The Dissertation Committee for Byeongcheol Lee
certifies that this is the approved version of the following dissertation:

Language and Tool Support for Multilingual Programs

Committee:

Kathryn S. McKinley, Supervisor

William R. Cook

Robert Grimm

Martin Hirzel

Miryung Kim

Calvin Lin

Language and Tool Support for Multilingual Programs

by

Byeongcheol Lee, B.E., B.E., M.A.

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

August 2011

Acknowledgments

I would like to express my sincere gratitude to Kathryn McKinley for supporting and mentoring me. Kathryn has listened carefully to my research ideas and positively guided my progress. She has given me valuable feedback and encouragement so that I have made the right decisions at critical times. As a result, I have grown as a scholar over the course of my doctoral program, and I am confident to pursue an academic career.

Martin Hirzel and Robert Grimm have been enthusiastic supporters, sharing their research skills. Calvin Lin, William Cook, and Miryung Kim read long documents, attended my talks, and gave valuable feedback. I am thankful for guidance and help from department administrators: Lindy Aleshire, Lydia Griffith, and Gem Naivar. I thank Eileen McGinnis for proofreading this long dissertation.

Living and studying in Austin has been a great pleasure due to many kind people in the department: Michael Bond, Katherine Coons, Taewon Cho, Taehwan Choi, Curtis Dunham, Boris Grot, Sung Ju Hwang, Jungwoo Ha, Dong Li, Ivan Jibaja, Maria Jump, Chang Hwan Peter Kim, Doo Soon Kim, Jaechul Kim, Jongwook Kim, Joohyun Kim, Sangman Kim, Milind Kulkarni, Gene Moo Lee, Juhyun Lee, Sangmin Lee, Bert Maher, Na Meng, Donald Nguyen, Dimitris Prountzos, Donghyuk Shin, Han Hee Song, Jennifer Sartor, Sooel Son, Suriya Subramaniam, Xin Sui, and Ben Wiedermann. They have given me both technical and personal support.

Samsung Foundation of Culture generously provided me with a fouryear fellowship for my Ph.D. I am also grateful to the individuals at IBM Research who invited me for collaborative research and funded me with an internship.

Lastly and most importantly I would like to thank my parents as well as my brother and sister for their selfless love and support.

Language and Tool Support for Multilingual Programs

Publication No. ____________________

Byeongcheol Lee, Ph.D.
The University of Texas at Austin, 2011

Supervisor: Kathryn S. McKinley

Programmers compose programs in multiple languages to combine the advantages of innovations in new high-level programming languages with decades of engineering effort in legacy libraries and systems. For language interoperation, language designers provide two classes of multilingual programming interfaces: (1) foreign function interfaces and (2) code generation interfaces. These interfaces embody the semantic mismatch for developers and multilingual systems builders. Their programming rules are difficult or impossible to verify. As a direct consequence, multilingual programs are full of bugs at interface boundaries, and debuggers cannot assist developers across these lines.

This dissertation shows how to use *composition* of single language systems and *interposition* to improve the safety of multilingual programs. Our compositional approach is scalable by construction because it does not require any changes to single-language systems, and it leverages their engineering efforts. We show it is effective by composing a variety of multilingual tools that help programmers eliminate bugs. We present the first concise taxonomy and formal description of multilingual programming interfaces and their programming rules. We next compose three classes of multilingual tools: (1) **Dynamic**

bug checkers for foreign function interfaces. We demonstrate a new approach for automatically generating a dynamic bug checker by interposing on foreign function interfaces, and we show that it finds bugs in real-world applications including Eclipse, Subversion, and Java Gnome. (2) **Multilingual debuggers for foreign function interfaces**. We introduce an intermediate agent that wraps all the methods and functions at language boundaries. This intermediate agent is sufficient to build all the essential debugging features used in single-language debuggers. (3) **Safe macros for code generation interfaces**. We design a safe macro language, called *Marco*, that generates programs in any language and demonstrate it by implementing checkers for SQL and C++ generators. To check the correctness of the generated programs, Marco queries single-language compilers and interpreters through code generation interfaces. Using their error messages, *Marco* points out the errors in program generators.

In summary, this dissertation presents the first concise taxonomy and formal specification of multilingual interfaces and, based on this taxonomy, shows how to compose multilingual tools to improve safety in multilingual programs. Our results show that our compositional approach is scalable and effective for improving safety in real-world multilingual programs.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The hypothesis of this dissertation is that multilingual programming tools can be built with relatively low effort by combining single-language tools. We first explains the need for and value of designing multilingual tools. We next introduce our taxonomy and formal specification of multilingual interfaces, and our approach for composing their tools. We conclude with suggestion for how this dissertation could affect language designers and practitioners.

1.1 Multilingual Programs

Programmers compose programs in multiple languages to combine the advantages of innovations in new high-level programming languages with decades of engineering effort in legacy libraries and systems. For instance, Java designers delegate low-level operations, such as OS system calls and hardware accesses, to low-level native C code. Programmers use multilingual bindings to write their programs in high-level languages such as Java, C#, OCaml, PHP, and JavaScript while consuming routines from libraries written in C/C++. These libraries include legacy code and highly tuned architecture-specific algorithms. Multilingual systems pervade our critical infrastructure. For example, front-end web browsers, middle-tier application servers, and backend web servers all coordinate multiple languages, such as HTML documents, JavaScript programs, and SQL queries.

For language inter-operation, designers provide two classes of multilingual interfaces: (1) foreign function interfaces and (2) code generation interfaces. The difference between them is whether the programs in different languages exchange data (foreign function interface) or code (code generation interface). *Foreign function interfaces* include the Java Native Interface (JNI), Python/C API, and OCaml FFI. These interfaces are public functions exposed to virtual machines and interpreters for purpose of exchanging data. *Code generation interfaces* are stream channels from a program to the compiler or interpreter of a target language. For instance, a Java web program sends SQL queries to the back-end database management system. Using code generation interfaces, a program in one language generates and executes another program in a different language.

These interfaces result in a semantic mismatch that compromises end-to-end safety. For instance, Java native methods must make up for the difference between the automatic memory management policy in Java (e.g., garbage collection) and manual memory management policy in C/C++ (e.g., `malloc` and `free`). Multilingual programming interfaces are very hard to use correctly because they have thousands of programming rules. Some of these rules cannot be statically verified. As a direct consequence, multilingual programs are full of interface bugs [25, 26, 43, 45, 48, 75–77]. Worse, multilingual programming interfaces prevent debuggers from examining code language barriers. For instance, Java debuggers cannot debug C code because it is outside of the Java Native Interface while C debuggers cannot debug Java code because it is inside the Java Native Interface.

In previous approaches for checking and debugging multilingual programs, programmers must learn new language constructs or annotations [8, 35, 74], or tool writers expend significant effort to retarget to additional programming languages [26, 32, 40, 43, 48, 62, 66, 76, 77, 83]. Consequently, these solutions do not scale to multiple languages. In other words, they are either nonexistent, or a single monolithic tool must reason about all languages at once.

1.2 Interposition in Composing Multilingual Systems

The goal of this dissertation is to significantly decrease the complexity of building multilingual tools to improve the safety of multilingual programs. For this purpose, we avoid re-implementing all runtime systems, compilers, interpreters, and debuggers for single programming languages. Instead, we interpose on multilingual programming interfaces in order to compose multilingual tools. For instance, our multilingual tools wrap Java native methods and JNI functions. This compositional approach scales well to many languages and their interfaces for two reasons. First, it does not require any change to single-language tools. Second, it leverages all the engineering effort that has been applied to develop single-language tools.

Given this interposition principle, we show that it can be effective to compose multilingual tools. This dissertation provides the first principled approach for describing and reasoning about key programming language semantic elements, between which multilingual tools must communicate and translate. We first formalize multilingual programming interfaces (Chapter 2). With this specification, we show how to design and compose three classes of multilingual tools: dynamic bug checkers for foreign function interfaces (Chapter 3), multilingual debuggers for foreign function interfaces (Chapter 4), and macro systems for code generation interfaces (Chapter 5).

Multilingual Programming Interfaces. Chapter 2 describes two dominant multilingual interfaces: foreign functions and code generation. Using foreign function interfaces, programs in different languages exchange code and data at the granularity of the functions and methods. Participating programs must respect different programming language semantics and translate between thread states, types, and resources. Using code generation interfaces, a program in one language manufactures a program in another language. Then, it delegates the execution of the program to compilers and interpreters. The generating program must respect syntax, scope, and types in the target language. We introduce the first formal specification and taxonomy of multilingual interfaces. The specification eases the multilingual tool developer's work and helps users by simplifying 1,500+ rules to a dozen small state machines.

These multilingual programming interfaces provide a foundation for the principle of interposition. These interfaces define clearly boundaries among different languages. To interpose foreign function interfaces, we wrap functions and methods. To interpose code generation interfaces, we execute the functions and methods of the compilers and interpreters that receive programs.

Dynamic Bug Detection for Foreign Function Interfaces. Chapter 3 shows how to design and build a dynamic bug checking tool. Our dynamic bug checkers wrap all the methods and functions at language boundaries. They keep track the data values exchanged in language transitions. This transition history is summarized as a collection of our state machines that change their states due to language transition events. The error states in the state machines indicate that the program violates FFI constraints.

Our dynamic bug detectors for Java and Python interpose on language transitions and do not require any changes in the multilingual programs and single language systems. Dynamic bug checkers add only what is necessary at the language boundary. Our Java tool detects bugs in real-world applications including Eclipse, Subversion, and Java Gnome. A large fraction of the bugs we reported were confirmed and fixed.

Multilingual Debuggers for Foreign Function Interfaces. Chapter 4 explores the design and implementation of composable multilingual debuggers. Our intermediate agent wraps all the methods and functions at language boundaries. Then, it solves all the implementation problems in composing multilingual debuggers out of single language debuggers. For instance, programmers cannot run any `jdb` queries when `gdb` suspends the debugee. To activate `jdb` from a C/C++ breakpoint, we ask `gdb` to activate a Java breakpoint in the agent. This `gdb` command wakes up `jdb` transparently without relying on how `gdb` and `jdb` are implemented. We show that this intermediate agent is sufficient to compose all the essential debugging features from multiple debuggers.

Our composition method does not require any changes in single language debuggers, but it implements only what is necessary between single language debuggers. We demonstrate this approach by building debuggers for Java and C, and Jeannie [35]. The result is a powerful debugger that controls multiple programming languages.

Macro Systems for Code Generation Interfaces. Chapter 5 illustrates that our interposition principle extends to syntax checkers for code generation interfaces. To improve safety in these interfaces, we designed a macro language, Marco, that generates programs in many languages and demonstrate it by implementing checkers for SQL and C++ generators. To check the wellformeness of the generated programs, we devise an oracle query system based on code generation interfaces. Our Marco system sends query programs to these interfaces. Based on the error messages, it statically and dynamically points out the errors in the Marco programs.

Our Marco system practices the principle of interposition in code generation interfaces. It does not require any change in target language systems, including C++ compilers and relational database management systems. Instead, it introduces an oracle query analysis framework that takes as plug-in components the compilers and interpreters that recognize syntactic, semantic, scope rules in the target languages. The analysis framework queries the compilers and interpreters about the generated programs by sending program fragments and receiving the error messages. The result is an error checking system for macros that scales to many languages.

1.3 Contributions

This dissertation is the first to show how to compose multilingual tools. The contributions include:

1. *Taxonomy and Specification of FFI Constraints into State Machines.* We show FFI constraints are derived from language semantic mismatch in thread states, types, and resources. We encode class of constraints as a few state machines where language transitions change states and an error state indicates violation of an FFI constraint. Based on our insight on how to map from FFI constraints and language transitions to state machines and state transitions, we synthesize dynamic FFI bug detectors that only interpose on language transitions.

2. We design and build *Jinn*, a dynamic bug detector for Java Native Interface (JNI). We generate it using a new process for synthesizing dynamic analysis for foreign interfaces at language boundaries. Jinn runs on stock JVMs, checks all the JNI programming rules, and finds serious JNI bugs in a real-world applications.

3. *Composition of Mixed-Environment Debuggers.* We introduce a mechanism for constructing multilingual debuggers that scale to many execution environments. The construction mechanism requires very little change to the runtime environments. It only adds an intermediate agent to the debugee process. Then, it leverages the single-environment debuggers to control all the environments. We show that it scales to a variety of runtime environments, compilers, and single-environment debuggers.

4. The first mixed-environment debugger for Java and C called *Blink*. We used the composition mechanism to build a mixed-environment debugger. We show that Blink implements the de facto standard set of debugging commands in a variety of runtime environments. Prior to Blink, debuggers could not inspect state in multiple languages without a unified runtime at once nor transition across language boundaries.

5. A code generation checker using *oracle query analysis.* We introduce a mechanism for checking well-formedness of fragments in foreign languages that scales to any languages. It improves correctness of the macro programming practice that generates foreign language programs.

6. *Marco*, a macro programming language that expresses macros for multiple programming languages. We implement a code generator checker

that verifiers well-formedness of fragments in Marco programs that generate SQL and C++ using the oracle query analysis.

1.4 Impact

This dissertation seeks influence language designers, tool developers, and programmers. For language designers, our classification work will lay the foundation for documenting and enforcing FFI programming rules. For tool developers, our work will lead them to build composable multilingual tools that assist programmers in debugging their programs. For programmers, multilingual tools will help them to write correct multilingual programs that take advantage of innovations in new programming languages and decades of engineering efforts in legacy libraries.

Chapter 2

The Essence of Multilingual Programming Interfaces

This chapter characterizes multilingual programming interfaces, presents a new taxonomy and specification for reasoning about multilingual interfaces, and explains how to use this taxonomy to generate and build composable tools for multilingual systems. Section 2.1 starts with an example that shows how Eclipse uses the two types of multilingual programming interfaces: (1) foreign function interfaces, and (2) code generation interfaces. Section 5.5.2 describes foreign function interfaces and Section 2.3 presents code generation interfaces. Section 2.4 present a taxonomy of what kinds of rules govern multilingual interfaces and how to map these rules to state machines.

2.1 Motivating Example

Eclipse is an interactive development environment that helps programmers write, edit, and execute their programs. Figure 2.1 presents a snapshot of Eclipse showing a Java method comment in a tool tip. For this single task, Eclipse invokes legacy libraries, including the Mozilla application framework and an SQLite database management system. Eclipse sends the comment as an HTML document to the Mozilla application framework, which renders the HTML document. The Mozilla application framework sends SQL queries to read and write the history of visited HTML documents in an SQLite database.

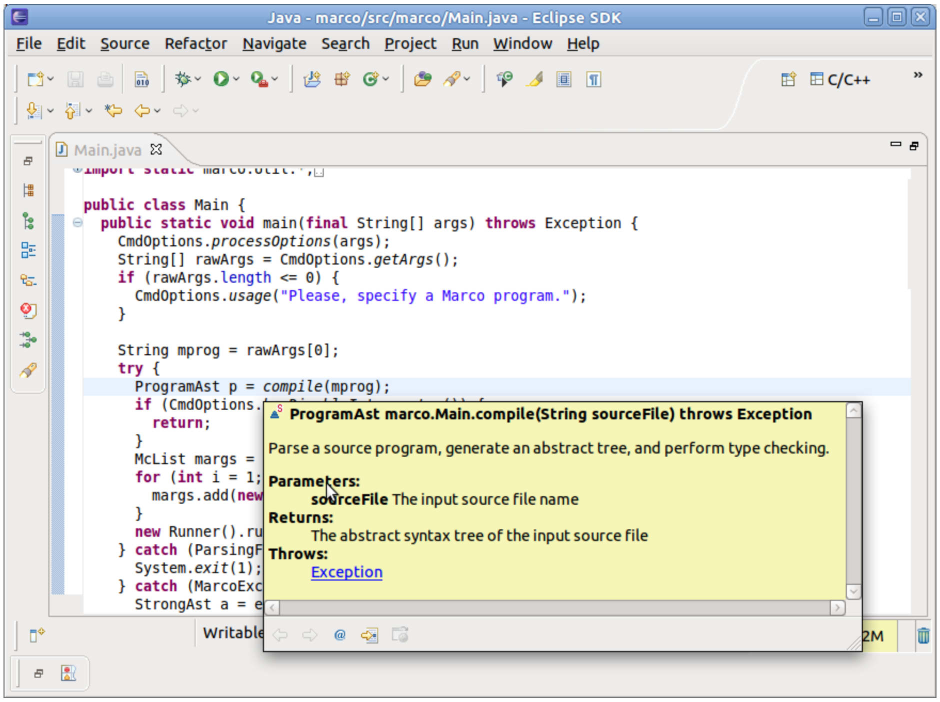


Figure 2.1: A screenshot of the Eclipse 3.5.1 interactive development environment running on Linux. Eclipse benefits from the clean thread model, type safety, and garbage collection in Java. On the other hand, it leverages decades of engineering effort in legacy libraries from the Mozilla application framework and SQLite database management system.

For this single task, the software written in different languages communicate with each other using foreign function interfaces and code generation interfaces.

Figure 2.2 illustrates the Java Native Interface (JNI), a foreign function interface, in Standard Widget Toolkit (SWT). The native modifier claims that the vtblCall method has its definition in native code. The body of that method is empty. The VtlbCall function in C defines the implementation of the native method. The C code dynamically invokes a C function using the pointer value

```
  33. public class XPCOM extends C {
 546.     static final native int _VtblCall(int fnNumber, int /*long*/ ppVtbl);
2267. }
9766. jint Java_org_eclipse_swt_internal_mozilla_XPCOM__1VtblCall__II
          ( JNIEnv *env, jclass that, jint arg0, jint arg1 )
9771. {
9778.     rc = (jint)((jint ( *)(jint))(*(jint **)arg1)[arg0])(arg1);
9784.     return rc;
9785. }
```

Figure 2.2: A Java native method in Eclipse SWT 3.5.1 that executes a routine in the Mozilla application framework using a foreign function interface. The first three lines are from XPCOM.java. The last six source lines are from xpcom.cpp. after C++ preprocessing.

from Java code. The code illustrates a foreign function interface between Java and C++.

Figure 2.3 shows a C++ method in the Mozilla framework that generates an SQL query and sends it to an SQLite database. The C++ method prepares for an SQL program and keeps it in the local variable, query. Then, it sends the query to the database management system. This C++ method illustrates code generation interface between C++ and SQL.

These multilingual programming interfaces are used pervasively in a single program. They have many interface constraints that programmers must manually ensure. Sections 5.5.2 and 2.3 respectively discuss these constraints for foreign function interfaces and code generation interfaces.

2.2 Foreign Function Interfaces

Foreign function interfaces (FFIs) consist of declarations and functions [49, 81]. For instance, JNI programmers add a native modifier to the

```
nsresult
Connection::databaseElementExists(enum DatabaseElementType aElementType,
   const nsACString &aElementName,
   PRBool *_exists)
{
   nsCAutoString query(SELECT name FROM sqlite_master WHERE type = ');
   switch (aElementType) {
     case INDEX:
       query.Append(index);
       break;
     case TABLE:
      query.Append(table);
      break;
   }
   query.Append(' AND name =');
   query.Append(aElementName);
   query.Append(');

   sqlite3_stmt *stmt;
   int srv = ::sqlite3_prepare_v2(mDBConn, query.get(), -1,&stmt, NULL);
...
}
```

Figure 2.3: A C++ method in the Mozilla Application Framework 1.9.2 that sends an SQL query to the SQLite database management using a code generation interface from mozStorageConnection.cpp.

declaration of a Java native method. In Java, the Java native method is empty, and its implementation is written in C or C++. The C or C++ body calls several JNI functions. This type of foreign function interface is used by high-level programming languages including JNI [49], Python/C [81] and Ocaml/C [47].

Each FFI has many programming rules that programmers must respect at the foreign function interface level. For instance, programmers must ensure 1,500+ rules before calling 229 JNI functions. These rules are loosely specified in the Java Native Interface book [49].

