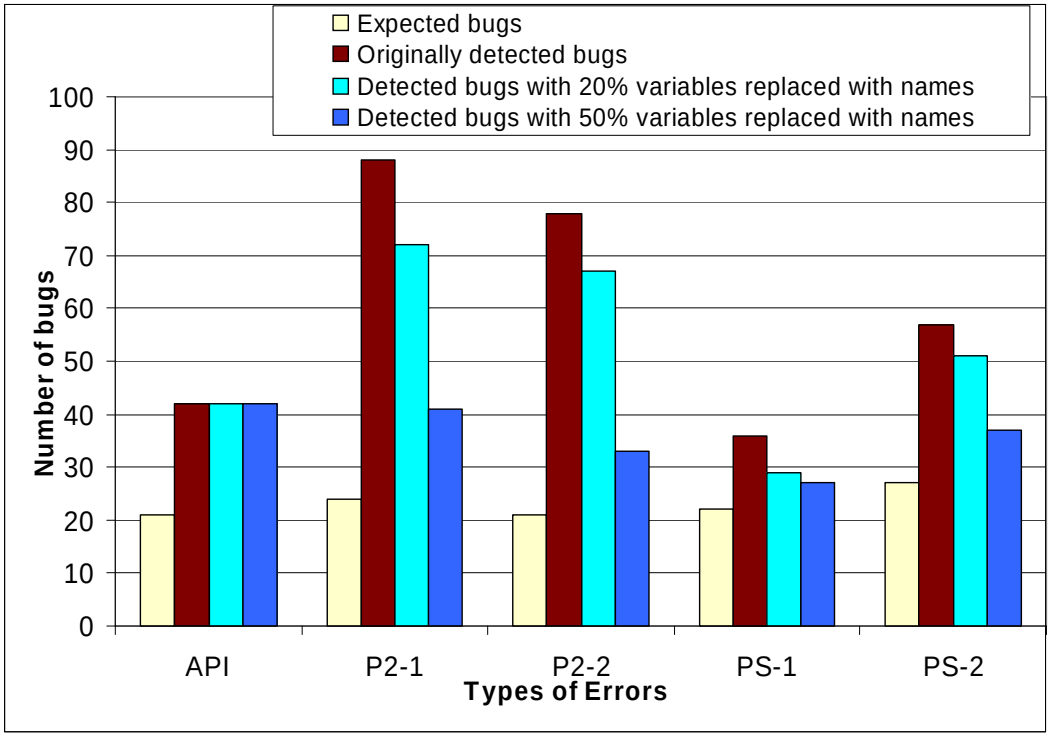


Figure 4.14: The distribution of the number of detected and expected errors by their types. The difference between the number of detected and expected errors is the number of false positives.

4.10 Summary

We present a novel solution called Viola for finding bugs in components interacting via XML data. Viola is a helpful bug finding and testing tool that assists test engineers by detecting a situation at compile time when one component modifies XML data so that it becomes incompatible for use by other components. We implemented a prototype of Viola in C++ and Java using EDG Java and C++ and XML parsers. Viola's static analysis mechanism reports some potential errors for a system of interoperating components. We tested Viola on open source and commercial systems, and we detected a number of known and

unknown errors in these applications with good precision thus proving the effectiveness of our approach.

Concrete instances of the model shown in Figure 1.2 are common even in small software systems. Two components that inadvertently modify the same environment variable work fine when running on separate computers, however, they malfunction when put on the same machine. The reason is that the first component sets the value of the environment variable, and this value is not recognized by the other component thus leading to an error that is extremely difficult to catch. Similar situations occur when components use databases, XML and HTML data, or system registries.

One of the applications of our work is to detect errors when components interact via XML data using XML parsers as underlying platforms. However, components may violate each other's properties by using any kind of data hosted by different platforms. This is known as emergent phenomena in complex systems, specifically software, that occur due to interactions between the components of a system over time. Emergent phenomena are often unexpected, nontrivial results of simple interactions of simple components. Currently, no compiler checks interoperating components for violations that occur as results of these interactions, even when components are located within the same program. We believe that this thesis is the first research step in this direction.