This section describes how three classes of constraints summarize all JNI rules and how to encode these constraints in eleven state machines. We argue for the generality of this approach based on our analysis of Python/C. As far as we are aware, no previous work specifies the JNI and other FFIs formally nor observe or exploits the required mapping of language syntax and semantic elements between languages.

We observe that the JNI constraints fall into three classes: (1) Thread state constraints ensure that the JVM thread is in an expected state before calls from C. (2) Type constraints ensure that C passes valid arguments to Java. (3) Resource constraints ensure that C code manages JNI resources correctly. Table 2.1 summarizes these constraints and indicates the number of times a JNI routine requires each constraint type. For example, the "JNIEnv* state" constraint appears 229 times, because all 229 JNI functions requires them as preconditions for their execution.

We now fully specify the JNI rules. Chapter 3 shows example state machines for all these rules. However, this taxonomy is general and captures the essence of how languages differ. (1) How they handle exceptions and their

Constraint	Count	Description
	Thread state constraints	
JNIEnv* state	229	Current thread matches JNIEnv* thread
Exception state	209	No exception pending for sensitive call
Critical-section state	225	No critical section
	Type constraints	
Fixed typing	157	Parameter matches API function signature
Entity-specific typing	131	Parameter matches Java entity signature
Access control	18	Written field is non-final
Nullness	416	Parameter is not null
	Resource constraints	
Pinned or copied string or array	12	No leak or double-free
Monitor	1	No leak
Global or weak global reference	247	No leak or dangling reference
Local reference	284	No overflow or dangling reference

Table 2.1: Classification and number of JNI constraints.

threading model. For example, returning to the calling thread and locking disciplines (2) Differences between their type systems, calling conventions, and rules about values. (3) Difference between explicitly managed and garbage collected memory, addresses, and local versus global references.

2.2.1 Thread State Constraints

To enter the JVM through any JNI function, C code must satisfy three conditions: (1) The JNI environment pointer JNIEnv* and the caller belong to the same thread. (2) Either no exception is pending, or the callee is exception-oblivious. (3) Either no critical region is active, or the callee is critical-region oblivious.

JNIEnv* state constraint. All calls from Java to C implicitly pass a pointer to the JNIEnv structure, which specifies the JVM-internal and thread-local state. All calls from C to Java must explicitly pass the current pointer when invoking a JNI function.

Exception state constraints. When Java code throws an exception and returns to C, the C code does not automatically transfer control to the nearest exception handler. The program must explicitly consume or propagate the pending exception. This constraint results from the semantic mismatch in how C and Java handle exceptions. Any JNI call may lead to Java code that throws an exception, which causes a transition to the "exception pending" state when the JNI call returns.

Critical-section state constraints. JNI defines the phrase "JNI critical section" to describe a piece of C code that has direct access to a Java string or array, during which the JVM may take drastic measures such as disabling the garbage collector. To provide safe access, a critical section starts with GetStringCritical or GetPrimitiveArrayCritical and ends with the matching ReleaseStringCritical or ReleasePrimitiveArrayCritical. C code should hold these resources only for a short time. To prevent deadlock, C code must not interact with the JVM other than to acquire or release critical resources. In other words, during a critical section, C code must only call one of the four functions that get/release arrays/strings. We call these four functions critical-section *insensitive* and all the remaining JNI functions critical-section *sensitive*.

2.2.2 Type Constraints

When Java code calls a Java method, the compiler and JVM check type constraints on the parameters. However, when C code calls a Java method, the compiler and JVM do not check type constraints, and type violations cause unspecified JVM behavior. For example, given the Java code

```
Collections.sort(ls, cmp);
```

the Java compiler checks that class Collections has a static method sort and that the actual parameters ls and cmp conform to the formal parameters of sort. Consider the equivalent code expressed with Java reflection:

```
Class clazz = Collections.class;
Method method =
  clazz.getMethod("sort", List.class, Comparator.class);
method.invoke(Collections.class, ls, cmp);
```

The Java compiler cannot statically verify its safety, but if the program is unsafe at runtime, then the JVM throws an exception. In JNI, this code is expressed as follows.

```
jclass clazz = (*env)->FindClass(env, "java/util/Collections");
jmethodID method = (*env)->GetStaticMethodID(env, clazz,
   "sort", "(Ljava/lang/List;Ljava/util/Comparator;)V");
(*env)->CallStaticVoidMethod(env, clazz, method, ls, cmp);
```

Since the C code expresses Java type information in strings, standard static type checking cannot resolve the types, and even sophisticated interprocedural analysis cannot always resolve them [25, 77]. Consequently, the C compiler does not statically enforce typing constraints on the "Collections" and "sort" names or the ls and cmp parameters. Furthermore, and unlike Java reflection, JNI does not even dynamically enforce typing constraints on the clazz and method descriptors. This interface is a potential source of inadvertent errors. Furthermore, malicious C code can abuse it, breaking the Java safety guarantees.

Fixed typing constraints. Type constraints require the runtime type of the actuals to conform to the formals. For many JNI functions, the parameter type is, in fact, *fixed* by the function itself. For example, in CallStaticVoidMethod(env, clazz, method, ls, cmp), the clazz actual must always conform to type java.lang.Class.

Entity-specific typing constraints. A plethora of JNI functions call Java methods or access Java fields. JNI references Java methods and fields via *entity IDs*. For example, in CallStaticVoidMethod(env, clazz, method, ls, cmp), parameter method is a method ID. In this case, the method must be static, and the method parameter constrains the other parameters. In particular, the clazz must declare the method, and ls and cmp must conform to the formal parameters of the method.

Access control constraints. Even when type constraints are satisfied, Java semantics may prohibit accesses based on visibility and final modifiers. For example, in SetStaticIntField(env, clazz, fid, 42), the field identified by fid may be private or final, in which case the assignment follows questionable coding practices. The JNI specification is vague on legal accesses with respect to their visibility and final constraints. After some investigation, we found that in practice, JNI usually ignores visibility, but honors the final modifier. Ignoring visibility rules seems surprising, but as it turns out, this permissiveness is consistent with the behavior of reflection, which may suppress Java access control when setAccessible(true) was successful. Honoring final is common sense. Despite the fact that reflection may mutate final fields, mutating them interferes with JIT optimizations, concurrency, and the Java memory model.

Nullness constraints. Some JNI function parameters must not be null. For example, in CallStaticVoidMethod(env, clazz, method, ls, cmp), the parameters env, clazz, and method must not be null. At the same time, some JNI functions do accept null parameters. For example, the initial array elements in NewObjectArray. Since the JNI specification is not always clear on which parameters may be null, we determined these constraints experimentally. We uncovered 416 non-null constraints among the 210 JNI functions that define parameters.

2.2.3 Resource Constraints

A JNI resource is a piece of Java-related data that C code can acquire or release through JNI calls. For example, C code can acquire a Java string or array. Depending on the JVM implementation, the JVM either pins the string or array to prevent the garbage collector from moving it, or copies the array, and then passes C code a pointer to the contents. Other JNI resources include various kinds of opaque references to Java objects, which C code can pass to JNI functions and which give C code some control over Java memory management. Finally, JNI can acquire or release Java monitors, which are a mutual-exclusion primitive for multi-threaded code.

APIs with manual or semi-automatic memory management suffer from well-known problems: (1) Section 3.1.2 illustrates one such problem: a use after a release corrupts JVM state through a dangling reference. There are three other common resource errors. (2) An acquire at insufficient capacity causes an overflow. (3) A missing release at the end of reference lifetime causes a leak. (4) A second release is a double-free.

Pinned or copied string or array constraints. C code can temporarily obtain direct access to the contents of a Java string or array. JVMs may pin or copy the object to facilitate garbage collection. To make sure the JVM unpins the object or frees the copy, the C code must properly pair acquire/release calls to avoid dangling references and leaks.

Monitor constraints. A monitor is a Java mutual exclusion primitive. A monitor of a Java must be acquired before calling the wait method on the object. JVMs checks this rule by throwing an IllegalMonitorStateException exception. After being acquired, it must be released eventually to avoid a deadlock. The C code must properly call the MonitorExit function to release the monitor.

Global reference or weak global reference constraints. A global or weak global reference is an opaque pointer from C to a Java object that is valid across JNI calls and threads. These references are explicitly managed because the garbage collector needs to update them when moving objects and also treat global (but not weak) references as root. The C code must properly pair acquire/release calls to avoid dangling references and leaks.

Local reference constraints. JNI manages local references semi-automatically: acquire and release are more often implicit than explicit. Native code implicitly *acquires* a local reference when a Java native call passes it to C or when a JNI function returns it. The JVM *releases* local references automatically when native code returns to Java, but the user can also manually release one (DeleteLocalRef) or several (PopLocalFrame) local references.

2.2.4 Generality

We examined a number of language interfaces and Python/C in depth (cf. Section 3.4). We found this taxonomy captures the interfaces between languages.

2.3 Code Generation Interfaces

The other most widely used multilingual interface is code generation, in which a program generates another program as a string. The runtime then must translate to the target language. The target-language system parses, analyze, and executes the program. For instance, a middle-end web program generates SQL queries as Java strings. These queries are sent through Java Database Connectivity (JDBC) to a database management system. In the SPL system, primitive operators generate C++ compilation units. This code generation practice works when the host programming language has a string type and the target programming language may be parsed as a sequence of characters. Since these characteristics are ubiquitous, many multilingual programs use code generation interfaces.

Code generation interfaces are bidirectional. A host-language program sends a program to a target-language processing system. The target-language processing system accepts or rejects the program. In the case of rejection, it might explain the reasons. For instance, a Java program sends an SQL query to a database management system through JDBC connectivity. If the query is acceptable, the database management system returns the Java objects representing a relational table. If the query is unacceptable, the JDBC driver throws a Java exception containing error messages.

While code generation interfaces are flexible, programmers must manually check the correctness of generated code. The correctness criteria are divided into constraints in three areas: syntax, scope, and semantics. These constraints are what the target language processors check internally. For instance, a SQL processor will parse a sequence of characters for syntactic constraints. Once successful, it will apply scope rules to map uses of identifiers to their definitions. Then, it will type check expressions and statements.

The key difficulty lies in that programming language designers make quite diverse decisions in syntax, scope, and semantics. This section examines two example constraints: C++ for depth and SQL for breath.

2.3.1 Syntactic Constraints

Syntactic constraints are specified as context free grammars. For instance, programing language books devote chapters and sections to describing grammars [27, 39, 44, 69]. We characterize these grammars in syntactic richness and ambiguity. For instance, consider LISP and C++. A LISP grammar has a few nonterminals, and its parsers do not backtrack. On the other hand, the C++ grammar contains hundreds of nonterminals, and its parsers frequently backtracks.

2.3.2 Scope Constraints

Scope constraints are enforced during the code generation process. Even if generated code is syntactically correct, the resulting code could be quite counter intuitive. For instance, consider the following C macro that swaps values in two integer variables:

```
#define SWAP(x,y) { int tmp = x; x = y; y = tmp;}
int main() {
  int tmp=1,b=2;
  SWAP(tmp, b)
  printf("tmp =%d and b = %d\n", tmp);
}
```

A C preprocess generates the following program:

```
int main() {
  int tmp=1,b=2;
  { int tmp = tmp; tmp = b; b = tmp;}
  printf("tmp =%d and b = %d\n", tmp);
}
```

C compilers accept the expanded program without any syntax error, but the compiled program produces the undefined value for the `tmp` variable. The `SWAP` macro failed to ensure the hygienic code generation constraint where a local variable in a macro body must not capture a free variable in macro parameters.

2.3.3 Semantics Constraints

The generated code must respect type constraints in statically typed target languages. Statically typed programming language specifications define their own typing rules [27, 39, 44, 69]. We leave classification and analysis of semantic constraints as future work, and address syntactic constraints here.

2.4 Taxonomy of Multilingual Systems

Table 2.2 presents how and which programming systems help programmers to specify, enforce, and check multilingual programming rules. The environment column shows the interface type. Multilingual programming interfaces must adhere to all six classes of interface constraints in the second column. Some program specific constraints have nothing to do with these interfaces but require programmers to reason about control and data flow in multiple environments at once. In this case, multilingual programs complicate the programmer's task by requiring them to be fluent in multiple languages to correctly implement their code. Programming systems in Columns 3-5 check constraints at various stages. Language design approaches are the most powerful, but they do not support legacy multilingual programs. Static analyses verify many legacy programs, but they are not complete and sound in general for undecidable constraints. Dynamic approaches complement static analysis since they can be designed not to report false alarms.

Each entry in Table 2.2 cites prior work or refers to work in this thesis. All our solutions focus on composable solutions that reuse existing runtime systems, languages, compilers, and interpreters as much as possible. For foreign function interfaces, Chapter 3 presents dynamic bug finders that scale

Environment	**Constraint class**	**Language design**	**Static analysis**	**Dynamic analysis**
	Thread state	[35],[74]	[48][43]	Chapter 3
Foreign function interface	Type	[35],[74]	[26]	Chapter 3
	Resource	[35]	[43]	Chapter 3, [66]
	Syntax	Chapter 5,[88]	Chapter 5	
Code generation interface	Scope	Chapter 5,[88]	Chapter 5	Chapter 5
	Semantics	[88]	[77]	
Cross lingual		[28]	[76]	Chapter 4

Table 2.2: Taxonomy of multilingual programming rules including how and which tools check them.

to J9, HotSpot, Java, C/C++, and Python. For code generation interfaces, Chapter 5 introduces a programming language, Marco, that is specialized to the task of generating programs in SQL and C++. The Marco language includes both static analysis and dynamic analysis. For the classes of bugs that involve inter-language flow of data and control, we propose an approach for composing mixed-environment debuggers that scales to J9, HotSpot, gcc, and Microsoft C++ compiler in Chapter 4.

Chapter 6 discusses related work in detail, but the remainder of this Chapter gives a flavor of some alternatives to FFIs, FFI specification, and dynamic verification.

Language Approaches to FFI Safety. Two language designs propose to replace the JNI. SafeJNI [74] combines Java with CCured [57], and Jeannie safely and directly nests Java and C code into each other using quasi-quoting [35]. Both SafeJNI and Jeannie define their language semantics such that static checks catch many errors and both add dynamic checks in translated code for other errors. From a purist's perspective, preventing FFI bugs while writing code is more economical than spending time to fix them after the fact. Another approach generates language bindings for annotated C and C++ header files [8, 38]. Ravitch et al. reduce the annotations required for generating idiomatic bindings [62]. our FFI tools are more practical than these approaches because they do not require developers to rewrite or annotate their code in a different language.

Static FFI Bug Checkers. A variety of static analyses verify existing foreign function interfaces [25, 26, 43, 48, 75, 76]. All static FFI analysis approaches suffer from false positives because the specification includes dynamic properties, such as non-null reference parameters, valid Java class and method names in string parameters, and less than 16 local references. Static analysis cannot typically guarantee these properties. For instance, J-Saffire reports false positives and warnings [26]; Tan et al. report a false positive rate of 15.4% [48]; and BEAM reports a false positive while missing the actual bug in a program reported in Section 3.1.1.

Dynamic FFI Bug Checkers. Some JVMs provide built-in dynamic JNI bug checkers, enabled by the `-Xcheck:jni` command-line flag. While convenient, these error checkers only cover limited classes of bugs, and JVMs implement them inconsistently. NaturalBridge's BulletTrain ahead-of-time Java compiler performed several ad-hoc JNI integrity checks on language transitions [56]. Our Blink debugger provides JNI bug checkers that work consistently for different JVMs, but its coverage is limited to two bugs: validating exception state and nullness constraints [45]. These kinds of checks are easy to implement because they require no bookkeeping.

State Machine Specifications. Several programmable bug checkers take state machine specifications and report errors when state machines reach error states. For instance, Metal [21] and SLIC [7] are languages for specifying state machines that are then used to find bugs through static analysis. Dwyer et al. survey state-machine driven static analyses [20]. On the dynamic side, Allan et al. turn FSMs into dynamic analyses by using aspect-oriented programming [1]; Chen and Rosu synthesize dynamic analyses from a variety of specification formalisms, including FSMs [14]; and Arnold et al. implement FSMs for bug detection in a JVM, controlling the runtime overhead by sampling [2]. While in principle these specification languages are expressive enough to describe many FFI constraints, in practice none of them address the unique challenges of multilingual software.

Chapter 3

Automatically Finding Bugs at Foreign Language Interfaces

Many multilingual programs are composed of libraries or frameworks written in a variety of programming languages, communicating through foreign function interfaces. Foreign function interfaces typically consist of several hundred of functions and thousands of programming rules. It is tedious and error prone to ensure that these multilingual programs respect all the programming rules. This chapter shows how to synthesize dynamic bug detectors that detect, report, and stop the erroneous multilingual programs that break these programming rules. Our dynamic bug detectors find a class of bugs that other bug finders ignore. Furthermore, they detect dozens of bugs in several real-world applications. Most of these bugs are confirmed and fixed. The time overhead of our dynamic analysis is 14%. These results suggest that our dynamic bug finders are applicable to realistic development environments.

We start by presenting a motivating example that shows how our dynamic analysis detects a bug that breaks an FFI constraint in Section 3.1. Section 3.2 illustrates how to synthesize dynamic bug finders that completely check all the interface constraints from state machine specifications for the JNI. Section 3.4 demonstrates that our approach generalizes to the FFI for Python/C. We evaluate our bug detectors in Section 3.5.

3.1 An Example JNI Bug and Detector

This section illustrates and motivates our approach using an example. It provides some additional JNI background, an example JNI bug, and a state machine that captures this bug. It then describes how to use this state machine to dynamically detect the example bug on language transition boundaries at JNI calls and returns.

The JNI is designed to hide JVM implementation details from native code while also supporting high-performance native code. Hiding JVM details from C code makes multilingual Java and C programs portable across JVMs and gives JVM vendors flexibility in memory layout and optimizations. However, achieving portability together with high performance leads to 229 API functions and 1,500+ usage rules. For instance, JNI has functions for calling Java methods, accessing fields of Java objects, and obtaining a pointer into a Java array as described in the Java Native Interface book [49]. To hide JVM implementation details, these functions go through an indirection, such as method and field IDs, or require the garbage collector to pin arrays. Developers using JNI avoid indirection overhead on the C side by, for example, caching method and field IDs, and pinning resources. At the same time, JVM developers avoid implementation complexity by requiring explicit calls to mark references as global and to release pinned objects.

3.1.1 Example FFI Bug

Figure 3.1 shows a simplified version of an FFI bug from the GNOME project's Bugzilla database (Bug 576111) [78]. GNOME is a graphical user interface that makes heavy use of several C libraries. In the example, Line 1 defines a C function `Java_Callback_bind` that implements a Java native method

```
1.  JNIEXPORT void JNICALL Java_Callback_bind(JNIEnv *env,
2.    jclass clazz, jclass receiver, jstring name, jstring desc)
3.  {         /* Register an event call-back to a Java listener. */
4.    EventCallBack* cb = create_event_callback();
5.    cb->handler = callback;
6.    cb->receiver = receiver;  /* receiver is a local reference.*/
7.    cb->mid = find_java_method(env, receiver, name, desc);
8.    if (cb->mid != NULL) register_callback(cb);
9.    else destroy_callback(cb);
10. }                              /* receiver is a dead reference. */
11. static void callback(EventCallBack* cb, Event* event) {
12.   JNIEnv* env = find_env_pointer_from_current_thread();
13.   jvalue* jargs = marshal_event(cb, env, event);
14.          /* BUG: dereference of now invalid cb->receiver. */
15.   (*env)->CallStaticVoidMethodA(
16.     env, cb->receiver, cb->mid, jargs);
17. }
```

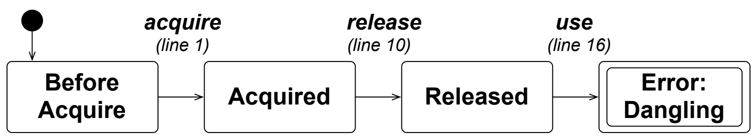
Figure 3.1: JNI invalid local reference error in a call-back routine from GNOME (Bug 576111) [78].

using the JNI. An example *call from Java to C* takes the following form:

```
Callback.bind(receiverClass, "methodName", "description");
```

This call invokes the C function `Java_Callback_bind`, which registers a new C heap object `cb`, storing the receiver class and method name passed as parameters from Java. The C function `callback` referenced on Line 5 is defined starting at Line 11. It uses the `cb` parameter object to call from C code to the specified Java method. Line 15 shows this *call from C to Java*. It uses a JNI API function `CallStaticVoidMethodA`, which resides in a `struct` referenced by the JNI environment pointer `env`.

This code is buggy. The parameter `receiver` in Line 2 is a local reference.



State transition	*Language transition*	*Triggering functions*
Acquire	Call:Java→C	Native method taking reference e.g., Java_Callback_bind
	Return:Java→C	JNI function returning reference e.g., GetObjectField
Release	Return:Java→C	DeleteLocalRef
	Return:C→Java	Any native method e.g., Java_Callback_bind
Use	Call:C→Java	JNI function taking reference e.g., CallStaticVoidMethodA
	Return:C→Java	Native method returning reference e.g., Class.getClassContext

Figure 3.2: A resource tracking state machine for local references and the mapping from state transitions to Java and C language transitions (calls and returns) to dynamically detect the bug in Figure 3.1.

A local reference in JNI is only valid until the enclosing function returns, because, otherwise, Java virtual machine's (JVM's) garbage collector would need to communicate with the C runtime about live references. Thus, cb−>receiver becomes invalid when the function returns at Line 10. However, Line 6 stores receiver in a heap object, letting it escape. When Line 16 retrieves receiver from the heap and uses it as a parameter to CallStaticVoidMethodA, it is an invalid dangling reference, and the JVM's garbage collector may have either moved the object or reclaimed it and reused the corresponding memory.

The JNI specification merely says that this reference is invalid and leaves the consequences up to the vendor's Java implementation [49]. This kind of bug is difficult to find with static analysis because it involves complex data flow through the heap as well as complex control flow through disjoint indirect calls and returns across languages. For instance, the syntax analysis in J-BEAM [43] misses this bug.

3.1.2 Example FFI Bug Detector

This section shows how to identify this bug dynamically using a state machine. Figure 3.2 shows a simplified state machine that enforces local usage rules, applied to the receiver parameter at runtime. On entry to the method (Figure 3.1: Line 1), the state of receiver transitions from Before Acquire to Acquired. When the method returns back to Java (Line 10), the state transitions from Acquired to Released. Finally, the call from C to Java at Line 16 uses the reference cb−>receiver, triggering a transition to the Error: Dangling state and thus detecting the bug.

While prior work used state machines to find bugs [1, 2, 7, 14, 20, 21], it was not clear if FFI specifications could be characterized with state machines nor how to map and generate dynamic analysis automatically.

The table in Figure 3.2 shows more generally where state transitions occur. For example, dynamic analysis must execute the Acquire transition for all reference parameters on all calls from Java to C. On return from C to Java, dynamic analysis must execute the Release transition for all local references. To instrument both calls and returns, we wrap these calls. For example, our dynamic checker replaces Java_Callback_bind with the wrapper function wrapped_Java_Callback_bind shown in Figure 3.3. The instrumentation attaches state machines to entities (threads, parameters, and return values) by using thread-local storage (refs).

We also instrument the JNI functions that implement the C API for interacting with the Java virtual machine. For example, the Use transition in the table happens on calls from C to Java if the callee is a JNI function taking a reference, such as CallStaticVoidMethodA. Such a use is an error if the reference is in the Released state. Figure 3.4 shows the wrapper with the instrumentation.

For illustration purposes, these example wrappers omit other checks our system performs. For example, JNI limits the number of available local references, so there is another possible error state for overflow. Developers may manually manage the number of available local references with the JNI functions PushLocalFrame and PopLocalFrame and the corresponding dynamic analysis requires instrumentation to count references. The figures also omit checks for thread state, exception state, and parameter nullness. Section 3.3 explains all the constraints we check and their encoding in state machines.

```
void wrapped_Java_Callback_bind(JNIEnv *env,
  jclass clazz, jclass receiver, jstring name, jstring desc)
{
  /* Instrument Call:Java→C for Acquire state transition. */
  jobject_set refs = jinn_acquire_thread_local_jobject_set();
  if (clazz != NULL) { jinn_refs_acquire(refs, clazz); }
  if (receiver != NULL) { jinn_refs_acquire(refs, receiver); }
  if (name != NULL) { jinn_refs_acquire(refs, name); }
  if (desc != NULL) { jinn_refs_acquire(refs, desc); }
  /* Call the wrapped native method. */
  Java_Callback_bind(env, clazz, receiver, name, desc);
  /* Instr. Return:C→Java for Release state transition. */
  jinn_release_thread_local_jobject_set(refs);
}
```

Figure 3.3: Wrapper for function Java_Callback_bind from Figure 3.1 with instrumentation for Acquire and Release state transitions.

```
void wrapped_CallStaticVoidMethodA(JNIEnv *env,
  jclass clazz, jmethodID mid, jvalue *args)
{
  /* Instrument Call:C→Java for Use state transition. */
  jobject_set refs = jinn_get_thread_local_jobject_set();
  if ((clazz != NULL) && !jinn_refs_contains(refs, clazz)) {
    /* Raise a JNI exception. */
    return jinn_throw_JNIException(env, Error: dangling);
  }
  /* Call the wrapped JNI function. */
  CallStaticVoidMethodA(env, clazz, mid, args);
}
```

Figure 3.4: Wrapper for function CallStaticVoidMethodA from Figure 3.1 Line 15 with instrumentation for Use state transition.

3.2 Dynamic Analysis Synthesis

We use state machine specifications like the one in Figure 3.2 to synthesize a dynamic analysis. Each state machine specification describes state transitions, which are triggered by language transitions. Their cross-product yields thousands of checks in the dynamic analysis. For example, before executing the JNI call in Line 15 of Figure 3.1, the analysis enforces at least eight constraints:

- The Java interface pointer, `env`, matches the current C thread.
- The current JVM thread does not have pending exceptions.
- The current JVM thread did not disable GC to directly access Java objects including arrays.
- `cb->mid` is not NULL.
- `cb->receiver` is not NULL.
- `cb->receiver` is not a dangling JNI reference.
- `cb->receiver` is a reference to a Java `Class` object.
- The formal arguments of `cb->mid` are compatible with the actual arguments in `cb->receiver` and `jargs`.

Hand-coding all these constraints would be tedious and error-prone. Instead, we specify state machines as follows.

Defining state machine states and transitions: Each FFI constraint is defined by a state machine. The individual states are encoded as C data structures and the transitions as C code, which also checks whether a transition has, in fact, been triggered. For example, the if-statement in

Line 6 of Figure 3.4 is a transition check for determining whether the entity is currently in the Released state and should therefore transition to the Error: Dangling state. Each state machine specification M_i has a set of state transitions M_i.stateTransitions.

Mapping state transitions to language transitions: Each specification has a function M_i.languageTransitionsFor that maps state transitions to language transitions. The synthesizer consults this mapping to inject context-specific instrumentation into wrapper functions. For example, Figure 3.2 illustrates a mapping. Figures 3.3 and 3.4 show generated wrappers. Each state transition $s_a \rightarrow s_b$ may occur at a set

$$L = M_i.\mathsf{languageTransitionsFor}(s_a \rightarrow s_b)$$

of language transitions. Each language transition ℓ in this set is a record containing the fields function, direction (Call or Return), and entities (threads, parameters, and return values).

Applying state machines to entities: At runtime, the wrappers attach state machines to entities and then transition the entity-specific state machine(s) based on context, encoding the state machine states in thread-local storage. For example, the wrapper in Figure 3.3 associates a state machine with the receiver reference, transitions its state to Acquired, and encodes this information by adding the reference to the thread-local list refs. As already mentioned above, the analysis developer specifies state machine encodings as a set of mutable data structures and functions that manipulate those structures.

Algorithm 1 Input: state machine specifications $M_1, \dots, M_n$. Output: FFI wrapper functions instrumented with dynamic checker.

for each state machine specification $M_i \in \{M_1, \dots, M_n\}$ **do**
    for each state transition $s_a \to s_b \in M_i$.stateTransitions **do**
        let $L = M_i$.languageTransitionsFor$(s_a \to s_b)$
        for each language transition $\ell \in L$ **do**
            let w be the wrapper for ℓ.function
            add the following synthesized code to the start or end of w, depending on whether ℓ.direction is Call or Return:
            for each entity $e \in \ell$.entities **do**
                if e satisfies the transition check for $s_a \to s_b$ **then**
                    modify the state machine encoding to record the transition of e from s_a to s_b.

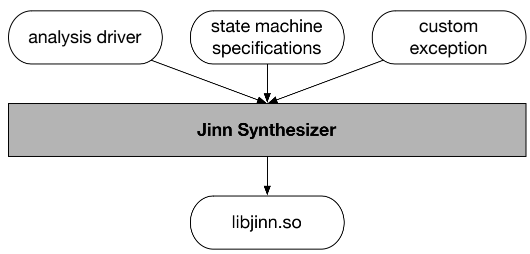


Figure 3.5: Structure of *Jinn* Synthesizer.

The state machine specifications consisting of these three components (state transitions, mappings from state transitions to language transitions, and state machine encodings) serve as input to Algorithm 1. The algorithm computes the cross product of state transitions and FFI functions, and then generates a wrapper for each FFI function that performs the appropriate state transformations and error checking. This functionality is the core of the Jinn Synthesizer component in Figure 3.5.

The synthesizer takes two additional inputs: an analysis driver and a custom exception. The output of the synthesizer is *Jinn*—a shared object file that the JVM dynamically loads using the JVM tools interface (JVMTI). The analysis driver initializes the state machine encodings and dynamically injects the generated, wrapped FFI functions into a running program. The custom exception defines how the dynamic analysis reports errors. *Jinn* monitors runtime events and program state. When *Jinn* detects a bug, it throws the custom exception. If the exception is not handled, the JVM prints a message with the JNI constraint violation and the faulting JNI function call. If *Jinn* is invoked within a debugger, the programmer can inspect the call chain, program state, and other potential causes of the failure.

3.3 State Machines

This section describes how three classes of JNI constraints map to state machines. Language transitions are mapped to state transitions. Error states indicate that a current sequence of language transitions violates a JNI constraint.

3.3.1 Thread State Constraints

To enter the JVM through any JNI function, C code must satisfy three conditions: (1) The JNI environment pointer JNIEnv* and the caller belong to the same thread. (2) Either no exception is pending, or the callee is exception-oblivious. (3) Either no critical region is active, or the callee is critical-region oblivious.

JNIEnv* state constraint. Figure 3.6 shows a state machine for JNIEnv* state constraint. All calls from Java to C implicitly pass a pointer to the JNIEnv structure, which specifies the JVM-internal and thread-local state. All calls

JNIEnv* state

Observed entity: A thread.

Error(s) discovered: JNIEnv* mismatch.

State machine encoding: Map from thread IDs to their expected JNIEnv* pointers.

State machine diagram: Trivial, omitted for brevity.

State transition	*Language transition*	*Triggering functions*
JNI call	Call:C$\rightarrow$Java	Any JNI function e.g., CallVoidMethod

Figure 3.6: A state machine for JNIEnv* state constraints.

from C to Java must explicitly pass the correct pointer when invoking a JNI function. When the program creates a native thread, *Jinn* learns about the JNIEnv* pointer from the JVM and retrieves the thread ID from the operating system. It enters both into the state machine encoding, which is a map from thread ID to JNIEnv* pointer. Later, when a native thread calls any of the 229 JNI functions, *Jinn* looks up the expected JNIEnv* from the state machine encoding and compares it to the actual parameter of the call, reporting an error if the pointers differ.

Exception state constraints. Figure 3.7 shows a state machine for exception state constraints. When Java code throws an exception and returns to C, the C code does not automatically transfer control to the nearest exception handler. The program must explicitly consume or propagate the pending exception. This constraint results from the semantic mismatch in how C and

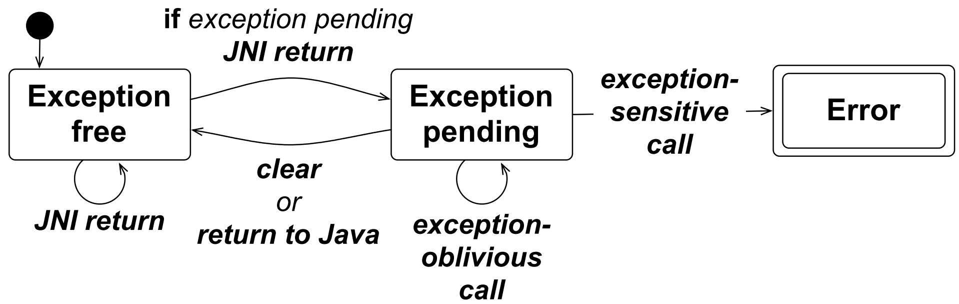
Exception state

Observed entity: A thread.

Error(s) discovered: Unhandled Java exception.

State machine encoding: Internal JVM structures.

State machine diagram:



State transition	*Language transition*	*Triggering functions*
JNI return	Return:Java→C	Any JNI function e.g., CallVoidMethod
Clear or return to Java	Return:Java→C Return:C→Java	ExceptionClear Return from any native method
Exception-oblivious call	Call:C→Java	Small set of clean-up functions e.g., ReleaseStringChars
Exception-sensitive call	Call:C→Java	All other JNI functions e.g., GetStringChars

Figure 3.7: A state machine for exception state constraints.

Java handle exceptions. Any JNI call may lead to Java code that throws an exception, which causes a transition to the "exception pending" state when the JNI call returns. The JVM internally records this state transition for each Java thread, so *Jinn* does not need to interpose on JNI returns to track exception states. It can instead simply rely on the JVM-internal data structure for its state machine encoding. If the program returns from a JNI call and an exception is pending, the program must consume or propagate the exception. To do so, the programmer may first select from one of 20 *exception-oblivious* JNI functions that query the exception state and release JVM resources before calling JNI's ExceptionClear function. If the programmer calls any of the remaining *exception-sensitive* JNI functions while an exception is pending, *Jinn* intercedes and wraps the pending exception in an error report to the user.

Critical-section state constraints. Figure 3.8 shows a state machine for critical section state constraints. JNI defines the phrase "JNI critical section" to describe a piece of C code that has direct access to a Java string or array, during which the JVM may take drastic measures such as disabling the garbage collector. A critical section starts with GetStringCritical or GetPrimitiveArrayCritical and ends with the matching ReleaseStringCritical or ReleasePrimitiveArrayCritical. C code should hold these resources only for a short time. To prevent deadlock, C code must not interact with the JVM other than to acquire or release critical resources. In other words, during a critical section, C code must only call one of the four functions that get/release arrays/strings. We call these four functions critical-section *insensitive* and all the remaining JNI functions critical-section *sensitive*. *Jinn* encodes the state machines by keeping, for each thread, a tally of the number of times that thread has acquired a specific critical resource. *Jinn* instruments the four "get" and "release" calls to manage

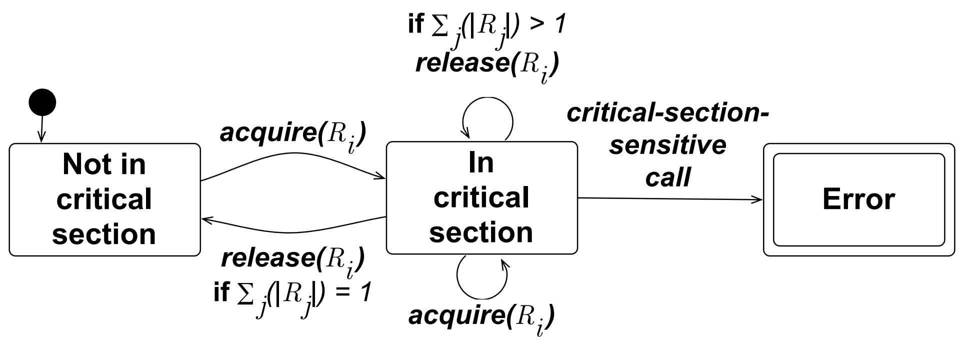
Critical-section state

Observed entity: A thread.

Error(s) discovered: Critical section violation.

State machine encoding: Map from a critical resource R_i to the number of times a given thread has acquired that resource.

State machine diagram:



State transition	*Language transition*	*Triggering Functions*
Acquire	Return:Java→C	GetStringCritical or GetPrimitiveArrayCritical
Release	Return:Java→C	ReleaseStringCritical or ReleasePrimitiveArrayCritical
Critical-section sensitive call	Call:C→Java	All other JNI functions e.g., CallVoidMethod

Figure 3.8: A state machine for critical section state constraints.

these counts. Each acquisition of a resource R_i must be matched by a corresponding release. When the list of critical resources for a thread toggles between empty and non-empty, the critical-section state machine transitions correspondingly. *Jinn* interposes on all the 225 critical-section sensitive functions to verify that the thread currently maintains no critical resources and that releases are well-matched.

Critical sections are tricky because they prohibit calls to most JNI functions, including those that Jinn uses for its own error checking. For example, *Jinn* does not check whether or not the argument to ReleaseStringCritical is in fact a Java string since that would require calling IsAssignableFrom from within a critical region. At the same time, C code cannot exercise much JNI functionality while in a critical section and can legally call only four functions—to acquire more critical sections and to release them again.

3.3.2 Type Constraints

When Java code calls a Java method, the compiler and JVM check type constraints on the parameters. However, when C code calls a Java method, the compiler and JVM do not check type constraints, and type violations cause unspecified JVM behavior. For example, given the Java code

```
Collections.sort(ls, cmp);
```

the Java compiler checks that class Collections has a static method sort and that the actual parameters ls and cmp conform to the formal parameters of sort. Consider the equivalent code expressed with Java reflection:

```
Class clazz = Collections.class;
Method method =
  clazz.getMethod("sort", List.class, Comparator.class);
method.invoke(Collections.class, ls, cmp);
```

The Java compiler cannot statically verify its safety, but if the program is unsafe at runtime, then the JVM throws an exception. In JNI, this code is expressed as follows:

```
jclass clazz = (*env)->FindClass(env, "java/util/Collections");
jmethodID method = (*env)->GetStaticMethodID(env, clazz,
   "sort", "(Ljava/lang/List;Ljava/util/Comparator;)V");
(*env)->CallStaticVoidMethod(env, clazz, method, ls, cmp);
```

Since the C code expresses Java type information in strings, standard static type checking cannot resolve the types and even sophisticated interprocedural analysis cannot always resolve them [25, 77]. Consequently, the C compiler does not statically enforce typing constraints on the "Collections" and "sort" names or the ls and cmp parameters. Furthermore, and unlike Java reflection, JNI does not even dynamically enforce typing constraints on the clazz and method descriptors. In contrast, *Jinn* does enforce these and other JNI type constraints dynamically.

Fixed typing constraints. Figure 3.9 presents a state machine for fixed type constraints. Type constraints require the runtime type of actuals to conform to the formals. For many JNI functions, the parameter type is, in fact, *fixed* by the function itself. For example, in CallStaticVoidMethod(env, clazz, method, ls, cmp), the clazz actual must always conform to type java.lang.Class. We extracted this and comparable constraints by scanning the JNI header file for C parameters (e.g., jstring) with well-defined corresponding Java types

Fixed typing

Observed entity: A reference parameter.

Error(s) discovered: Type mismatch between actual and formal parameter to JNI function.

State machine encoding: Map from entity IDs to their signatures.

State machine diagram: Trivial, omitted for brevity.