Chapter 5

Related Work

We divide related work into three categories: middleware services that enable component interoperability, language and type checking approaches, and formal verification methods.

Component interoperability is a functional aspect that programmers should be able to add to or remove easily from existing software. If such changes are complex then interoperable software is hard to maintain and evolve. Low-level approaches (e.g. RPC, message passing, Document Object Model) provide APIs that enable programs to cross process boundaries in order to access foreign objects and invoke their methods. This approach is tedious and error prone because it requires the steep learning curve to master various vendor-dependent APIs that deal, for example, with marshalling and unmarshaling data.

IDL-based approaches (DCOM, CORBA, Mockingbird) require programmers to define interfaces in an Interface Definition Language (IDL) that are implementation language neutral and can be translated into language-dependent client and server classes using an IDL compiler. This approach suffers from multiple drawbacks; notably the necessity

to deal with an additional type system (the IDL), and to maintain client and server sets of code. In addition, this approach is hardly transparent since programmers are required to use a complex, hard-to-learn platform-dependent API.

IDL-based approaches are also difficult to maintain and evolve because reversing the initial decision to share objects once the client/server wrapper code is generated and implemented requires software to be rewritten. Suppose that a programmer creates a Java program whose objects are not shared by different programs. If this decision is reversed then these previously non-shared Java classes should be recoded as interfaces in an IDL, and then client/server code in implementation languages should be generated using an IDL compiler. Clearly, this approach can require major rework of the existing code that is laborious and costly when applied to large software projects. However, much larger amount of work is required if programmers decide to make a Java class not to be interoperable after its IDL-based specification is created and client/server code is generated and implemented. This class has to be recoded by removing its IDL compiler generated code and writing its new implementation. To do this change requires significant programming investment, and is very expensive at the maintenance stage of a project.

When a programmer creates an interface using IDL s/he can select certain types to declare interface members because they may closely map to desired types in the selected implementation language. For example, if an IDL interface is used to generate C++ wrapper code then IDL types that define this interface are likely to be C++-friendly. When the same IDL interface is used to generate Java wrapper code, programmers may replace some IDL types with Java-friendly types. Mockingbird [25] is an IDL-based tool for developing interoperating distributed components that generates adapter code that reconciles existing friendly IDL-based data types. However, this approach leads to software that is difficult to

maintain and evolve. It also suffers from problems with IDL-based approaches described in [81].

PolySPIN [27] did away with the IDL approach by directly mapping types between different FTSs. A tool called PolySPINNER analyzes class definitions written in different languages, matches their structure, and generates code that enables objects of matched classes to interoperate seamlessly, i.e. if objects of types t1 and t2 exist in different FTSs, for example, in Java and C++ correspondingly. Both types have to exist in Java and C++ to begin with. After applying PolySPIN approach, a call to method f1 of an object of type t1 is translated by the generated code into the call to the matched method f2 of some object of type t2. The problem with this approach is that it requires complex matching mechanism to determine isomorphisms between foreign types.

Exu [54] is an alternative approach to IDL. It enables C++ classes to be accessed from Java classes using Java Native Interface (JNI). For any C++ class Exu generates a corresponding Java proxy classes and JNI-based interoperability code. This approach is limited as it works only for Java classes that interoperate with C++ classes. In addition, it is difficult to maintain and evolve Exu-based systems because generated isomorphic classes may be changed by programmers.

Finally, generator-based approaches (e.g. JAXB, Apigen) can do automatic mapping for individual languages. For example, if an XML schema contains thousands of types then thousands of corresponding classes are generated in a host programming language that map to these XML types. This approach leads to serious problems with evolution and maintenance of generated code, like a complex naming mechanism, and results in a significantly increased compilation time of the system. In addition, as it often happens in commercial development, a schema may not exist to generate corresponding classes at the time of writ-

ing an interoperating program. For example, it is customary to write prototype code that manipulates some XML data before a complete XML schema that describes this data is created, without a schema most generator approaches fail. Various generators are used as part of programming environments and as standalone tools to analyze FTSs and generate corresponding types in host programming languages. Most are generators that take XML schemas and generate corresponding classes in Java and C++ [15][7][20]. This approach requires sophisticated name management software and produces software that is difficult to maintain and evolve.

One of the perceived benefits of existing approaches to component interoperability is that type checking is free since host language compilers perform it when compiling generated types and interoperability code. For example, given an XML schema, a generator can produce corresponding classes in Java along with interoperability code. A Java compiler performs type checking of the generated classes. Suppose that during the maintenance phase of interoperating components the XML schema has been changed. The Java compiler is not aware of this change and it would compile the generated classes without producing any warnings. However, the resulting program fails at run time because the interoperability code attempts to access XML objects that are either changed or do not exist. Obviously, this type checking does not meet its primary objective i.e., to produce errors when semantic inconsistencies with interoperating components exist.

Polilinguality is implemented by virtual machines (VM) such as JVM and .Net CLR. These platforms also have limitations. Each has its own FTS and does not cover all other existing type schemas which exist beyond their scope (for example, there is no virtual machine that would reify HTML types to Java type system). Not only do separate low-level APIs exist for each platform, they are also vendor-dependent. These APIs introduce signif-

icant complexity and nonuniformity into programming FTSs, let alone the steep learning curve required to master each platform. Extending any VM platform to support a new type system is difficult because it is very complex and fragile. Various bridges are offered to interwork different components [40]. For example, a JVM-COM bridge makes Enterprise Javabeans (EJBs) runs as COM objects. However, bridges offer complicated APIs that are vendor-dependent. Again, using bridges for FTS programming leads to nonuniform and complex code and requires time to learn their APIs.

Related work on verifying and testing software that accesses and manipulates XML data falls into two major categories: systems that use type checking and verification techniques for XML manipulating programs, and model checkers that automate the verification process for XML-unrelated software artifacts.

An automated verification system for XML data manipulation operations translates XML data and XPath expressions to Promela, the input language of the SPIN model checker [42]. The techniques of this system constitute the basis of a web service analysis tool that verifies linear temporal logic properties of composite web services. Unlike Viola, this system cannot be applied to arbitrary C++ and Java programs, however, Viola can use its ideas to further improve the verification process of interoperating components.

Currently, there are various language design projects that address this problem by making XML a first-class data type at the language level (e.g., XJ, XLinq, Xact, and Cω) [49][59][29]. While some success is demonstrated, there are three major problems with these approaches. First, they impose additional type systems and coding practices on programmers, and it serves as an inhibiting factor for adopting these approaches. Next, for these approaches to be sound (i.e., to ensure the absence of bugs if the compiler reports no errors) programmers should not compute names of XML data elements at runtime. This

constraint limits programmers to a small class of applications. Finally, given the large number of legacy systems that has been written and are being written using API calls exported by XML parsers, it is unlikely that these systems will be rewritten adhering to some of these approaches.

Generator-based approaches (e.g. JAXB, Apigen, Castor) can do automatic mapping for individual languages. For example, if an XML schema contains thousands of types then thousands of corresponding classes are generated in a host programming language that map to these XML types. This approach leads to serious problems with evolution and maintenance of generated code, like a complex naming mechanism, and results in a significantly increased compilation time of the system. Various generators are used as part of programming environments and as standalone tools to generate corresponding types in programming languages. Most are generators that take XML schemas and generate corresponding classes in Java and C++ [18][19][20]. This approach requires sophisticated name management software.

Our work uses a variety of ideas introduced in different model checkers. Most of these model checkers use the same abstract-verify-refine verification paradigm that Viola is based on. Unlike other model checkers that determine whether programs match specifications or satisfy certain logic predicates (invariants), Viola concentrates on verifying that two components interoperating using XML data do not violate the predefined safety properties. In doing so, Viola employs many common techniques used in other model checkers, but in a novel way.

MAGIC is a model checker that creates models of C components using the method of predicate abstraction, and compositionally verifies computed models using SAT solvers [32]. Like our research, MAGIC uses state machines called linear transition systems to ex-

press the desired behavior of systems, and it operates on the source code of C components. By contrast, the Viola's goal is to verify that two components do not violate each other's properties by computing incorrect data.

MOPS is a model checker for verifying that programs do not violate predefined security properties [33]. Like our research, MOPS uses FSAs to describe security properties of programs source code, and it computes models of verified programs by analyzing API calls that affect security properties. Unlike our approach MOPS is used strictly to discover violations of security properties rather than to verify component interoperability.