State transition	*Language transition*	*Triggering functions*
JNI call	Call:C→Java	JNI function defining a parameter with a fixed type, e.g., clazz parameter to CallStaticVoidMethod

Figure 3.9: A state machine for fixed typing constraints.

(e.g., java.lang.String). We extracted additional fixed typing constraints from the informal JNI explanation in [49]. For example, FromReflectedMethod has a jobject parameter, whose expected type is either java.lang.reflect.Method or java.lang.reflect.Constructor. Overall, *Jinn* interposes on 151 JNI functions to verify 157 fixed typing constraints. For each check, *Jinn* obtains the class of the actual using GetObjectType and then checks compatibility with the expected type through IsAssignableFrom.

Entity-specific typing constraints. Figure 3.10 presents the state machine for entity-specific typing constraints. A plethora of JNI functions call Java methods or access Java fields. JNI references Java methods and fields via *entity IDs*. For example, in CallStaticVoidMethod(env, clazz, method, ls, cmp), the parameter method is a method ID. In this case, the method must be static, and the method parameter constrains the other parameters. In particular, the

Entity-specific typing

Observed entity: A pair of ID parameters.

Error(s) discovered: Type mismatch for Java field assignment or between actual and formal of a Java method.

State machine encoding: Map from entity IDs to their signatures.

State machine diagram: Trivial, omitted for brevity.

State transition	*Language transition*	*Triggering functions*
JNI call	Call: C→Java	JNI function defining parameters with interrelated types, e.g., clazz and method in CallStaticVoidMethod

Figure 3.10: A state machine for entity-specific typing constraints.

clazz must declare the method, and ls and cmp must conform to the formal parameters of the method. *Jinn* records method and field signatures upon return from JNI functions that produce method and field IDs. The entity ID constrains the types of method parameters or field values as well as the receiver class (for static entities) or object (for instance entities) for each of 131 JNI functions that access a Java entity. When a program calls one of these functions that take an entity ID, *Jinn* interposes on the call to verify that the function conforms to the entity's typing constraints.

Access control constraints. Figure 3.11 presents the state machine for access control constraints. Even when type constraints are satisfied, Java semantics may prohibit accesses based on visibility and final modifiers. For example, in SetStaticIntField(env, clazz, fid, 42), the field identified by fid may be private or final, in which case the assignment follows questionable coding practices.

Access control

Observed entity: A field ID.

Error(s) discovered: Assignment to final field.

State machine encoding: Map from field IDs to their modifiers.

State machine diagram: Trivial, omitted for brevity.

State transition	*Language transition*	*Triggering functions*
JNI call	Call:C→Java	Set<*Type*>Field or SetStatic<*Type*>Field

Figure 3.11: A state machine for access control constraints.

The JNI specification is vague on legal accesses with respect to their visibility and final constraints. After some investigation, we found that in practice, JNI usually ignores visibility, but honors the final modifier. Ignoring visibility rules seems surprising, but as it turns out, this permissiveness is consistent with the behavior of reflection, which may suppress Java access control when setAccessible(true) was successful. Honoring final is common sense. Despite the fact that reflection may mutate final fields, mutating them interferes with JIT optimizations and concurrency and complicates the Java memory model. As with entity-specific typing, *Jinn* keeps track of field IDs, as well as which fields are final. *Jinn* raises an error if native code calls any of the 18 JNI functions that might assign to a final field.

Nullness constraints. Figure 3.12 presents the state machine for nullness constraints. Some JNI function parameters must not be null. For example, in CallStaticVoidMethod(env, clazz, method, ls, cmp), the parameters env, clazz, and

Nullness

Observed entity: A reference parameter.

Error(s) discovered: Unexpected null value passed to JNI function.

State machine encoding: None.

State machine diagram: Trivial, omitted for brevity.

State transition	*Language transition*	*Triggering functions*
JNI call	Call:C→Java	JNI function defining a parameter that must not be null, e.g., method parameter to CallStaticVoidMethod

Figure 3.12: A state machine for nullness constraints.

method must not be null. At the same time, some JNI functions do accept null parameters — for example, the initial array elements in NewObjectArray. Since the JNI specification is not always clear on which parameters may be null, we determined these constraints experimentally. We uncovered 416 non-null constraints among the 210 JNI functions that define parameters. *Jinn* reports to the user when the program violates any of these constraints.

3.3.3 Resource Constraints

A JNI resource is a piece of Java-related data that C code can acquire or release through JNI calls. For example, C code can acquire a Java string or array. Depending on the JVM implementation, the JVM either pins the string or array to prevent the garbage collector from moving it or copies the array and then passes C code a pointer to the contents. Other JNI resources include various kinds of opaque references to Java objects, which C code can pass to JNI functions and which give C code some control over Java memory management. Finally, JNI can acquire or release Java monitors, which are a mutual-exclusion primitive for multi-threaded code.

APIs with manual or semi-automatic memory management suffer from well-known problems: (1) Section 3.1.2 illustrated one such problem: a use after a release corrupts JVM state through a dangling reference. There are three other common resource errors. (2) An acquire at insufficient capacity causes an overflow. (3) A missing release at the end of reference lifetime causes a leak. (4) A second release is a double-free. The *Jinn* analysis depends on the resource (e.g., array, string reference, object). In a few cases, *Jinn* cannot detect certain error conditions because they are underspecified or hidden in C code. For instance, *Jinn* currently cannot detect when C code uses an invalid C pointer without calling a JNI function. In a few cases, *Jinn* need not check resource-related errors since the JVM or other *Jinn* state machines already trap them. For example, when the JVM throws an OutOfMemoryError exception, *Jinn* already checks for correct exception handling.

While the state machines and error cases for all kinds of JNI resources are similar, they differ in the details due to the above reasons. Figures 3.13, 3.14, 3.15, and 3.16 show these four different resource cases separately, and we now discuss each in more detail.

Pinned or copied string or array constraints. Figure 3.13 shows a state machine for pinned or copied string or array constraints. C code can temporarily obtain direct access to the contents of a Java string or array. JVMs may pin or copy the object to facilitate garbage collection. To make sure the JVM unpins the object or frees the copy, the C code must properly pair acquire/re-

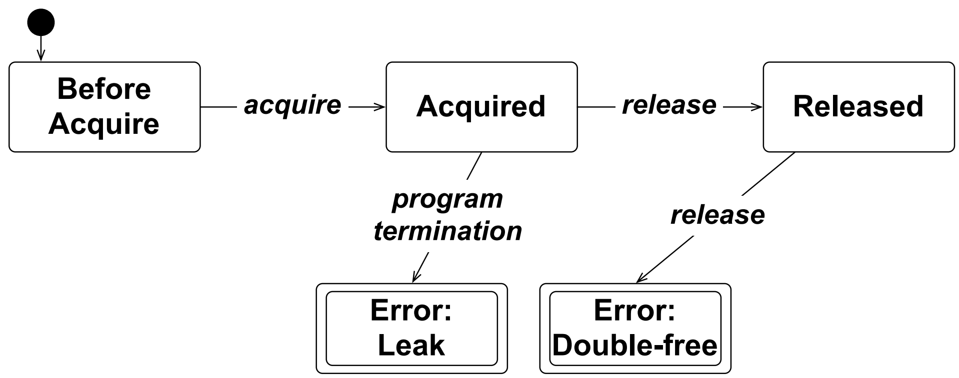
Pinned or copied string or array

Observed entity: A Java string or array that is pinned or copied.

Error(s) discovered: Leak and double-free.

State machine encoding: A list of acquired JVM resources.

State machine diagram:



State transition	*Language transition*	*Triggering functions*
Acquire	Return:Java→C	Get<*Type*>ArrayElements and similar getter functions
Release	Return:Java→C	Release<*Type*>ArrayElements and similar release functions
Program termination		JVMTI callback

Figure 3.13: A state machine for pinned or copied string or array.

lease calls. *Jinn* reports a leak for any resource that has not been released at program termination. *Jinn* reports a double-free for a resource it has already evicted from its state machine representation due to an earlier free. *Jinn* does not check for dangling references because their uses happen in C code. *Jinn* does not check for overflow (i.e., an out-of-memory condition) in this state machine because its exception checking subsumes this check.

Monitor constraints. Figure 3.14 shows a state machine for monitor constraints. A monitor is a Java mutual exclusion primitive. *Jinn* need not check overflow or double-free for monitors since the JVM already throws exceptions. *Jinn* cannot check dangling monitors, since that requires divining when the programmer intended to release it. *Jinn* does report if a monitor is not released at program termination, which indicates a risk of deadlock.

Global reference or weak global reference constraints. Figure 3.15 shows the state machine for a global reference and a weak global reference. *A global* or *weak global reference* is an opaque pointer from C to a Java object that is valid across JNI calls and threads. These references are explicitly managed because the garbage collector needs to update them when moving objects and also treat global (but not weak) references as root. *Jinn* reports a leak for any unreleased global or weak global reference at program termination. *Jinn* reports a dangling reference error if the program uses a reference after a free. *Double-free* is a special case of the dangling reference error, and *overflow* is a special case of *Jinn*'s exception state constraints.

Local reference constraints. Figure 3.16 shows a state machine for local reference constraints. JNI manages local references semi-automatically:

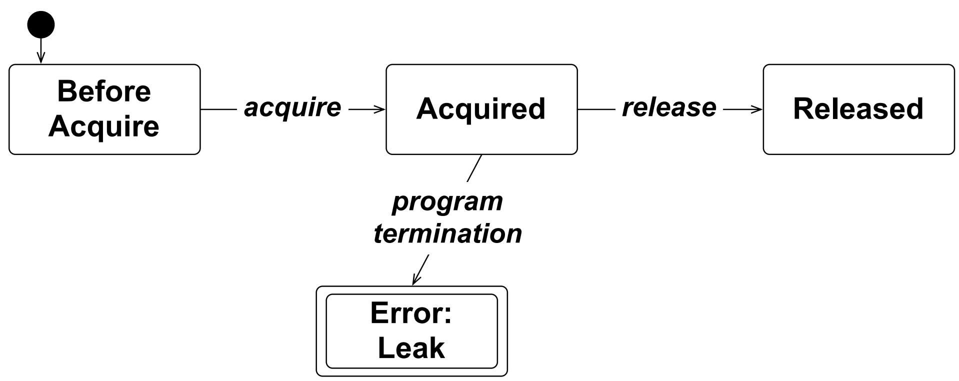
Monitor

Observed entity: A monitor.

Error(s) discovered: Leak.

State machine encoding: A set of monitors currently held by JNI and, for each monitor, the current entry count.

State machine diagram:



State transition	*Language transition*	*Triggering functions*
Acquire	Call:C$\rightarrow$Java	MonitorEnter
Release	Call:C$\rightarrow$Java	MonitorExit
Program termination		JVMTI callback

Figure 3.14: A state machine for monitor constraints.

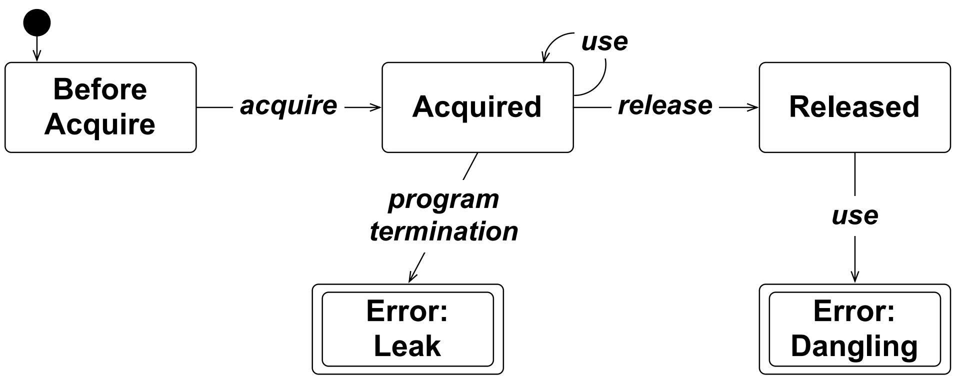
Global reference or weak global reference

Observed entity: A global or weak global JNI reference

Error(s) discovered: Leak and dangling reference.

State machine encoding: A list of acquired global references.

State machine diagram:



State transition	*Language transition*	*Triggering functions*
Acquire	Return:Java→C	NewGlobalRef and NewWeakGlobalRef
Release	Return:Java→C	DeleteGlobalRef and DeleteWeakGlobalRef
Use	Call:C→Java	JNI function taking reference e.g., CallVoidMethod
	Return:C→Java	Native method returning reference, e.g., Class.getClassContext
Program termination		JVMTI callback

Figure 3.15: A state machine for global reference or weak global reference.

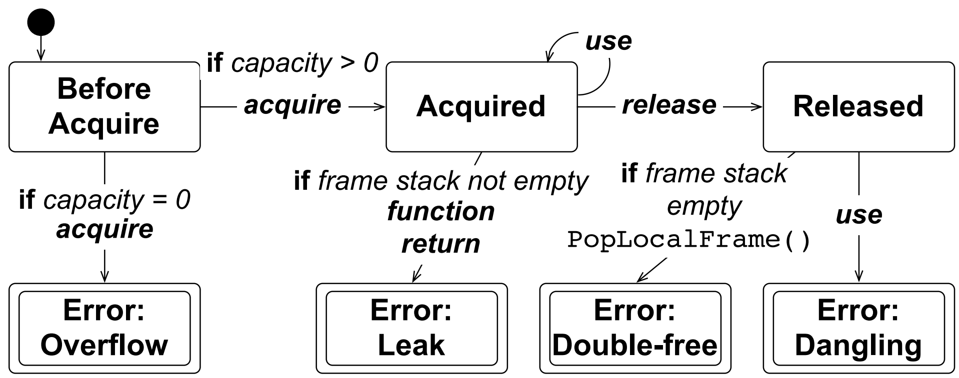
Local reference

Observed entity: A local JNI reference

Error(s) discovered: Overflow, leak, dangling, and double-free.

State machine encoding: For each thread, a stack of frames. Each frame has a capacity and a list of local references.

State machine diagram:



State transition	*Language transition*	*Triggering functions*
Acquire	Call:Java→C	Native method taking reference
	Return:Java→C	JNI function returning reference e.g., GetObjectField
Release	Return:Java→C	DeleteLocalRef or PopLocalFrame
	Return:C→Java	Return from any native method
Use	Call:C→Java	JNI function taking reference e.g., CallVoidMethod
	Return:C→Java	Native method returning reference, e.g., Class.getClassContext

Figure 3.16: A state machine for local reference.

acquire and release are more often implicit than explicit. Native code implicitly *acquires* a local reference when a Java native call passes it to C or when a JNI function returns it. The JVM *releases* local references automatically when native code returns to Java, but the user can also manually release one (DeleteLocalRef) or several (PopLocalFrame) local references. *Jinn* enters the reference into its state machine encoding upon acquire and removes it upon release. *Jinn* performs bookkeeping to support overflow checks since the JNI specification only guarantees space for up to 16 local references. If more are needed, the user must explicitly request additional capacity with PushLocalFrame and later release that space with PopLocalFrame. *Jinn* keeps track of local frames and checks four error cases as follows: (1) *Jinn* detects *overflow* if the current local frame exceeds capacity. (2) JNI releases individual local references automatically; *Jinn* checks for *leaked* local reference frames when native code returns to Java. (3) *Jinn* checks that local references passed as parameters to JNI functions are not *dangling* and, furthermore, belong to the current thread. (4) *Jinn* detects a *double-free* when DeleteLocalRef is called twice for the same reference or if nothing is left to pop on a call to PopLocalFrame.

3.4 Generalization

This section demonstrates that our technique generalizes to other languages by applying it to Python/C 2.6 [81]. We first discuss the similarities and differences between JNI and Python/C. We then present a synthesized dynamic checker for Python/C's manual memory management. We leave to future work the full specification of Python/C FFI constraints and the complete implementation of a dynamic analysis for these constraints.

3.4.1 Python/C Constraint Classification

Like JNI, the Python/C specification describes numerous rules that constrain how programmers can combine Python and C. These constraints fall into the same classes from Section 5.5.2: (1) interpreter state constraints, (2) type constraints, and (3) resource constraints.

State constraints. Python/C constrains the behavior of exceptions and threads. Python/C's exception constraints mirror those of JNI: C code should immediately handle the exception or propagate it back to Python. While not explicitly stated in the manual, these constraints also imply that native code should not invoke other Python/C functions while an exception is pending. For thread constraints, Python/C differs slightly from JNI because Python's threading model is simpler than Java's. For each instantiation of the Python interpreter, a thread must possess the Global Interpreter Lock (GIL) to execute. The Python interpreter contains a scheduler that periodically acquires and releases the GIL on behalf of a program's threads.

Python/C permits C code to release and re-acquire the GIL around blocking I/O operations. It also permits C code to create its own threads and bootstrap them into Python. Because C code may manipulate thread state directly, the programmer may write code that deadlocks. For example, the programmer may accidentally acquire the GIL twice. As a result, Python/C requires bookkeeping for the GIL similar to that for JNI critical sections discussed in Section 3.3.1.

Type constraints. Because Python is dynamically typed, types in Python/C are less constrained than in JNI. The Python interpreter performs dynamic type checking for many operations on built-in types. However, sometimes the interpreter forgoes these type checks—as well as some null checks—for performance reasons. Consequently, if a program passes a mistyped value to a Python/C call, the program may crash or exhibit undefined behavior. A dynamic analysis based on the type constraints of Section 3.3.2 would enable reliable detection of these errors, at the cost of reintroducing dynamic checking for some Python/C functions.

Resource constraints. Python employs reference counting for memory management. To Python code, reference counting is transparent and fully automatic. However, native-code programmers must manually increment and decrement a Python object's reference count, according to the Python/C manual's instructions. To this end, the Python/C manual defines a notion of *reference co-ownership*. Each reference that co-owns an object is responsible for decrementing the object's reference count when it no longer needs that object. Neglecting to decrement leads to memory leaks. C code may also *borrow* a reference. Borrowing a reference does not increase its reference count, but using a borrowed reference to a freed object is a dangling reference error. The Python/C manual specifies which kinds of references are returned by the various FFI functions. A dynamic checker must track the state of these references in order to report usage violations to the user.

3.4.2 Synthesizing Dynamic Checkers

To ensure that FFI programs correctly use co-owned and borrowed references, we implemented a use-after-release checker for Python/C's reference counting memory management.

```
1.  static PyObject* dangle_bug(PyObject* self, PyObject* args) {
2.    PyObject *pythons, *first;
3.              /* Create and delete a list with a string element.*/
4.    pythons = Py_BuildValue([ssssss],
5.      Eric, Graham, John, Michael, Terry, Terry);
6.    first = PyList_GetItem(pythons, 0);
7.    printf(1. first = %s.\n, PyString_AsString(first));
8.    Py_DECREF(pythons);
9.                                   /* Use dangling reference. */
10.   printf(2. first = %s.\n, PyString_AsString(first));
11.             /* Return ownership of the Python None object. */
12.   Py_INCREF(Py_None);
13.   return Py_None;
14. }
```

Figure 3.17: Python/C dangling reference error. The borrowed reference first becomes a dangling reference when pythons dies.

Example memory management error. Figure 3.17 contains an example Python/C function that mismanages its references. The reference first in Line 6 is borrowed from the reference pythons. When Line 8 decrements the reference count for pythons, the reference dies. The Python/C manual states that the program should no longer use first, but the program uses this reference at Line 10. This use is a dangling reference error, and Python's semantics are undefined for such a case. In practice, Figure 3.17's behavior depends on whether the interpreter reuses the memory for first between the implicit release in Line 8 and the explicit use in Line 10.

Synthesizer and generated checker. Our synthesizer takes a specification file that lists which functions return new or borrowed references. The generated checker detects memory management errors by tracking co-owned references and their borrowers. For example, the checker determines that `pythons` is a co-owned reference and that `first` borrows from `pythons`. When a co-owner relinquishes a reference by decrementing its count, all its borrowed references become invalid. If the program uses an invalid borrowed reference, as Figure 3.17 does on Line 10, then the checker signals an error.