SLAM is a model checker for C programs that is based on the method of counterexample-driven refinement [26]. SLAM extracts boolean programs from C programs and performs the reachability analysis on the extracted boolean programs by combining interprocedural dataflow analysis and the binary decision diagrams techniques. If a path that leads to an error is not reachable, then SLAM tools analyze the feasibility of executing this path in the actual program by refining boolean programs. Like our research, SLAM builds abstract programs and performs path analysis in order to catch errors. Unlike Viola, SLAM does not address verification of interoperating components with respect to the safety properties defined in terms of exchanged data, and SLAM analyzes execution paths in programs, not in the data that they manipulate.

Blast is a model checker for C programs based on a lazy abstraction algorithm. BLAST uses specifications for temporal properties written in C syntax [50]. For model checking Blast uses the predicate abstraction method [46] to find bugs or prove the specification.

Moped is a combined linear temporal logic and reachability checker for pushdown systems [78]. Since pushdown systems can express programs with recursive procedures, its

power is equal to or greater than that of Viola's. Moped can also process boolean programs and interact with the SLAM checker.

SLAM, MOPED, and BLAST use the predicate abstraction method [46]. Like our research, these model checkers can verify safety properties of programs using their source code. However, these model checkers are not designed to verify properties of interoperating components that use platform API calls to interact by accessing and modifying data. By contrast, our solution abstracts away properties of interoperating components that are not related to their interacting using data, and it analyzes paths in data computed by symbolically executing abstract programs

A static program analysis method checks structural properties of code by computing an initial abstraction of the code that over-approximates the effect of function calls [79]. Like Viola, this method then refines the computed abstractions by inferring a context-dependent specification for each function call, so that only as much information about a function is used as is necessary to analyze its caller. Rather than concentrating on specifications for function calls, Viola analyzes API calls that access and manipulate XML data.

Navigation traversal paths are integral part of FOREL. Until recently, the automation of traversal of object structures using succinct representations has been unique to Demeter [39]. The connection between reification expressions that navigate to foreign objects and traversal specifications in Demeter is following. The latter is used to generate code that performs the required traversal to the destination object while the former is the code that performs the traversal. In adaptive programming, a change of requirements for a foreign program induces changes of the traversal specifications and the subsequent regeneration of the code that manipulates foreign objects.

XML is a new standard for defining and processing markup languages for the web

that uses grammars (also called document type definitions or schemas) to define a markup language for a class of documents. These grammars are akin to class graphs in Demeter. Because of significance of XML as a data interchange standard, an effort is made to integrate XML in the type system of various languages [69][68][49].

FOREL language is similar in its functionality to XPath, a language introduced by W3 Consortium to select subsets of XML document elements [34]. XPath expressions are used to describe sets of objects, in the sense that the value of an expression is an unordered collection of objects without duplicates. The way elements are selected in XPath is by navigation, somewhat resembling the way one selects files from an interactive shell, but with a much richer language. XPath was proposed as input to a universal object model walker for arbitrary Java objects [8].

In the context of programming languages, traversals are frequently used as a part of attribute grammars, for traversing abstract syntax trees [44]. Using conventional programming techniques, the details of traversals must be hard-coded in the attribute grammar; this fact makes attribute grammars hard to maintain, say in the case of some modifications in the grammar [56]. In the Eli system [47], this problem is addressed by separating the details of the grammar from the underlying algorithm, using traversal specifications which basically correspond to single edge strategy graphs.

Meta-programming techniques have also been developed for traversals. In [31], a simple kind of traversal (corresponding to a one layer tree graph) is used in a metaprogram; this traversal scans all objects and executes the specified code at the desired targets.

ArchJava is an example of the most recent sophisticated language support for user-defined architectural connectors [24, 23]. ArchJava enables a wide range of connector abstractions, including caches, events, streams, and remote method calls. Developers can

describe both the run-time semantics of connectors and the typechecking semantics.

Chapter 6

Conclusion, Recap, and Future Work

Our work extends the state-of-the-art research in software interoperability in two directions. First, we proposed an abstraction that simplifies modeling component interoperability. Specifically, foreign objects (i.e., objects that are not defined in a host programming language) are viewed as graphs and abstract operations are used for accessing and manipulating these objects. These operations navigate to data elements, read and write data, add and delete data elements, and load and save data. We used these operations as a basis for the framework and bug finding approaches.

Next, we built a framework called ROOF and a language extension called FOREL based on this model. We showed that ROOF with FOREL is a simple and effective way to develop easily maintainable and evolvable systems of interoperating components by reifying foreign type instances and their operations into first-class language objects and enabling access to and manipulation of them. By doing so we were able to hide the tremendously ugly, hard-to-learn, hard-to-maintain, and hard-to-evolve code that programmers must write or generate today, i.e., we simplified code of interoperating components, making it scalable

and easier to write, maintain, and evolve.

Quality of the code of interoperating components and their scalability are critical for large-scale applications. Weaving interoperability into the fabric of enterprise-level architectures often reduces the scalability of the resulting system. In this thesis we analyzed the sources of nonscalability for systems of interoperating components. Our analysis is based on presenting a clique architecture for systems of interoperating components, in which nodes represent components and edges represent platform APIs needed for interoperability.

The complexity of a system of interoperating components is approximately the number of edges in the clique architecture. That is, when the number of edges is minuscule, the complexity of a system of interoperating components 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, the complexity of a system of interoperating components is $O(n^2)$. This is not scalable.

A large-scale system of interoperating components is a system where the number of edges (APIs) is excessive. Such systems are common and are notoriously difficult to develop, maintain, and evolve. Current approaches do not support large-scale systems of interoperating components well. They are often limited to specific languages (e.g., typical CORBA platforms allow Java, C++, etc. programs to interoperate, but there are no facilities for accessing HTML or XML data or objects in C# programs). This leads to a proliferation in tools and their API calls, which noticeably increases the accidental complexity of the resulting code, loss of uniformity in the way programs are written, thus rendering resulting systems extremely difficult to maintain and evolve [30].

Among many benefits, ROOF defines a single API platform that all programmers can use; so instead of having $O(n^2)$ possible API platforms for achieving component-to-component communication, a single, standard, and clear set of API calls is used. New framework implementations are easy to add, and consequently, this is a scalable approach.

While ROOF and FOREL offer new approaches for developing interoperating components, many components are still written using low-level platform API calls. It is not likely that millions of lines of legacy software would be replaced in the near future using ROOF and FOREL (although we hope that it will!). In the meantime, we used our abstraction to design and build Viola, a novel solution for finding bugs in components interacting via XML data and helping test engineers to validate reported errors. Viola creates models of the source code of components and computes approximate specifications of the data (i.e., schemas) that these components exchange. The input to Viola is the component's source code, schemas for the XML data used by these components, and FSAs that model abstract operations on data with low-level platform API calls. These FSAs are created by expert programmers who understand how to use platform API calls to access and manipulate XML data.

Viola uses control and data flow analyses along with the provided FSAs to extract abstract operations from the component source code. Next, these operations are symbolically executed to compute approximate schemas of the data that would be output by these components. That is, given the schema of the input data, Viola reengineers the approximate schema of the data that would be output by some component from its source code.

The reengineered and expected schemas are compared to determine if they match each other. If a mismatch between them is found, it means that some component modifies the data incorrectly so that runtime exceptions may be thrown by other components using

this incorrect data. To confirm this, Viola analyzes paths to data elements accessed and modified by these components to determine whether the schema mismatch results in actual errors. Sequences of operations leading to some potential errors are reported to help test engineers validate and reproduce errors.

Viola is a helpful bug finding tool whose static analysis mechanism reports some potential errors for a system of interoperating components. We tested Viola on open source and commercial systems, and detected a number of known and unknown errors in these applications with good precision thus showing the potential of this approach.

Concrete instances of the model shown in Figure 1.2 are common even in small software systems. Two components that inadvertently modify the same environment variable work fine when running on separate computers, however, they malfunction when put on the same machine. The reason is that the first component sets the value of the environment variable, and this value is not recognized by the other component thus leading to an error that is extremely difficult to catch. Similar situations occur when components use databases, XML and HTML data, or system registries.