Interposing on language transitions. Integrating the dynamic analysis with Python/C is more challenging than for JNI. Python lacks an interface comparable to the JVM tools interface and thus requires that the dynamic analysis be statically linked with the interpreter. Furthermore, Python/C bakes in some of the Python interpreter's implementation details, which makes the API less portable than JNI and complicates interposing on language transitions. In particular, (1) Python/C makes prevalent use of C macros, (2) the Python interpreter internally uses Python/C functions, and (3) some variadic functions lack non-variadic counterparts.

Python/C makes extensive use of C macros. Some macros directly modify the interpreter state without executing a function call. Because Python/C does not execute a function, our dynamic analysis has nothing to interpose on and cannot track the behavior. We overcame this limitation by replacing the macros with equivalent functions. This change requires programmers to re-compile their native code extensions against our customized interpreter, but it does not require them to change their code.

Because the Python interpreter internally calls Python/C functions, the dynamic analysis cannot easily detect application-level language transitions. Even if it could detect such transitions, function interposition and transition detection would significantly slow down the interpreter. We overcame this limitation by creating an interpreter-only copy of every Python/C function.

We then used automatic code-rewriting to make the interpreter call the unmodified copies. Our dynamic analysis interposes on the originals, which are used by native code extensions.

A *variadic* C function such as `printf` takes a variable number of arguments. Our synthesizer interposes on each variadic function by wrapping it with code that calls an equivalent, non-variadic version of the function such as `vprintf`. Python does not provide non-variadic equivalents for all its variadic functions; where necessary, we added the non-variadic equivalents.

Despite these implementation challenges, our Python/C dynamic analysis substantially follows from our JNI dynamic analysis, thus providing evidence of the generality of our approach. Both FFIs have large numbers of constraints that fall into three classes. Both FFIs also support specifying the constraints as state machines, mapping state machine transitions to language transitions, and then applying the state machines to program entities.

3.5 Results

This section evaluates the performance, coverage, and usability of *Jinn* to support our claim that it is the most practical FFI bug finder to date.

3.5.1 Methodology

Experimental environments. We used two production JVMs, Sun HotSpot Client 1.6.0_10 and IBM J9 1.6.0 SR5. We conducted all experiments on a Pentium D T3200 with 2GHz clock, 1MB L2 cache, and 2GB main memory. The machine runs Ubuntu 9.10 on the Linux 2.6.31-19 kernel.

JNI programs. We used several JNI programs: microbenchmarks, SPECjvm98 [68], DaCapo [9], Subversion 39004 (2009-08-31), Java-gnome-4.0.10, and Eclipse 3.4. The microbenchmarks are a collection of 16 small JNI programs, which are designed to trigger one each of the error states in the eleven state machines shown in Figures 3.6, 3.7, 3.8, 3.9, 3.10, 3.11, 3.12, 3.13, 3.14, 3.15, and 3.16. The microbenchmarks also cover all pitfalls in Table 6.1 with exception of Pitfall 8, which cannot be detected at the language boundary. SPECjvm98 and DaCapo are written in Java, but exercise native code in the system library. Subversion, Java-gnome, and Eclipse mix Java and C in user-level libraries. Except for Eclipse 3.4, we use fixed inputs.

Dynamic JNI checkers. We compare three dynamic JNI checkers. Two of them — IBM and SUN JVMs — use runtime checking and are turned on by the -Xcheck:jni option. The third — *Jinn*— is turned on by the -agentlib:jinn option in any JVM.

Experimental data. We collected timing and statistics results by taking the median of 30-100 trials to statistically tolerate experimental noise. The runtime systems show non-deterministic behavior from a variety of sources: micro architectural events, OS scheduling, and adaptive JIT optimizations.

3.5.2 Performance

This section evaluates the performance of *Jinn*. Table 3.1 shows the results. *Jinn* adds instructions to every language transition between the JVM and native libraries, interposing and checking transitions. The second column counts the total number of transitions between Java and C in the system li-

Benchmark	Language transition counts	Normalized execution times Runtime checking	*Jinn* Interposing	*Jinn* Checking
antlr	441,789	1.04	0.98	1.05
bloat	839,930	1.02	1.19	1.20
chart	1,006,933	1.02	1.08	1.12
eclipse	8,456,840	1.01	1.17	1.20
fop	1,976,384	1.07	1.14	1.37
hsqldb	206,829	0.88	1.04	1.05
jython	56,318,101	1.03	1.10	1.16
luindex	1,339,059	1.03	1.08	1.13
lusearch	4,080,540	1.04	1.09	1.21
pmd	967,430	1.04	1.10	1.13
xalan	1,114,000	1.01	1.17	1.19
compress	14,878	0.98	1.09	1.08
jess	153,118	0.99	1.22	1.17
raytrace	29,977	1.04	1.16	1.14
db	133,112	0.99	1.01	1.02
javac	258,553	1.06	1.16	1.14
mpegaudio	46,208	1.00	1.01	1.04
mtrt	32,231	1.01	1.11	1.14
jack	1,332,678	1.04	1.10	1.21
GeoMean		1.01	1.10	1.14

Table 3.1: *Jinn* performance on SPECjvm and DaCapo with HotSpot.

braries using HotSpot. The third column shows the execution times of runtime checking for HotSpot. Execution times are normalized to production runs of HotSpot without any dynamic checking. The fourth column reports *Jinn*'s framework overhead due to interposition on language transitions. The fifth column reports the total time, which includes state machine encoding, transitions, and error checking. On average, *Jinn* has a modest 14% execution time overhead and most of the overhead (all but 4%) comes from runtime interposition, rather than executing the analysis code.

3.5.3 Coverage of *Jinn* and JVM Runtime Checking

This section shows that *Jinn* covers qualitatively and quantitatively more JNI bugs than the state-of-art dynamic checking in production JVMs.

Quality. We run the 16 microbenchmarks with HotSpot, J9, and *Jinn*. Figure 3.18 compares their error messages on the representative *Exception-State* microbenchmark, which violates the exception state constraints of Section 3.3.1. The C code in the benchmark ignores an exception raised by Java code and calls two JNI functions: GetMethodID and CallVoidMethod. HotSpot reports that there were two illegal JNI calls but does not identify the offending JNI function calls. J9 reports the first JNI function (GetMethodID) but does not show the calling context for the first bad JNI call because J9 aborts the JVM.

Jinn reports both illegal JNI calls, their calling contexts, and the source location of the original Java exception. In addition to precise reports, *Jinn*'s error reporting integrates with debuggers. Java debuggers like jdb and Eclipse JDT can catch the custom exception, and programmers can then inspect the

```
WARNING in native method:
  JNI call made with exception pending at
  ExceptionState.call(Native Method) at
  ExceptionState.main(ExceptionState.java:5)
WARNING in native method:
  JNI call made with exception pending at
  ExceptionState.call(Native Method) at
  ExceptionState.main(ExceptionState.java:5)
```

(a) HotSpot

```
JVMJNCK028E JNI error in GetMethodID: This function
  cannot be called when an exception is pending
JVMJNCK077E Error detected in ExceptionState.call()V
JVMJNCK024E JNI error detected. Aborting.
JVMJNCK025I Use -Xcheck:jni:nonfatal to continue running
  when errors are detected.
Fatal error: JNI error
```

(b) J9

```
Exception in thread main JNIAssertionFailure:
  An exception is pending in CallVoidMethod.
  at jinn.JNIAssertionFailure.assertFail
  at ExceptionState.call(Native Method)
  at ExceptionState.main(ExceptionState.java:5)
Caused by: jinn.JNIAssertionFailure:
  An exception is pending in GetMethodID.
  ... 3 more
Caused by: java.lang.RuntimeException:
  checked by native code
  at ExceptionState.foo(ExceptionState.java:9)
  ... 2 more
```

(c) *Jinn*

Figure 3.18: Representative JVM and *Jinn* error messages using a microbenchmark that violates the *exception state* constraint.

Java state to find the failure's cause. Even better, the Blink Java/C debugger (Chapter 4) can present the entire program state, including the full calling context consisting of both Java and C frames.

Quantity. The behavior of the production runs without dynamic checkers ranges from ignoring the bug to simply crashing to raising a null pointer exception—none of which are correct. The dynamic checkers built into the HotSpot and J9 JVMs also behave inconsistently in more than half of our microbenchmarks (9 of 16). *Jinn* is the only dynamic bug-finder that consistently detects and reports the JNI bugs in our 16 microbenchmarks by throwing an exception. Quantitative coverage of *Jinn*, HotSpot, and J9 is 100%, 56%, and 50%, respectively, with exceptions, warnings (print to console and keep running), and errors (print to console and terminate) counting as valid bug reports. *Jinn*'s 100% coverage on our own, specifically designed test suite is hardly surprising and does not imply that *Jinn* catches all JNI bugs. But the low JVM coverage demonstrates that error checking in previous practice was at best incomplete. Furthermore, JNI constraint violations are common and well-documented [25, 26, 43, 45, 48, 75, 76], underlining the need for better constraint enforcement.

3.5.4 Usability with Open Source Programs

This section evaluates the usability of *Jinn* based on our experience of running *Jinn* over Subversion, Java-gnome, and Eclipse. All these open-source programs are in wide industrial and academic use with a long revision history. These case studies show that *Jinn* finds errors in widely-used programs. In fact, *Jinn* found bugs in every substantial Java program we tested.

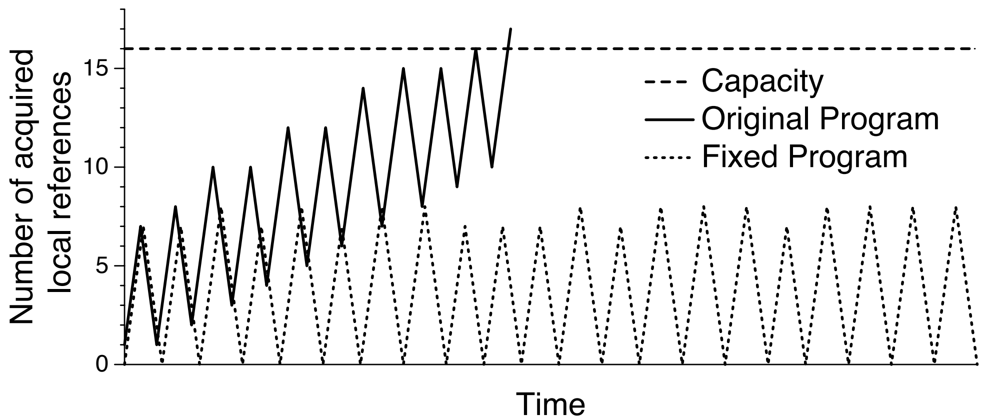


Figure 3.19: Time-series of acquired local references with leak and its fix in the fourth execution of a Java native method: Java_org_tigris_subversion_javahl_SVNClient_info2.

3.5.4.1 Subversion

Running Subversion's regression test suite under *Jinn*, we found two overflows of local references and one dangling local reference.

Overflow of local references. *Jinn* found that Subversion allocated more than 16 local references in two call sites to JNI functions: line 99 in Outputer.cpp and line 144 in InfoCallback.cpp. Figure 3.19 compares the time-series of acquired local references for the original and the fixed program. The original program overflows the pool of 16 local references without requesting more capacity—as detected by *Jinn* when acquiring yet another local reference. One reported source line is:

```
jstring jreportUUID =
  JNIUtil::makeJString(info->repos_UUID);
```

After looking at the calling context, we found that the program misses a call to DeleteLocalRef. We inserted the following lines:

```
env->DeleteLocalRef(jreportUUID);
if (JNIUtil::isJavaExceptionThrown()) return NULL;
```

After re-compiling, the program passes the regression test even under *Jinn*, since the number of active local references never exceeds 8. This overflow did not crash HotSpot and J9 but represents a time bomb. A highly optimized JVM may crash if it assumes that JNI code is well-behaved and eliminates bound checking of the bump pointer for local references.

Use of dangling local reference. The use of a dangling local reference happens at the execution of a C++ destructor when the C++ variable path goes out of scope in file CopySources.cpp.

```
{
  JNIStringHolder path(jpath);
  env->DeleteLocalRef(jpath);
} /* The destructor of JNIStringHolder is executed here. */
```

At the declaration of path, the constructor of JNIStringHolder stores the JNI local reference jpath in the member variable path::m_jtext. Later, the call DeleteLocalRef releases the jpath local reference, and thus path::m_jtext dies. When the program exits from the C++ block, it calls the destructor of JNIStringHolder. Unfortunately, this destructor uses the dead JNI local reference:

```
JNIStringHolder::~JNIStringHolder() {
  if (m_jtext && m_str)
    m_env->ReleaseStringUTFChars(m_jtext, m_str);
}
```

The JNI function ReleaseStringUTFChars uses the dangling JNI reference (m_jtext). This bug is not syntactically visible to the programmer because the C++ destructor feature obscures control flow when releasing resources. In our experience, this bug did not crash production JVMs. To understand it better, we looked at the internal implementation of ReleaseStringUTFChars in an open-source Java virtual machine (Jikes RVM). In Jikes RVM, ReleaseStringUTFChars ignores its first parameter, rendering the fact that the actual is a dangling reference irrelevant. If other JVMs are implemented similarly, this bug will remain hidden. Nonetheless, the code again represents a time bomb, because the bug will be exposed as soon as the program runs on a JVM where the implementation of ReleaseStringUTFChars uses its first parameter. For example, a JVM may internally represent strings in UTF8 format as proposed by Zilles [90] and then share them directly with JNI.

3.5.4.2 Java-gnome

Running Java-gnome's regression test suite under *Jinn*, we found one nullness bug and one dangling local reference.

Nullness. *Jinn* reports a bug identified previously in the Blink debugger paper (Chapter 4). Note, however, that Blink requires running the Java program in a full-fledged debugger while *Jinn* is a light-weight dynamic checker.

Use of dangling local reference. *Jinn* reports and diagnoses bug 576111 for the Java-gnome project, which violates a constraint on semi-automatic resources. *Jinn* reports that Line 348 of binding_java_signal.c violates a local reference constraint.

```
(*env)->CallStaticVoidMethodA(env, bjc->receiver,
                               bjc->method, jargs);
```

The bjc->receiver is a dead local reference. A Java-gnome developer confirmed the diagnosis. This bug did not crash HotSpot and J9, but, as noted before, bugs that are only benign due to implementation characteristics of a specific JVM vendor are time bombs and should be fixed.

3.5.4.3 Eclipse 3.4

We opened a Java project in Eclipse-3.4, and *Jinn* reported one violation of the entity-specific subtyping constraint in line 698 of callback.c in its SWT 3.4 component.

```
result =
  (*env)->CallStaticSWT_PTRMethodV(env, object, mid, vl);
```

The object must point to a Java class that has a static Java method identified by mid. The actual class did not have the static method, but its superclass declares the method. It is challenging for the programmer to ensure this constraint, because the source of the error involves dynamic callback control and a Java inner class. Because the production JVM may not use the object value, this bug has survived multiple revisions.

3.6 Summary

This thesis seeks to improve the correctness of multilingual programs. This chapter shows how to use a through FFI specification to generate dynamic analyses that check FFI code. We show how to encode constraints in about a dozen state machines. The three classes — thread state, type, and resource — capture the key semantic mismatches that multilingual interfaces must negotiate. The state machines, in turn, capture the complete constraints for correctly using such interfaces. This chapter show how to use synthesis for mapping the state machine specifications into context-sensitive dynamic bug checkers inserted at language transitions. Notably, we generate dynamic bug checkers for JNI and Python/C. We show that *Jinn*, the synthesized bug checker for JNI, uncovers previously unknown bugs in widely-used Java native libraries. Our approach to multilingual constraint representation, constraint generation, and FFI usage verification is the most general, concise, practical, and effective one to date.

Chapter 4

Interactively Examining Bugs across Language Interfaces

Some multilingual programming bugs are a source of fatal failures that cannot be analyzed automatically. Programmers must next resort to the time consuming task of examining the program states in a sequence of computational events to find the source of their errors. Debuggers aid programmers by letting them stop the program at a particular event and query the program state. Single-language debuggers limit the scope of events and states to a particular language while multilingual debuggers do not have such limitation. Multilingual debuggers are strictly desirable, but their construction costs are quite prohibitive.

To substantially reduce this cost, this chapter introduces a novel composition approach to building mixed-environment multilingual debuggers. Our approach uses an intermediate agent that interposes on language transitions, controlling and reusing single-environment debuggers. We implement debugger composition in *Blink*, a debugger for Java, C, and the Jeannie programming language. We show that Blink is (1) relatively simple: it requires modest amounts of new code; (2) portable: it supports multiple Java Virtual Machines, C compilers, operating systems, and component debuggers; and (3) powerful: composition eases debugging, and supports new mixed-language expression evaluation and Java Native Interface (JNI) bug diagnostics.

Section 4.1 starts by describing the compositional approach for reducing the engineering cost of building portable mixed-environment debuggers. We support the feasibility of this approach in Section 4.2 by building *Blink*. In Section 4.3, we extend the state of the art by evaluating mixed language expressions written in Jeannie [35]. Section 4.4 validates that *Blink* is simple, portable, and powerful. Section 4.5 discusses how the compositional approach generalizes to more languages and environments. Section 4.6 presents our extension to Jeannie.

4.1 Debugger Composition

This section describes our approach to composing mixed-environment debuggers out of single-environment debuggers. We use our implementation of Blink for Java and C as our running example. Section 4.5 presents requirements and mechanisms for generalizing composition to other mixed-language environments.

4.1.1 Debugger Features

Our goal is to provide all the standard debugging features in a mixed environment. When a user debugs a program, she wants to find and correct a defect that results in erroneous data or control flow, which leads to erroneous output or a crash [89]. Rosenberg identifies three essential features in support of this quest [63]:

Execution control: The debugger controls the execution of the debuggee process by starting it, halting it at breakpoints, single-stepping through it, and eventually tearing it down. Typical interactive commands are

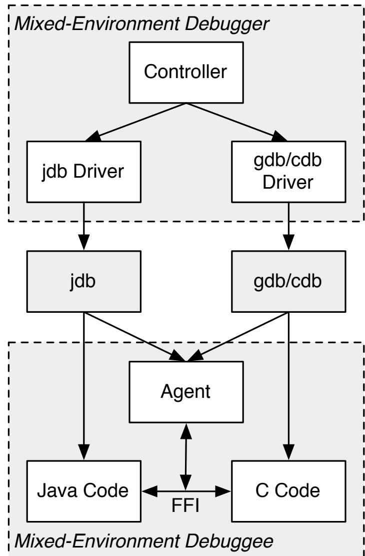


Figure 4.1: Agent-based debugger composition approach.

`run`, `break`, `step`, `continue`, and `exit`.

Context management: The debugger keeps track of where in the code the debuggee process is and, on demand, reports source code listings and call stack traces. Typical interactive commands are `list` and `backtrace`.

Data inspection: Users query the debugger to inspect data with source language expressions, such as `print` or `eval`.

4.1.2 Intermediate Agent

Our approach to implementing these standard debugger features for a mixed environment is to compose single-environment debuggers through an intermediate agent. The mixed-environment debugger consists of a controller and one driver for each single-environment component debugger. Figure 4.1 illustrates this structure for the case of Java and C using `jdb` for Java, and `gdb` or `cdb` for C (depending on whether we run on Linux or Windows). The debuggee process runs both Java and C, and the intermediate agent coordinates the debuggers. The intermediate agent has two complementary responsibilities:

Language transition interposition: When the debuggee switches environments on its own, the agent alerts the corresponding single-environment debugger, so this debugger can track context or take over if necessary.

Debugger context switching: When an interactive user command requires the debugger to switch environments, the agent transitions the debuggee into the appropriate state and issues the command to the appropriate single-environment debugger.