One of the applications of our work is to detect errors when components interact via XML data using XML parsers as underlying platforms. However, components may violate each other's properties by using any kind of data hosted by different platforms. This is known as emergent phenomena in complex systems, specifically software, that occur due to interactions between the components of a system over time. Emergent phenomena are often unexpected, nontrivial results of simple interactions of simple components. Currently, no compiler checks interoperating components for violations that occur as results of these interactions, even when components are located within the same program. We believe that this thesis is the first research step in this direction.

Our work has limitations. It is not clear what effort is required in general to create FSAs for different platform APIs. While it is possible to use systems to extract these FSAs from the source code of components, it remains to be seen whether automatically extracted FSAs have enough precision to be used in Viola. When error checking C++ components that use pointers, especially function pointers, the number of false positives for API errors increases significantly. In addition, if no explicit names of data objects are used in components, then the number of false positives would be excessive. However, in our experience once the source of a possible error is located, determining whether it leads to actual errors is easier.

The goal of my future work is to improve the productivity of software engineers developing interoperating components. One aspect of this agenda is providing software engineers with reliable solutions for making software. It means that such solutions should enable programmers to develop interoperating components without writing significant amount of additional code, and to provide tools to find bugs in written code at compile time. Based on the current state-of-the-art of software interoperability I believe that there are ample opportunities for novel research in this area. Not only do I want to develop useful tools and techniques for creating and maintaining interoperating components, but I also want to deepen my understanding of the underlying principles behind component interoperability and use these principles to automate various aspects of error detection using program analysis and model checking.

I would like to expand my approach to detecting errors in interoperating components that lead to *byzantine failures*[60]. Specifically, I am interested in finding bugs that result from operating on incorrect data. For example, one component expects to receive data that denotes salary, but the other component sends zip codes. Since salaries and zip codes

are expressed as integers, detecting such errors is difficult. I plan to develop a framework and tools that developers can use to find these bugs.

Most broadly, my future research will address a range of issues in software reliability and evolution of interoperating components. Using program analysis and model checking techniques, I plan to develop algorithms for finding errors in interoperating components statically, design effective and efficient methods for composing components, and to detect invariants and other useful assertions in large-scale systems of interoperating components. One of the major uses of invariants and assertions is to help programmers understand legacy systems of interoperating components. A Bell Labs study shows that up to 80% of programmer's time is spent discovering the meaning of legacy code when trying to evolve it [38], and Corbi reports that up to 50% of the maintenance effort is spent on trying to understand code [36]. Thus, the extra work required to discover invariants and assertions in interoperating programs is likely to reduce development and maintenance time, as well as to improve software quality.

Bibliography

[1] Adobe PDF/XML architecture - working samples. *http://partners.adobe.com/public /developer/en/xml/AdobeXMLFormsSamples.pdf*.

[2] The book and employees projects. *http://totheriver.com/learn/xml/xmltutorial.html*.

[3] Building software that is interoperable by design. *http://www.microsoft.com/mscorp/execmail/2005/02-03interoperability-print.asp*.

[4] eXtensible Markup Language (XML). *http://www.w3.org/XML/*.

[5] The happycoding website. *http://www.java.happycodings.com/XML/index.html*.

[6] Homeowners applications. *http://www.sambito.net/AddExampleWeb/navJava.htm*.

[7] Java architecture for xml binding (jaxb). *http://java.sun.com/xml/jaxb/*.

[8] Jaxen web site. *http://jaxen.org*.

[9] Kla-tencor's archer analyzer automated, real-time overlay metrology analysis. *http://www.kla-tencor.com/products/archer10/archer_analyzer_tech_factsheet.html*.

[10] Kla-tencor's software boosts overlay metrology. *http://www.siliconstrategies.com/story/OEG20020708S0052*.

[11] The probemsg project. *http://www.akadia.com/services/java-xml-parser.html*.

[12] Programmar web site. *http://www.programmar.com/*.

[13] Edison Design Group. *http://www.edg.com*.

[14] XML Schema. *http://www.w3.org/XML/Schema*.

[15] xadl 2.0 project, apigen for xarch schemas. *http://www.isr.uci.edu/projects/xarchuci/tools-apigen.html*.

[16] *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries.* Institute of Electrical and Electronics Engineers, January 1991.

[17] *Cost Analysis of Inadequate Interoperability in the* U.S. *Capital Facilities Industry, GCR 04-867*. NIST, August 2004.

[18] *Institute for Software Research, University of California, Irvine, x*ADL *2.0 project,* A*pigen for x*A*rch schemas,*. http://www.isr.uci.edu/projects/xarchuci/tools-apigen.html, 2004.

[19] *Sun Microsystems, Java Architecture for XML Binding (JAXB),*. http://java.sun.com/xml/jaxb, 2004.

[20] *Castor XML databinding framework,*. http://www.castor.org/xml-framework.html, 2005.

[21] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semi-structured Data and* XML. Morgan Kaufmann, 1999.

[22] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semi-structured Data and* XML. Morgan Kaufmann, October 1999.

[23] J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning with archjava. In *Proc. European Conference on Object-Oriented Programming*, 2002.

[24] J. Aldrich, C. Chambers, and D. Notkin. Archjava: Connecting software architecture to implementation. In *International Conference on Software Engineering (ICSE), Orlando, Florida*, pages 187–196, 2002.

[25] J. S. Auerbach, C. Barton, M. Chu-Carroll, and M. Raghavachari. Mockingbird: Flexible stub compilation from pairs of declarations. In *ICDCS*, pages 393–402, 1999.

[26] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

[27] D. J. Barret. *Polylingual Systems: An Approach To Seamless Interoperability*. PhD dissertation, University of Massachusets at Armherst, May 1988.

[28] D. J. Barrett, A. Kaplan, and J. C. Wileden. Automated support for seamless interoperability in polylingual software systems. In *SIGSOFT FSE*, pages 147–155, 1996.

[29] G. M. Bierman, E. Meijer, and W. Schulte. The essence of data access in c*mega*. In *ECOOP*, pages 287–311, 2005.

[30] F. P. Brooks. *The Mythical Man-Month: Anniversary Edition.* Addison-Wesley, August 1995.

[31] R. D. Cameron and M. R. Ito. Grammar-based definition of metaprogramming systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(1):20–54, 1984.

[32] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Trans. Software Eng.*, 30(6):388–402, 2004.

[33] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *ACM Conference on Computer and Communications Security*, pages 235–244, 2002.

[34] J. Clark and S. DeRose. Xml path language (xpath). 1999.

[35] L. A. Clarke and D. J. Richardson, editors. *Symbolic evaluation methods for program analysis.* Prentice-Hall, 1981.

[36] T. A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[37] B. Curtis, H. Krasner, and N. Iscoe. A field study of the software design process for large systems. *Commun. ACM*, 31(11):1268–1287, 1988.

[38] J. W. Davison, D. Mancl, and W. F. Opdyke. Understanding and addressing the essential costs of evolving systems. *Bell Labs Technical Journal*, 5(2):44–54, 2000.

[39] Demeter research group. *http://www.ccs.neu.edu/research/demeter/*, 2005.

[40] W. Emmerich. *Engineering Distributed Objects*. John Wiley, July 2000.

[41] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV*, pages 232–247, 2000.

[42] X. Fu, T. Bultan, and J. Su. Model checking XML manipulating software. In *ISSTA*, pages 252–262, 2004.

[43] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *ICSE*, pages 179–185, 1995.

[44] G. Goos and W. Waite. *Compiler Construction*. Springer Verlag, 1984.

[45] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *International Conference on Software Engineering, Edinburgh, UK*, pages 1–18, 2004.

[46] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.

[47] R. W. Gray, S. P. Levi, V. P. Heuring, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, 1992.

[48] M. Grechanik, D. S. Batory, and D. E. Perry. Design of large-scale polylingual systems. In *ICSE*, pages 357–366, 2004.

[49] M. Harren, M. Raghavachari, O. Shmueli, M. G. Burke, R. Bordawekar, I. Pechtchanski, and V. Sarkar. Xj: facilitating xml processing in java. In *WWW*, pages 278–287, 2005.

[50] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *SPIN*, pages 235–239, 2003.