The following subsections detail the agent responsibilities and how to satisfy them.

4.1.3 Language Transition Interposition

Language transition interposition is required for execution control because otherwise single-stepping is incomplete. Consider a Java and C debuggee suspended at a Java breakpoint: the Java debugger is in charge, and the C debugger is dormant. A single-step on a return statement to C causes a language transition to C. The agent must detect this transition because otherwise the Java debugger waits for control to return to Java code while the C debugger remains dormant.

Language transition interposition is also required for context management because otherwise stack traces are incomplete. Language transitions result in different portions of the stack belonging to different environments, but each single-environment debugger understands only the portions corresponding to its own language. To prepare for reporting the entire mixed-language stack, the agent must track all the seams.

The agent must capture all environment transitions, whether they are debuggee- or user-initiated. With two languages, there are four kinds of local transitions: mixed-language calls and returns (e.g., Java call to C, C call to Java, Java return to C, and C return to Java). The agent must also capture non-local control flow such as exceptions.

Our approach instruments all environment transitions to call agent code. For instance, in Figure 4.1, we interpose on transitions between Java and C code, instrumenting them to call the agent. One option for realizing this instrumentation is to modify the compiler or interpreter. However, to achieve portability across different JVMs and C compilers, we do not want to modify them. Instead, we leverage the fact that Java's foreign function interface (FFI) is wrapper-based and instrument the wrappers.

4.1.4 Debugger Context Switching

When one single-environment debugger is active and the user issues a command that only the other debugger can perform, the agent must assist in debugger context switching. For example, when the program is at a breakpoint in Java and the user wants to set a breakpoint in C, the agent must suspend the Java debugger and issue the command to the C debugger. Similarly, commands such as `backtrace` and `print` require one or more context

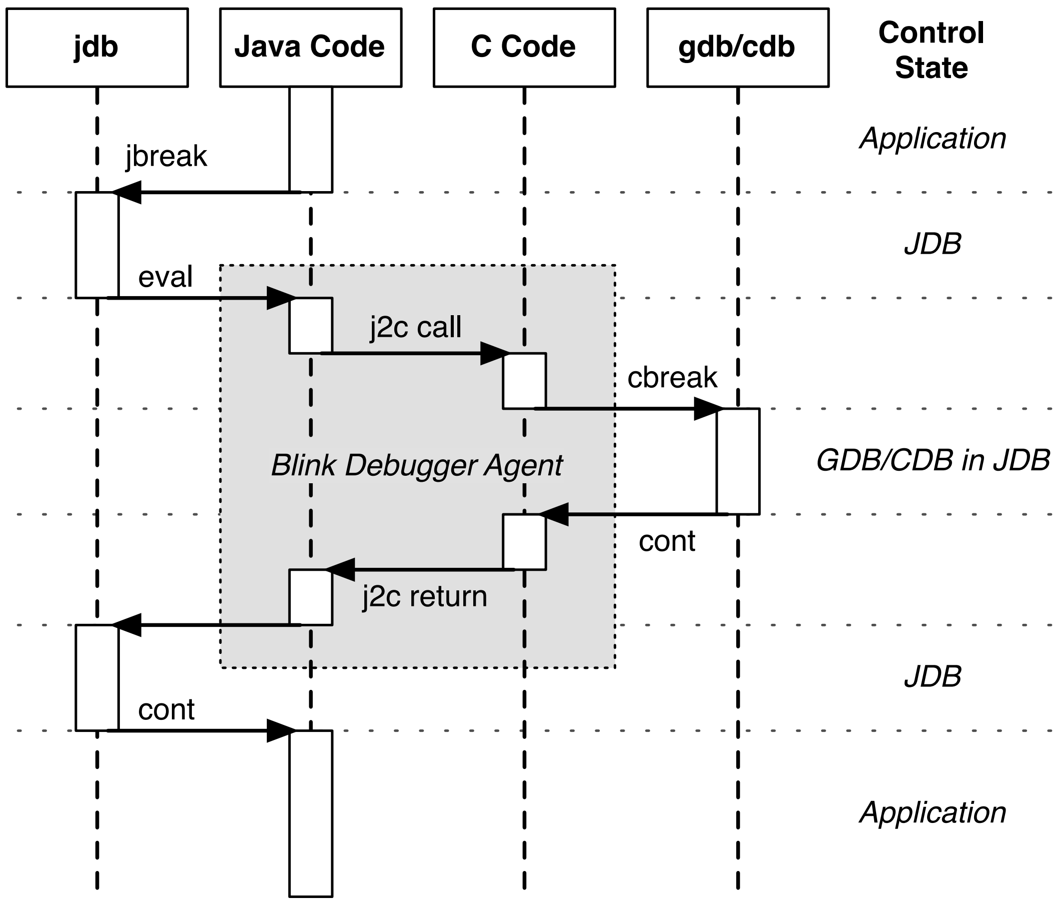


Figure 4.2: Debugger context switching example, using `j2c` helper function to switch from `jdb` to `gdb/cdb`. Blink also has a `c2j` helper function for switching in the other direction.

switches to tap into functionality from both single-environment debuggers. We switch debugger contexts with the following steps:

1. Set a breakpoint in a helper function in the other environment.
2. Call the helper function using expression evaluation.
3. At the breakpoint, activate the other debugger.
4. When the other debugger completes, return from the helper function, which returns control back to the original debugger.

Figure 4.2 illustrates context switching through the example of switching from `jdb` to `gdb`. Each vertical line represents an execution context, with the currently active context marked by a box overlaying the line. Horizontal arrows show control transfers between execution contexts. From top to bottom, the application starts out executing Java code. It hits a Java breakpoint, suspends itself, and activates `jdb`. Now, suppose the user requests a `gdb` debug action. At the moment, `gdb` is inactive and cannot accept user commands. Blink therefore initiates a debugger context switch by using the `jdb` function evaluation feature to call the debugger agent method `j2c`. The method `j2c` is a Java method that uses JNI to call C and has a breakpoint in the C part of the code. When execution hits the C breakpoint, `gdb` is activated and can perform the debug action requested by the user. When complete, `gdb`'s `continue` returns from the C code and Java method, at which point `jdb` wakes up again and is ready to accept commands. The user can either request additional debugging actions in Java or C or resume normal application execution with `continue`.

4.1.5 Soft-Mode Debugging

Debugger composition dictates *soft-mode debugging*, in which the debuggee process executes basic commands, such as `break`, `step`, and `backtrace`, on behalf of the debugger. In contrast, *hard-mode debugging* does not require the debuggee to run code on the debugger's behalf, except when users explicitly request it — for example, with a command to evaluate a function call. Debuggers for C, including `gdb` and `cdb`, are typically hard-mode. Java debuggers are typically soft-mode because Java's JDWP (Java Debugger Wire Protocol) expects an agent in the JVM that issues commands to the debuggee.

Soft-mode debugging is less desirable than hard-mode because running code in the debuggee changes its state and behavior and may thus lead to Heisenberg effects. The very act of debugging may change the behavior of the bug. Notably, the user may set a breakpoint in a C library shared by the application and JVM. The user expects to reach the breakpoint through a JNI call, but JVM code may instead reach the breakpoint through internal service code. Since the JVM is typically not reentrant (i.e., it assumes that no user code runs in the middle of a JVM service), debugger actions may now crash the JVM. For example, the JVM's allocator may temporarily leave a data structure in an inconsistent state, thus making it unsafe for the agent to allocate objects. Furthermore, even if the native breakpoint is not reachable from the JVM, JNI disallows JNI operations when exceptions are pending or garbage collection is disabled. Reentering the JVM without first clearing the exception or re-enabling garbage collection may crash or deadlock the system [43, 49].

Blink mitigates its use of soft-mode debugging by warning users of actions that might trigger a soft-mode inconsistency. Debugging actions in C are safe as long as the program entered native code through JNI, exceptions are cleared, and garbage collection is enabled. Since we already rely on language interposition, we detect whether the JVM is in a safe state. If the debugger is about to perform an action in C, but the JVM is in an unsafe state, the debugger warns the user. Instead of just warning the user, we could refuse to perform debug actions altogether. We chose a warning over refusal since unreliable information is better than no information.

4.2 Blink Implementation

This section explains Blink's implementation in detail.

4.2.1 Blink Debugger Agent

The Blink debugger agent is a dynamically linked library that includes both Java and native code and that executes within the JVM hosting the application. The host JVM loads and initializes the Blink agent using the Java Virtual Machine Tool Interface (JVMTI) [72]. Blink triggers single-environment debugger actions using their expression evaluation features. As far as the component debuggers are concerned, these actions are initiated by the application process.

Debugger context switching. Blink supports switching contexts between its component debuggers as illustrated in Figure 4.2. The helper functions `j2c` and `c2j` are part of the Blink debugger agent, and they contain hard-coded internal breakpoints. These internal breakpoints force the application to surrender control to the respective debugger.

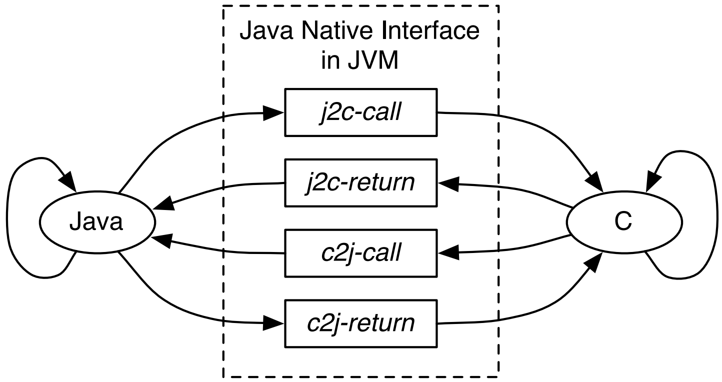


Figure 4.3: Transitions between Java and C.

Runtime transition interposition. The Blink agent interposes on all environment transitions to report full mixed run-time stack traces and to control single-stepping between environments. Figure 4.3 shows the four possible transitions between Java and C. Java exceptions are automatically propagated by JNI, and thus they do not result in additional environment transitions.

j2c call: Line 8 in Figure 4.4 is an example of a call from Java to C. It looks just like an ordinary method call, and in fact, with virtual methods, the same call in the source code may invoke native methods or Java methods. To interpose on j2c calls, the Blink agent wraps all JNI native methods. For example, the wrapper function for the native method `PingPong_cPong` on Line 14 in Figure 4.4 conceptually reads:

```
jint wrapped_PingPong_cPong(...) {
  j2c_call(); /* interposed j2c call */
  jint result = PingPong_cPong(...);
  j2c_return(); /* interposed j2c return */
  return result;
}
```

PingPong.java

```java
class PingPong {
  static { System.loadLibrary("PingPong"); }
  public static void main(String[] args) {
    jPing(3);
  }
  static int jPing(int i) {
    if (i > 0)
      cPong(i - 1);
    return i;
  }
  static native int cPong(int i);
}
```

PingPong.c

```c
#include <jni.h>
jint PingPong_cPong(
  JNIEnv* env, jclass cls, jint i
) {
  if (i > 0) {
    jmethodID mid = (*env)->GetStaticMethodID(
      env, cls, "jPing", "(I)I");
    (*env)->CallStaticIntMethod(env,cls,mid,i-1);
  }
  return i;
}
```

Figure 4.4: JNI mutual recursion example.

Wrappers are largely generic — i.e., they pass arguments to and results from the original native method implementation while also invoking the debugger agent. For this reason, Blink uses assembly code templates to instantiate each native method's wrapper. This approach is simple and general — i.e., does not require the full power of dynamic code generation. However, it does require some porting effort across architectures and operating systems. In our experiences with IA32 and PowerPC for Unix and Windows, the non-portable code amounts to only 10–20 lines of assembly.

j2c return: The Blink agent interposes on returns from a C function to a Java method through the JNI native method wrapper function shown above. The return looks just like an ordinary function return, and, in fact, the same C function can return sometimes to Java and sometimes to C.

c2j call: All calls from C to Java go through a JNI interface function, such as `CallStaticIntMethod` on Line 19 of Figure 4.4. Blink instruments every c2j interface function. All interface functions reside in a struct of function pointers pointed to by variable `JNIEnv* env` on Line 15 of Figure 4.4. During JVMTI initialization, Blink replaces the original function pointers by pointers to wrappers. Conceptually, the wrapper for `CallStaticIntMethod` reads:

```
int wrapped_CallStaticIntMethod(...) {
  c2j_call(); /* interposed c2j call */
  int result = jvm_CallStaticIntMethod(...);
  c2j_return(); /* interposed c2j return */
  return result;
}
```

c2j return: The same wrappers that interpose on c2j calls also interpose on c2j returns, as shown above.

4.2.2 Context Management

One basic debugger principle from Rosenberg's book [63] is: "Context is the torch in the dark cave." Users, unable to follow all the billions of instructions executed by the program, feel like they are being taken blindfolded into a dark cave when searching for the source of a bug. When the program hits a breakpoint, the debugger must provide context.

Source line number information. The most important question in debugging is: "Where am I?" Debuggers answer it with a line number. Java compilers provide line number information to `jdb`, and C compilers provide line number information to `gdb` or `cdb`, which Blink borrows.

Calling context backtrace. While "Where am I?" is the most important question, "How did I get here?" is a close second. Debuggers answer this question with a calling context backtrace, which shows the stack of function calls leading up to the current location. The JNI code in Figure 4.4 is an example of mixed-runtime calls that produce a mixed stack. In the beginning, the `main` method on Line 4 calls the `jPing` method with argument `3`, yielding the following stack:

`main:4` $\rightarrow$ `jPing(3):7`

Since `i` > 0, control reaches Line 8, where the Java method `jPing` calls native method `cPong` defined in C code as function `PingPong_cPong`:

main:4 → jPing(3):8 → cPong(2):17

The C function `cPong` calls back into Java method `jPing` by first obtaining its method ID on Line 18, then using the method ID in the call to `CallStaticIntMethod` on Line 19:

main:4 → jPing(3):8 → cPong(2):19 → jPing(1):7

Finally, after one more call from `jPing` to `cPong`, the mixed-environment mutual recursion comes to an end as it reaches the base case `i` = 0:

main:4 → jPing(3):8 → cPong(2):19 → jPing(1):8
→ cPong(0):17

At this point, the stack contains multiple and alternating frames from each environment. Unfortunately, each single-environment debugger only knows about a part of the stack since each environment uses its own calling convention. For example, a standard Java debugger shows all Java fragments, with gaps for the C parts of the stack:

main:4 → jPing(3):8 → *?(C)* → jPing(1):8 → *?(C)*

A standard C debugger has even less information. It only shows the bottom-most C fragment:

?(Java/C) → cPong(0):17

Neither `gdb` nor `cdb` understands the JVM implementation details for Java frames.

Blink weaves the complete stack from JVM call frames and native method frames by exploiting the Java native method wrappers discussed in

Section 4.2.1. The `j2c` wrapper saves its frame pointer and program counter in a thread local variable, and the `c2j` wrapper retrieves the saved frame pointer and program counter while also overwriting its old frame pointer and return address. Modifying the processor state accordingly guides the C debuggers to skip JVM-specific native frames between j2c and c2j wrappers and yields the following C frames:

`cPong(2):19` → `wrapped_CallStaticIntMethod`
→ `wrapped_PingPong_cPong` → `cPong(0):17`

Blink recognizes its agent wrapper functions and presents the interleaved Java and C stack:

`main:4` → `jPing(3):8` → `cPong(2):19` → `jPing(1):8`
→ `cPong(0):17`

Blink thus recovers and reports the full stack to the user as needed. These implementation details will vary for other languages, their environments, and their debuggers. As described below, the user can also inspect data from both languages at a breakpoint.

4.2.3 Execution Control

If context is the torch in the dark cave, then execution control is the means by which the user can get from point A to B in the cave when tracking down a bug. The debugger controls execution by starting up, tearing down, setting breakpoints, and stepping through program statements based on user commands.

Start-up and tear-down. The Blink controller starts the program in the JVM, attaches `jdb` and either `gdb` or `cdb`, and loads the Blink debugger agent. To load the agent, Blink uses JVMTI and the `-agentlib` JVM command line argument. To initialize the agent, Blink issues internal commands, such as setting two internal breakpoints: one for Java and the other for C.[1] After it initializes and connects all the processes but before the user program commences, Blink gives the user a command prompt. When the program terminates, Blink tears down `jdb` and `gdb`/`cdb` and exits.

Breakpoints. Breakpoints answer the question: "How do I get to a point in program execution?" Users set breakpoints to inspect program state at points they suspect may be erroneous. The debugger's job is to detect when the breakpoint is reached and then transfer control to the user. One of the key challenges for a mixed-environment debugger is setting a breakpoint for a location in an inactive environment. This functionality requires the debugger to transfer control to the other environment's debugger, set the breakpoint, and return control to the current environment's debugger. Blink takes the breakpoint request from the user and checks if the request is for Java or C. If the current environment does not match the breakpoint environment, Blink switches the debugging context to the target environment and directs the breakpoint request to the corresponding debugger.

Single stepping. Once the application reaches a breakpoint, the question is: "What happens next?" Users want to single step though the program,

[1]The internal breakpoints are multiplexed for several conditions. See Section 4.4.3 for the performance impact of evaluating these conditions.

examining control flow and data values to find errors. The *step into*, or simply `step`, command executes the next dynamic source line, which may be the first line of a method call, whereas the *step over*, or `next`, command treats method calls as a single step. The challenge for mixed-environment single-stepping is that while `jdb` can step through Java and `gdb` or `cdb` can step through C, they lose control when stepping into a call to the other environment or when returning to a caller from the other environment.

Blink maintains control during a step command as follows. It sets internal breakpoints at all possible language transitions, so if the current component debugger loses control in a single-step, then the other component debugger immediately gains control. Blink only enables transition breakpoints from the current environment to the other environment when the user requests a single-step. Furthermore, when the user requests step-over as opposed to step-into, Blink enables return breakpoints, as opposed to both call and return breakpoints. Note that Blink does not make any attempts to decode the current instruction but rather aggressively sets needed internal breakpoints just in case the single-step causes an environment transition and then operates on the user command. This approach greatly decreases debugger development effort since accurate Java single-stepping requires interpreting the semantics of all byte codes, and accurate C single-stepping requires platform-dependent disassembly.

Once Blink sets the internal breakpoints, it implements single-stepping by issuing the corresponding command to `jdb` or `gdb`/`cdb`. There are three possible outcomes:

- The component debugger's single-step remains in the same environment. Blink performs no further action.

- There is an environment transition and consequently an internal breakpoint intercepts it. Blink steps from the internal breakpoint to the next line.

- An exceptional condition, such as a segmentation fault, occurs. Blink abandons single stepping.

In all cases, Blink then disables its internal breakpoints, as is usual for breakpoint algorithms [63].

4.2.4 Data Inspection

Once the user arrives at an interesting point, the main question becomes: "Is the current state correct or infected?" This question is hard to answer automatically, so data inspection answers the simpler question, "What is the current state?" Blink delegates the inspection of application variables, including locals, parameters, statics, and fields, to the component debugger for the current environment, which provides the most local origin for a variable. If, however, the current component debugger does not recognize the variable, Blink tries the other component debugger.

4.3 Jeannie Mixed-Environment Expressions

The more powerful a debugger's data inspection features, the easier it is for the user to determine whether or not she is on the right track to finding a bug. For example, `gdb` provides expression evaluation with a read-eval-print loop (REPL). An interactive interpreter evaluates arbitrary source language expressions based on the current application state. While implementing a language interpreter requires a significant engineering effort, expression evaluation makes it easier to determine whether or not the current state is infected, especially if the evaluator supports function calls and side effects. Besides debugging, expression evaluation is useful for rapid prototyping, program understanding, and testing, as users of languages with REPLs readily attest.

Blink advances the state of the art of expression evaluation by accepting mixed-language expressions, which nest subexpressions from multiple languages with a language specification operator. The user writes mixed-language expressions, and we implement mixed-environment interpretation. It is based on the insight that, given single-environment interpreters, mixed-environment expression evaluation reduces to handing off subexpressions to the component debuggers and passing intermediate results between them.

Blink implements mixed-language expressions written in the Jeannie programming language syntax [35], which mixes Java and C code using the incantation "backtick period language," i.e., `` `.C `` and `` `.Java ``. A single backtick `` ` `` toggles when there are only two languages, as in Blink. For example, consider this native Java method declaration from the BuDDy binary decision diagram library [50]:

```
public static native int makeSet(int[] var);
```

The C function implementing this Java method looks as follows:

```
jint BuDDyFactory_makeSet(
  JNIEnv *env, jclass cls, jintArray arr
) {
  ... /* C code using parameters through JNI */
}
```

In the C function, the variable `arr` is an opaque reference to a Java integer array. Single-language expression evaluation could only print its address, which is not helpful for debugging. However, the mixed-environment expression `` `.C((`.Java arr).length) `` (or `` `((`arr).length) `` for short) changes to the Java language and accesses the Java field `length` of the C variable `arr`, returning the length of the Java array, which is much more meaningful to the user. Clearly, mixed-environment expression evaluation makes data inspection more convenient.

We add two features to Blink's debugger agent to support expression evaluation:

Convenience variables store the results of a (sub)expression evaluation in temporary variables.

Mixed-environment data transfer translates and transfers data between environments.

4.3.1 Convenience Variables

Application variables are named locations in which application code stores data during execution. Convenience variables are additional named locations provided by the debugger, in which the user interactively stores data for later use in a debugger session. Convenience variables behave like variables in many scripting languages: they are implicitly created upon first use, have global scope, and are dynamically typed. In addition to user-defined convenience variables, some debuggers support internal convenience variables — for example, to hold intermediate results during expression evaluation. In the mixed-environment case, the debugger must remember not only the values of convenience variables, but also their languages. Since `gdb` provides convenience variables (written "`$var`"), Blink reuses them to store C values. Since `jdb` and `cdb` lack this feature, Blink implements convenience variables in the debugger agent, using a table to map names to values and languages. The table is polymorphic to support dynamic typing.

4.3.2 Mixed-Environment Data Transfer

Mixed-environment data transfer is the only case where Blink must discover enough type information to treat the value appropriately, since the single-language debuggers usually perform this function. The Blink agent transfers data from a source to a target environment by first storing data in an array in the source environment. It then uses a helper Java method or JNI function to read from the array and returns the value to the target environment. One complication is that the array and the retrieval function must have the correct type since the semantics of a value depend on its type and language. For example, Blink must convert an opaque JNI reference in C to a pointer in Java. A struct or union in C, on the other hand, does not have a direct correspondence in Java. In the case of C values, `gdb` provides exactly what Blink needs: the `whatis` command finds the type of an expression without executing it and, in particular, without causing any side effects or exceptions. Since `jdb` lacks the necessary functionality, Blink distinguishes between different Java types for primitive values, such as numbers, characters, or booleans, and for references (i.e., objects or arrays) using a simple workaround. Blink instructs `jdb` to pass the value to a helper method that is overloaded for the different primitive and reference types. `Jdb`'s expression evaluation automatically selects the appropriate method, thus ensuring that

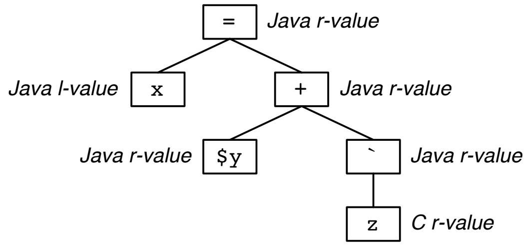


Figure 4.5: Reading the expression `x = $y + `z` when the current language is Java.

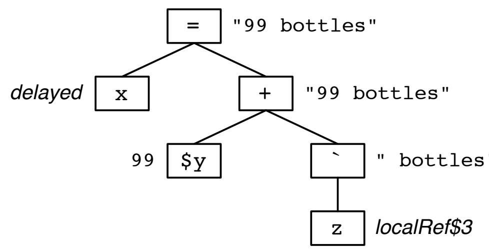


Figure 4.6: Evaluating the expression `x = $y + `z` when the current language is Java.

values can be correctly transferred to C.

4.3.3 Expression Evaluation (REPL)

This section explains each step of Blink's read-eval-print (REPL) loop.

Read. As suggested by Rosenberg [63], the "read" stage of Blink's REPL reuses syntax and grammar analysis code. We reuse the Jeannie grammar, which composes Java and C grammars [31, 35]. It is written in *Rats!*, a parser generator that uses packrat parsing for expressiveness and performance. The Jeannie grammar and *Rats!* are designed for composition. Section 4.6 discusses Jeannie in more detail.

Whereas a traditional compiler annotates the AST with types, Blink annotates the AST with: (1) which language (Java or C) is being used and (2) whether each AST node is an r-value (read-only) or an l-value (written-to on the left-hand side of an assignment). Figure 4.5 shows how Blink annotates the AST for the expression "`x = $y + `z`," assuming that the current language is Java. Node `x` is an l-value, and node `z` is a C r-value because `z`'s parent is the language toggle backtick `.

Blink uses the component debuggers for symbol resolution. As is usual in debuggers, application symbols such as variable and function names are resolved relative to the current execution context. User convenience variables, on the other hand, have global scope and do not require context-sensitive lookup.

Eval. The interpreter visits the AST in depth-first left-to-right post-order. Each node is executed exactly once and in the right order to preserve language semantics in the presence of side effects and to avoid surprising users if an exceptional condition, such as a segmentation fault, cuts expression evaluation short.

To evaluate an expression one AST node at a time, Blink uses temporary storage for subexpression results. For r-values, Blink evaluates the node and then stores the result in an internal convenience variable. For l-values, Blink evaluates their children but delays their own evaluation. These l-values are evaluated later as part of their parent, which is by definition an assignment. Figure 4.6 shows the example expression "`x = $y + `z,`" assuming that the convenience variable `$y` is currently the number `99`, and the C application variable `z` is currently an opaque JNI local reference `localRef$3`.

All leaves are variables, which Blink evaluates through the component debuggers' REPL. Blink directly uses any leaf literals without lookup. At inner nodes, Blink needs to perform evaluation actions. For the language toggle operator `, Blink performs a mixed-environment data transfer as described in Section 4.3.2. As shown in Figure 4.6, Blink discovers that the JNI reference `localRef$3` on the C side refers to the Java string `bottles` on the Java side. For other operators, such as + and =, Blink falls back on the REPL in the component debuggers. Note that in general, an inner node may call a user function and may thus execute arbitrary user code.

Print. When expression evaluation reaches the root of the tree, Blink prints the result. As recommended by Rosenberg, Blink disables user breakpoints for the duration of expression evaluation because the user would probably be surprised when expression evaluation hits a breakpoint in a callee [63]. However, there may be other exceptional conditions during expression evaluation, such as Java exceptions or C segmentation faults. In this case, Blink aborts the evaluation of the current expression, and the debug session continues at the fault point instead. Whether expression evaluation terminates normally or abnormally, Blink always nulls out internal convenience variables for sub-results and re-enables all user breakpoints.

4.4 Evaluation

This section evaluates our claim that debugger composition is an economical way to build mixed-environment debuggers and that the resulting debuggers are powerful. We show that Blink is relatively concise, new development cost is low, the space and time overheads are low, and the resulting tool is portable. Through the use of case studies, Section 4.4.4 demonstrates that Blink helps programmers to quickly find mixed-language interface bugs.

4.4.1 Methodology

We rely on single-environment debuggers, JVMs, C compilers, and operating systems. We use JDK 1.6 as implemented by Sun and IBM. For the debuggee running on Linux/IA32 machines, we use Sun's Hotspot Client 1.6.0_10 [71] and IBM's J9 1.6.0 (build pxi3260-20071123_01) [6]. We also use Sun's `javac` 1.6.0_10 and `gcc` 4.3.2 with the `-g` option. For Windows, we use Sun's Hotspot Client 1.6.0_10, Sun's `javac` 1.6.0_10, and Microsoft's C/C++ compiler (`cl.exe`) 15.00.21022.08. We use Sun's JDK 1.6.0 `jdb` and Microsoft's `cdb` 6.9.0003.113 debuggers, and GNU `gdb` 6.8 debugger running on Cygwin 1.5.25, a Unix compatibility layer for Windows.

4.4.2 Building Blink

Blink's modest construction effort leverages the large engineering effort and supported platforms of existing single-environment debuggers. To quantify this claim, we count non-blank non-commenting source lines of code (SLOC), which is an easily available but imperfect measure of the effort to develop and maintain a software package. Given the orders of magnitude differences in SLOC, we are confident that this metric reflects substantial differences in engineering effort.

4.4.2.1 Construction Effort

Table 4.1 shows the code sizes of Blink, `jdb`, `gdb`, and their components. The `jdb` line counts are for the `jdb` 1.6 sources in `demo/jpda/examples.jar`

Debugger	SLOC	#Files
Blink	9,481	41
Controller (front-end)	4,575	18
jdb driver (back-end)	391	1
gdb driver (back-end)	511	1
cdb driver (back-end)	546	1
Agent - Java (back-end)	1,515	9
Agent - C (back-end)	1,943	11
Java debugger - `jdb`	86,579	769
jdb (user-interface)	18,360	122
JDI (front-end)	16,983	256
JDWP Agent (back-end)	40,171	356
JVMTI (back-end)	11,065	35
C debugger - `gdb` 6.7.1	1,017,069	2,331
gdb	419,921	1,524
include	32,039	215
bfd	286,981	398
opcodes	278,128	194

Table 4.1: Debugger SLOC (source lines of code).

of Sun's JDK 1.6.0-b105. The JDI line counts are for the JDI implementation in the Eclipse JDT. The JDWP and JVMTI line counts are for the corresponding subdirectories of the Apache DRLVM. Blink adds a modest 9,481 SLOC to integrate 1,103,648 SLOC from the Java and C debuggers. The SLOC of the existing debugger packages are 9 to 107 times larger than Blink's. Although other researchers show how to build single-environment debuggers more economically than `gdb` [61, 64], Blink adds modestly to this effort. Blink only adds new code for interposing on environment transitions and for controlling the individual debuggers. Blink otherwise reuses existing debuggers for intricate platform-dependent features, such as instruction decoding for single-stepping or code patching for breakpoints.

4.4.2.2 Portability

To evaluate the effort required for porting Blink to multiple platforms, we measure the amount of platform-independent and -dependent code.

The basic composition framework requires 4,575 SLOC. Blink needs an additional 4,265 SLOC to support our initial configuration, which uses Sun's Hotspot JVM, `jdb`, and `gdb` running on Linux/IA32. Out of Blink's total 9,481 SLOC, approximately 1,500 SLOC implement platform-specific code in the agent and debugger drivers, representing about 16% of Blink's code base. Our native agent contains a small amount of non-portable platform- and ABI-specific code to access the native call stack. Furthermore, a small amount of debugger-specific code is required because `cdb` exposes a different user interface than the more expressive `gdb`. Consequently, Blink employs an internal adaptation layer to provide uniform access to either `gdb` on GNU platforms or `cdb` on Windows.

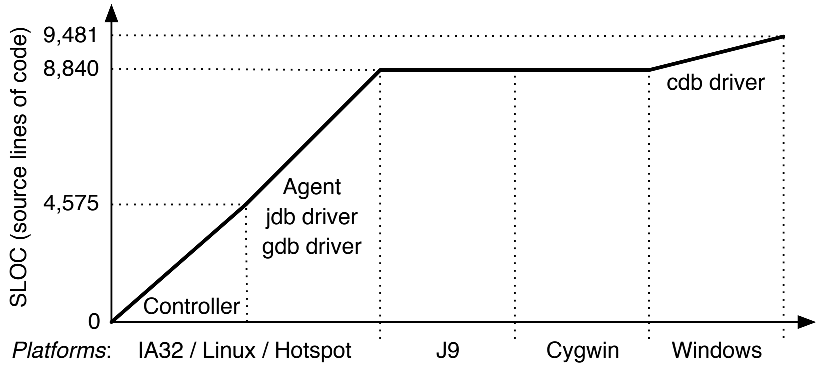


Figure 4.7: Blink portability and SLOC.

Figure 4.7 plots the cumulative SLOC for the Blink controller; then the code for supporting Hotspot, `jdb`, and `gdb` on Linux; then the code for supporting J9 on Linux; then `gcc` and `gdb` under Cygwin on Windows; and finally Microsoft's C and `cdb` on Windows. As shown in the figure, Blink requires no additional code to support IBM's J9 and Cygwin. Furthermore, it requires only 640 SLOC to support `cdb` on Windows. These results show that Blink's debugger composition is effective and requires only small amounts of code when adding more operating systems, JVMs, C compilers, and component debuggers.

4.4.2.3 Portability Tests

We now briefly describe some of our functionality tests. They give us confidence that our implementation is correct and complete on all supported platforms.

Context management. This test sets two breakpoints, at `jPing(PingPong.java:7)` and `cPong` (`PingPong.c:17`) in Figure 4.4. During execution, the application stops at each of these breakpoints twice, and, each time, the test issues the `backtrace` command.

Execution control. This test first sets a breakpoint at the `main` method of the mutual recursion example in Figure 4.4. From there, the test repeatedly uses the `step` command until the end of the program. This test exercises all cases of mixed-language stepping through calls and returns.

Data inspection. This test first sets a breakpoint in a nested context of two example programs in the Blink regression test suite. (The interested reader can find these programs in the open-source distribution of Blink [30].) When the application hits the breakpoint, the test evaluates a variety of expressions, covering primitive and compound data, pure expressions and assignments, language transitions, and user function calls.

Results. Currently, all these and other functionality tests succeed for the following configurations on IA32:

$$\left\{ \begin{array}{l} \text{Sun JVM} \\ \text{IBM JVM} \end{array} \right\} + \left\{ \begin{array}{l} \text{Linux} \\ \text{Cygwin} \end{array} \right\} + \texttt{gdb}$$

The "Cygwin" case uses Windows with the GNU C compiler instead of Microsoft's C compiler. We also tried the tests on PowerPC but found that `gdb` did not interact well with the JVM on that platform. Using a Linux/Power Mac G4 machine running IBM JDK 1.6.0 (SR1) and `gdb` 6.8, `gdb` reports an illegal instruction signal (`SIGILL`) when the debuggee resumes execution after a breakpoint in a shared library. We leave further investigation of different architectures to future work. We also test Blink with Microsoft's C compiler and Microsoft's C debugger:

Sun JVM + Windows + `cdb`

In this configuration, context management and execution control are fully supported, but data inspection is only partially supported because `cdb`'s expression evaluation features are incomplete when compared to `gdb`.

4.4.3 Time and Space Overhead

This section shows that the time and space overheads of Blink's intermediate agent are low.

Time Overhead. The time overhead of the intermediate agent is linearly proportional to the number of dynamic transitions between Java and C since it installs wrappers in both Java native methods and JNI functions. These wrappers add a small number of instructions to the dynamic instruction stream for each transition between Java and C.

To measure the performance impact of interposition in the intermediate agent, we ran several large Java programs with the Blink agent. We measured runtime and dynamic transition counts with Sun Hotspot 1.6.0_10 running on a Linux/IA32 machine on the SPECjvm98 and DaCapo Java v.2006-10 Benchmarks [9, 68]. These Java benchmarks exercise C code inside the standard Java library. The initial heap size was 512MB, and the maximum heap size was 1GB. The experiments used a Pentium D 2 GHZ running Linux 2.6.27. Each benchmark iterated once. The results are the median of 16 trials.

Table 4.2 shows the results. The column *environmental transition counts* shows the number of dynamic transitions between Java and C. The

Benchmark	**Environmental transition counts**			**Execution time in seconds**				**Normalized execution time**		
	Java → C	**Java ← C**	**Java ↔ C**	**Base**	**Active JVMTI**	**Interposed transitions**	**Checked transitions**	**Active JVMTI**	**Interposed transitions**	**Checked transitions**
antlr	221,309	249,411	470,720	4.58	4.41	4.64	4.65	0.96	1.01	1.02
bloat	594,644	233,795	828,439	8.50	8.48	9.32	9.41	1.00	1.10	1.11
chart	346,317	677,240	1,023,557	9.28	9.17	9.90	9.87	0.99	1.07	1.06
eclipse	2,631,281	6,206,930	8,838,211	50.70	50.88	59.76	58.69	1.00	1.18	1.16
fop	540,899	1,439,441	1,980,340	3.74	3.88	4.25	4.31	1.04	1.14	1.15
hsqldb	130,959	73,750	204,709	5.61	5.65	5.76	5.78	1.01	1.03	1.03
jython	13,525,019	42,859,171	56,384,190	11.83	11.66	12.27	12.37	0.99	1.04	1.05
luindex	441,090	936,565	1,377,655	9.28	9.35	9.94	10.01	1.01	1.07	1.08
lusearch	2,015,481	1,513,508	3,528,989	8.34	9.17	9.89	10.05	1.10	1.19	1.21
pmd	531,579	436,124	967,703	8.45	8.58	9.05	9.09	1.02	1.07	1.08
xalan	769,991	362,868	1,132,859	19.10	19.54	22.34	22.27	1.02	1.17	1.17
compress	5,958	9,960	15,918	3.71	3.73	3.96	4.00	1.01	1.07	1.08
jess	92,272	62,917	155,189	2.63	2.56	3.23	3.16	0.97	1.23	1.20
raytrace	18,170	12,375	30,545	1.44	1.43	1.64	1.68	0.99	1.14	1.17
db	53,225	80,733	133,958	9.54	9.57	9.72	9.74	1.00	1.02	1.02
javac	184,566	71,972	256,538	6.54	7.28	7.46	7.30	1.11	1.14	1.12
mpegaudio	25,733	21,588	47,321	2.81	2.81	2.77	2.79	1.00	0.99	0.99
mtrt	18,784	13,427	32,211	1.80	1.72	2.01	2.02	0.96	1.12	1.12
jack	418,681	886,216	1,304,897	3.90	3.87	4.22	4.38	0.99	1.08	1.12
GeoMean								**1.01**	**1.09**	**1.10**

Table 4.2: Performance characteristics of the Blink debug agent with Hotspot VM 1.6.0_10.

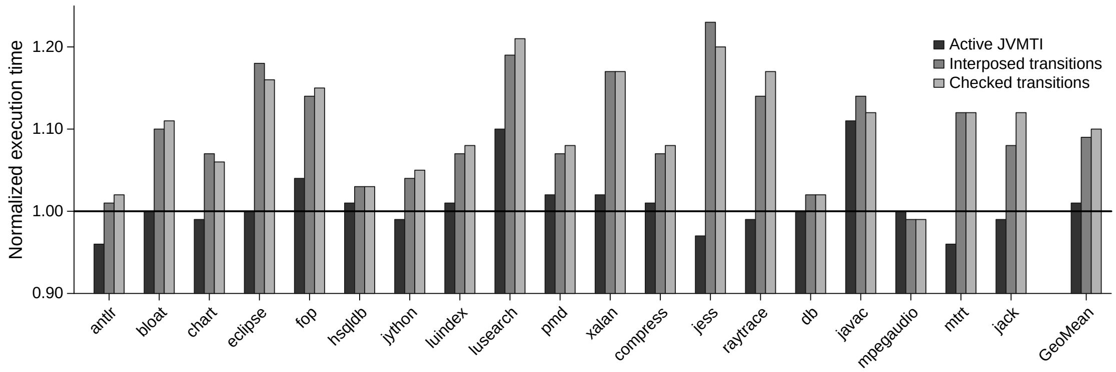


Figure 4.8: Time overhead of the Blink debug agent with Hotspot VM 1.6.0_10. Note the vertical axis starting at 0.9.

following columns show execution times in the four configurations—*Base*, *Active JVMTI*, *Interposed transitions*, and *Checked transitions*—and normalized execution times for the debugger configurations. The Base configuration represents production runs without any debugging-related overhead. In contrast, the fully functional agent needs to activate JVMTI, interpose transitions, and check transitions.