[51] http://www.metalex.nl/pages/welcome.html. *Metalex*, 2002.

[52] http://www.papinet.org. *papiNet*, 2002.

[53] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *ISSTA*, pages 14–25, 2000.

[54] A. Kaplan, J. Bubba, and J. C. Wileden. The exu approach to safe, transparent and lightweight interoperability. In *COMPSAC*, pages 393–394, 2001.

[55] A. Kaplan and J. C. Wileden. Toward painless polylingual persistence. In *POS*, pages 11–22, 1996.

[56] U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994.

[57] J. C. King. A program verifier. In *IFIP Congress (1)*, pages 234–249, 1971.

[58] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[59] C. Kirkegaard, A. Møller, and M. I. Schwartzbach. Static analysis of xml transformations in java. *IEEE Trans. Software Eng.*, 30(3):181–192, 2004.

[60] L. Lamport, R. E. Shostak, and M. C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[61] D. Lee, M. Mani, F. Chiu, and W. W. Chu. NeT & CoT: Inferring XML schemas from relational world. In *ICDE*, page 267, 2002.

[62] S. Liang. *Java Native Interface: Programmer's Guide and Specification.* Addison Wesley, June 1999.

[63] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.

[64] K. J. Lieberherr, B. Patt-Shamir, and D. Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Trans. Program. Lang. Syst.*, 26(2):370–412, 2004.

[65] D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.

[66] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI*, pages 48–61, 2005.

[67] J. Meier, S. Vasireddy, A. Babbar, and A. Mackman. Improving .NET application performance and scalability. *Microsoft Corporation*, 2004.

[68] E. Meijer and W. Schulte. Xml types for c#. In *BillG ThinkWeek Submission*, 2001.

[69] E. Meijer and W. Schulte. Unifying tables, objects, and documents. In *Proceedings Declarative Programming in the Context of Object-Oriented Languages (DP-COOL)*, 2003.

[70] D. Orleans and K. J. Lieberherr. DJ: Dynamic Adaptive Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, September 2001. Springer Verlag. 8 pages.

[71] D. E. Perry. Software architecture and its relevance for software engineering. In *Coordination*, 1997.

[72] D. E. Perry and A. A. Wolf. Foundations for the study of software architectures. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[73] B. C. Pierce. *Types and Programming Languages.* The MIT Press, 2002.

[74] R. Schmelzer. Breaking XML to optimize performance. Z*ap*T*hink* LLC - *special to SearchWebServices.com*, Oct. 2002.

[75] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 974–985, Hong Kong, China, August 2002.

[76] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, Amsterdam, The Netherlands, April 2001.

[77] D. Spinellis. A critique of the Windows application programming interface. *Computer Standards & Interfaces*, 20(1):1–8, Nov. 1998.

[78] D. Suwimonteerabuth, S. Schwoon, and J. Esparza. jMoped: A java bytecode checker based on Moped. In *TACAS*, pages 541–545, 2005.

[79] M. Taghdiri. Inferring specifications to detect errors in code. In *ASE*, pages 144–153, 2004.

[80] J. C. Wileden and A. Kaplan. Software interoperability: Principles and practice. In *ICSE*, pages 675–676, 1999.

[81] J. C. Wileden, J. Ridgway, and A. Kaplan. Why idls are not ideal. In *9th International Workshop on Software Specification and Design*, pages 2–11, 1998.

[82] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227, 1999.

Vita

Mark Grechanik was born in Zhitomir, then USSR (i.e., the Evil Empire), now Ukraine. After receiving his diploma in Electrical Engineering from the National Technical University of Ukraine (former Kiev Polytechnic Institute), he earned a Master of Science in Computer Science from the University of Texas at San Antonio. Concurrently with his academic activities, Mark has been working as a Software Consultant for various high-tech companies. Since 2004, Mark has also been an Adjunct Professor of Computer Science at Texas State University at San Marcos and at the University of Texas at San Antonio. In his spare time Mark likes to cook, play with cats, write books, and make money. Mark is married to Tina Grechanik since 1991.

Permanent Address: 3600 North Hills Dr Apt 207

Austin, TX 78731

This dissertation was typeset with LaTeX 2_ε by the author.