Figure 4.8 illustrates Blink's runtime normalized to the production runs. JVMTI, interposition, and transition checking add 1%, 8%, and 1% overhead, respectively. There are a few counter-intuitive speedups because the JIT and GC add non-determinism to the runtime. On average, Blink's total overhead is 10%. Figure 4.9 shows that the overhead is sub-linear to the total dynamic transition counts. Although the agent overhead is linearly proportional to the dynamic transition counts in theory, it is less in practice because environmental transitions contribute little to overall execution time. For an interactive tool, a 10% overhead is modest.

Main Java class	**Program**	**Java/C SLOC**	**Bug type**	**Bug site (source file:line)**
`UnitTest`	Java-gnome 4.0.10	64,171/67,082	Null parameter	Environment.c:48
`gconf.BasicGConfApp`	libgconf-2.16.2	796/1,157	Null parameter	org_gnu_gconf_ConfClient.c:425
`BadErrorChecking`	Blink-testsuite 1.14.3	15/9	Exception state	BadErrorChecking.c:21

Table 4.3: Studied JNI bugs. The two JNI bugs in `UnitTest` and `gconf.BasicGConfApp` are found when running these two programs with Blink. `BadErrorChecking` models exception handling bugs reportedly common in both user- and system-level JNI code [43, 75].

Main Java class	**Production run**		**Runtime checking (-Xcheck:jni)**		**Debugging session with J9 VM**		
					Single environment		**Mixed environment**
	Hotspot VM	**J9 VM**	**Hotspot VM**	**J9 VM**	**jdb**	**gdb**	**Blink**
`UnitTests`	running	crash	warning	warning	crash	fault	**breakpoint**
`gconf.BasicGConfApp`	running	crash	running	crash	crash	fault	**breakpoint**
`BadErrorChecking`	running	crash	warning	error	crash	fault	**breakpoint**

Table 4.4: Impact of JNI bugs under different configurations. *Running*: continue executing with undefined state. *Crash*: abort the JVM with a fatal error (e.g., segmentation fault). *Error*: exit JVM with error message. *Fault*: suspended by debugger due to an error inside the JVM, which becomes inoperable. *Breakpoint*: suspended by debugger, while JVM remains operable.

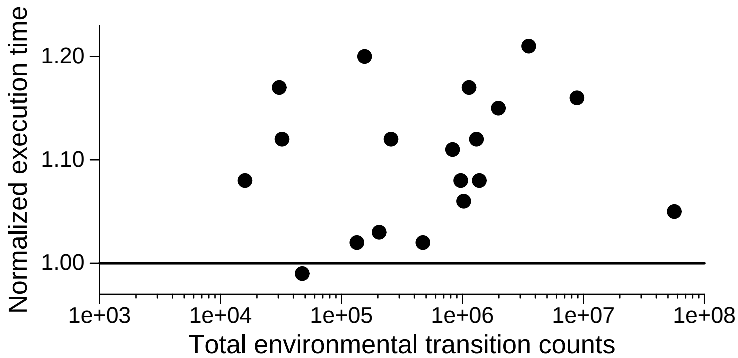


Figure 4.9: Environmental transitions and time overhead for the Blink debug agent with Hotspot VM 1.6.0_10. Note the logarithmic horizontal axis.

Space Overhead. The space overhead of running Blink is mostly due to additional code loaded into the debuggee. In particular, on Linux/IA32, the intermediate agent itself requires 388 KB, and the 229 JNI function wrappers introduce 174 KB of constant space overhead. Additionally, each native method incurs 11 bytes space overhead for its wrapper, instantiated from an assembly code template. Finally, each thread requires 156 bytes of thread-local storage used by the intermediate agent and less than 160 bytes for each wrapper activation on the stack for an environment transition. We do not measure total space overhead in a live system since it is small by design.

4.4.4 Feature Evaluation

This section explores how Blink saves programmers time and effort when diagnosing the source of mixed-environment bugs. We compare Blink to other tools using three case studies. In these studies, the other tools are not helpful, whereas Blink directly pinpoints the bugs.

We examine three common mixed-environment errors: one artificially recreated and two found in JNI programs in the wild. Table 4.3 lists the programs, lines of code, bug types, and bug sites. Blink directly identifies the two JNI bugs in `UnitTest` and `gconf.BasicGConfApp`. We also recreated an exception-handling bug in `BadErrorChecking`, which is reported as common in both user- and system-level JNI code [43, 75]. For each of these bugs, Table 4.4 compares Blink to production runs of Hotspot and J9, with runtime checking in Hotspot and J9 (configured with the `-Xcheck:jni` command line option), and with `jdb` and `gdb`.

In production runs with runtime checking, Hotspot and J9 behave differently, but neither JVM helps the user find bugs. Hotspot tends to silently ignore bugs without terminating, whereas J9 either crashes or reports errors. While seemingly improving stability, ignoring bugs in production runs may also corrupt state, which is clearly undesirable. The JVMs' runtime checking does not help much for two reasons. First, error messages are largely dependent on JVM internals and are inconsistent across the two JVMs. Second, and more importantly, the JVMs cannot interpret code and data in native code, where the JNI bugs originate.

Single-environment debuggers are also of limited use. The JNI bugs trigger segmentation faults, which are machine level events below the managed environment. As a result, the managed environment debugger (`jdb`) cannot catch the failure. The unmanaged environment debugger (`gdb`) catches this low-level failure, but detection is too late. For instance, the fault-inducing code never appears in the calling contexts of any thread when `gdb` detects the segmentation fault for J9 running `BadErrorChecking`.

Blink stops the programs immediately after it detects the JNI error conditions because it understands both environments. At the point of failure, programmers can inspect all the mixed-environment runtime state. We next discuss these errors in more detail, grouping them into two categories: (1) null parameters and (2) exception state checking.

Null Parameters. Semantics for JNI functions are undefined when their arguments are `(jobject)0xFFFFFFFF` or `NULL` [49]. Hotspot ignores these errors and J9 crashes in `gconf.BasicGConfApp` and `UnitTests`, which pass `NULL` to the `NewStringUTF` JNI function (see Table 4.4). `NewStringUTF` takes a C string and creates an equivalent Java string. Returning `NULL` for a `NULL` input may improve reliability, but it violates the specification of `NewStringUTF`:

> "Returns `NULL` if and only if an invocation of this function has thrown an exception." [49]

When Hotspot returns `NULL`, it should also post an exception. In addition, returning `NULL` may mislead JNI programmers into believing that `NewStringUTF` returns a `null` Java string when the input parameter is `NULL` [70]. J9 crashes and presents a low-level error message with register values and a stack trace. The error message does not include any clue to the cause of the bug. JVM runtime checking does improve the error message.

Blink detects the `NULL` parameter and presents the Java and C state on entry to the JNI function. Given the JNI failure in `gconf.BasicGConfApp`, a mixed-environment calling context tells the programmer that `NewStringUTF` does not return a `null` Java string for a `NULL` input with the following useful error message:

```
JNI warning:
NULL parameter to JNI Function: NewStringUTF
   425  return (*env)->NewStringUTF(env, val);
blink> where
[1] Java_org_..._1client_1get_1string
    (ConfClient.c:425)
[2] org.gnu.gconf.ConfClient.getString
    (ConfClient.java:440)
[3] gconf.BasicGConfApp.createConfigurableLabel
    (BasicGConfApp.java:128)
...
blink> _
```

Missing Exception State Checking. JNI does not define the JVM's behavior when C code calls a JNI function with an exception pending in the JVM. Consider this C source code from the `BadErrorChecking` micro-benchmark:

```
16. #include <jni.h>
17. JNIEXPORT void Java_BadErrorChecking_call (
18.   JNIEnv *env, jobject obj) {
19.   jclass cls = (*env)->GetObjectClass(
                      env, obj);
20.   jmethodID mid = (*env)->GetMethodID(
                      env, cls, foo, ()V);
21.   (*env)->CallVoidMethod(env, obj, mid);
22.   mid = (*env)->GetMethodID(
                      env, cls, bar, ()V);
23.   (*env)->CallVoidMethod(env, obj, mid);
24. }
```

At the call to Java in Line 21, the target Java method `foo` may raise an exception and then continue with the C code in Line 22, while the JVM has a pending exception. JNI rules require that the C code either returns immediately to the most recent Java caller or invokes the `ExceptionClear` JNI function. Consequently, the call to the JNI function `GetMethodID` in Line 22 leaves the JVM state undefined. In fact, Hotspot keeps running while

J9 crashes. This rule applies to 209 JNI functions out of 229 functions in JNI 6.0.

Writing the corresponding error checking code is tedious and error-prone. Previous work [43, 75] reports hundreds of bugs in JNI glue code. We briefly inspected the Java-gnome 4.0.10 code base and found two cases of missing error checking. One case never happens unless the JVM implements one JNI function incorrectly. The other case happens only when the JVM is running out of memory, throwing an `OutOfMemoryError` exception, which is rare and thus hard to find and test. For these reasons, we created the `BadErrorChecking` micro benchmark.

The intermediate agent in Blink detects calls to JNI functions while an exception is pending and asks Blink to stop the debuggee. Blink then warns the user of missing error checking and presents the calling context.

```
JNI warning: Missing Error Checking: GetMethodID
[1] Java_BadErrorChecking_call
      (BadErrorChecking.c:22)
[2] BadErrorChecking.main
      (BadErrorChecking.java:5)
...
blink> _
```

4.5 Generalization

The previous sections focus on composing debuggers for Java and C. Below, we discuss how to generalize our approach to additional environments. Section 4.6 describes our experience with extending Blink to include the Jeannie programming language, which mixes Java and C in the same methods.

4.5.1 More Languages, Same Environment

This section describes how to add a language if given a debugger for one of three environments: (1) native compiled, (2) interpreted, or (3) virtual machine execution.

Native multilingual debugging. Native environments use ahead-of-time compilers that generate assembly code for languages such as C/C++, Fortran, or Pascal. These languages interoperate by agreeing on a common object file format, an application binary interface (ABI), and a common debugging table format. To add a new native language to a native environment debugger, the compiler generates conforming debug tables along with conforming object code that obeys the ABI.

The *debug table* maps between language-level entities and environment-level entities. In particular, it maps program line numbers to addresses for execution control; addresses to program lines for context management; and variables, functions, etc. to reflective code snippets for data inspection. Debug table formats for native environments include dbx "stabs" [52], DWARF [24], and even PostScript [61]. This approach is however not bullet proof. For example, some of these formats do not support C++ identifiers and mangle the variable names. In these cases, the debugger implementation may compensate by adding functionality. This approach is not portable, but neither are the binaries.

Interpreted languages. Most scripting languages have an interpreter that directly executes source code, e.g., Perl, Python, or JavaScript. These interpreters typically support only one language and expose no debug table format for other languages. Therefore, multilingual debugging in these environments requires changing the interpreter itself [80].

Virtual machine environments. Virtual machines use ahead-of-time compilers to generate bytecode from source, and typically use a just-in-time (JIT) compiler to generate machine code. For example, compilers for Java, Scala, and Jython generate Java bytecodes, and modern Java virtual machines (JVMs) use a JIT to translate bytecode to machine code. Java bytecode contains debug tables as described in Sections 4.7.7–4.7.9 of the JVM specification [51]; JSR-45 introduces more expressive debug tables to better support multilingual debugging. Since there are multiple stages of compilation, each stage is responsible for keeping debug information intact. For instance, Java JITs either keep internal mappings for machine code, or use dynamic de-optimization [36]. Another example for a virtual machine supporting the debugging of multiple languages is Microsoft's Common Language Runtime (CLR).

4.5.2 More Environments, Same Languages

This section discusses the requirements for generalizing debugger composition to other mixed-language environments.

Requirement 1: Single-environment debuggers. As might be expected, debugger composition requires single-environment debuggers to compose. The single-language debuggers must support the features discussed in Section 4.1.1. The controller can extract these features through a command line interface (which is what we use), an API, or a wire protocol.

Requirement 2: Language transition interposition. Our approach requires instrumenting local and non-local control flow in all directions across environment boundaries. For Blink, we leverage Java's wrapper-based FFI to meet this requirement and instrument the wrappers. However, there are other viable implementation strategies for interposition. For example, given an interpreted language, the interpreter can call the instrumentation when encountering a transition. For a compiled language, the compiler can inject a call to the instrumentation when compiling a transition. Finally, when only compiled code is available, static or dynamic binary instrumentation can implement interposition.

Requirement 3: Debugger context switching. Our approach requires external interfaces to single-environment debugging functions, such as `print` or `eval`. Most single-environment debuggers provide these commands, including `jdb` and `gdb`. This ability is also a defining feature for languages with interactive interpreters, such as Perl, Python, Scheme, and ML. If, on the other hand, the single-environment debugger does not support direct function invocation, we must call the helper function through other means — for example, using an agent helper thread or a lower-level API underlying the single-environment debuggers.

Composing environments. Given two environments where one environment is the native C environment, it is easy to satisfy the above criteria. For instance, Perl, Python, and Ruby have debuggers and foreign function interfaces to C. We can thus satisfy the three requirements as follows: (1) reuse the `perldebug`, `pdb`, or `ruby-debug` single-environment debuggers and their interfaces; (2) extend the runtime systems to interpose calls to native methods; and (3) use `perldebug`, `pdb`, or `ruby-debug` to evaluate calls to native methods that trigger a C breakpoint.

For more than two environments ($N > 2$), there are $\frac{N \cdot (N-1)}{2}$ possible language transitions to interpose on and debugger context switches to perform. In theory, we could implement composition by adding agents for each pair of environments. In practice, the native C environment often acts as a bridge environment since most environments implement foreign function interfaces to C. Using C as a bridge environment, all the essential requirements are satisfiable: (1) N single-environment debuggers handle their corresponding N environments; (2) interposition captures transitions between the N environments and C because every transition goes through C; and (3) debugger context switching to any environment also goes through the bridging C environment.

4.6 Language Extension Case Study: Debugging Jeannie

This section shows how composition generalizes Blink to the Jeannie programming language [35]. Jeannie programs combine Java and C syntax in the same source file. This design eliminates many language-interface errors and simplifies resource management and multilingual programming. The Jeannie compiler produces C and Java code that executes in a native and JVM environment, respectively. Thus adding Jeannie to Blink serves as an example of debugging more languages in Blink's mixed environment.[2]

[2]Debugging Jeannie is distinct from borrowing Jeannie's expression evaluation functionality, which Blink also does and Section 4.3 described.

```
public static native void f(int x)
`.C{                                              A
  jint y = 0;
  `.Java{                                        B
    int z;
                                                C
                               D
    z = 1 + `(y = 1 + `(x = 1) ) ;
    System.out.println(x);
    System.out.println(z);
  }
  printf("%d\n", y);
}
```

Figure 4.10: Jeannie line number example.

Jeannie nests Java and C code in each other in the same file. Compared to JNI, Jeannie is more succinct and less brittle. For example, JNI obscures the Java type system, whereas Jeannie programs directly refer to Java fields and methods, which the Jeannie compiler type checks. In Jeannie, `` `. ``*language* specifies the language. As a shortcut, backtick `` ` `` toggles. For example, in Figure 4.10, the body of Java method `f` is the block A of C code. Block A contains a nested block B of Java code, with a nested C expression C, which, in turn, nests Java expression D. The Jeannie compiler emits separate Java and C files that implement the expected nesting semantics using JNI. In the example, the Jeannie compiler separates the code for the Java method declaration and snippets B and D into a Java file and puts the code for C snippets A and C into a C file. Jeannie's design supports adding more languages, but that is beyond the scope of this paper.

To add Jeannie to Blink, we changed the Jeannie compiler to generate and maintain debug tables for line numbers, method names, and variable locations, and we changed Blink to use these tables for Jeannie source-level debugging. The following sections illustrate how we extended Blink to support context management, execution control, and data inspection for Jeannie.

4.6.1 Context Management

Line numbers answer the question: "Where am I?" Call stacks answer the question: "How did I get here?"

Source line number information. To report the current location to the user, the debugger maps from low-level code offsets to source-level line numbers. The Jeannie compiler has access to source line numbers during translation, but relies on other compilers to generate low-level code. For debugging, we need to preserve line numbers through the second step. For Jeannie-generated Java code, we wrote a post-processor that rewrites Java bytecodes to reestablish the original line numbers from Jeannie sources. For Jeannie-generated C code, we rely on `#line` directives, which are supported by C compilers precisely to preserve debugging information for intermediate C code.

Calling context backtrace. Since Jeannie is a single language, Blink should show only the user-specified Jeannie methods and functions on the stack, instead of showing the generated single-language functions, which are just an implementation detail. For example, for the C source snippets A and C in Figure 4.10, the Jeannie compiler generates C functions `f_A` and `f_C`. For the Java snippets B and D, the Jeannie compiler generates Java methods `f_B` and `f_D`. When the application is suspended in D, the low-level call stack is:

$$\ldots \rightarrow \texttt{f} \rightarrow \texttt{f\_A} \rightarrow \texttt{f\_B} \rightarrow \texttt{f\_C} \rightarrow \texttt{f\_D}$$

but this trace is not reflected in the user's code. We changed the Jeannie compiler to generate a table mapping names of generated functions back to the original functions. Blink uses this mapping to hide low-level call frames and instead reports source-level names, e.g., just "$\ldots \rightarrow$ `f`".

4.6.2 Execution Control

Breakpoints answer the question "How do I get to a point in program execution?" Single-stepping answers the question "What happens next?"

Breakpoints. To support breakpoints in another language, the debugger needs to map from source-level lines to low-level code offsets. This requires similar debugging tables as for context management (Section 4.6.1), except in the opposite direction. In the case of Jeannie, there is one additional issue: Blink must delegate the breakpoint to the correct component debugger by using debugger context switching if necessary.

Single stepping. Stepping in Jeannie adds the challenge that a single source line may involve multiple languages. Line 6 in Figure 4.10 is an example. As discussed in Section 4.6.1, the Jeannie compiler tracks original line numbers even when code ends up in different source files. Blink implements Jeannie stepping by inspecting line numbers and iterating: it keeps stepping until the source line differs from the starting source line. For step-over, Blink records the current stack depth and then iterates, stepping until stack depth is less than or equal to the initial depth.

4.6.3 Data Inspection

Data inspection helps users determine if the current state is correct or infected. The compiler for each language must generate a table that maps source-level variable names to underlying variable access expressions in the generated code. The Jeannie compiler stores local variables in explicit environment records [35]. We extended the Jeannie compiler to provide the necessary mapping information through a separate symbol file, which Blink reads on demand.

4.7 Summary

Debugging is one of the most time-consuming tasks in software development. It requires a knack for formulating the right hypotheses about bugs and the discipline to systematically confirm or reject hypotheses until the cause of the bug is found [89]. Single-environment developers have long had good tools to help them navigate the debugging task systematically. However, mixed-language developers have been left in the dark. We propose and evaluate a new way to build cross-environment debuggers more easily using scalable composition. We use our compositional approach to develop Blink, a debugger for Java, C, and Jeannie. The open-source release of Blink is available as part of the xtc package [30]. Blink is the first full-featured debugger that is portable across different JVMs, operating systems, and C debuggers. Furthermore, Blink includes an interpreter (read-eval-print loop) for cross-environment expressions, thus providing users with a powerful tool not just for debugging but also for testing, program understanding, and prototyping.

Chapter 5

Code Interfaces: Generating Programs in any Language

Any program in one language can communicate with any program in another language if both languages have a string type and their programs are represented as strings. Specifically, a program generates another program as a string value, sends it to a compiler or an interpreter, and executes it. While this programming practice requires no extension to languages, compilers, and interpreters, there is no guarantee that the generated programs are syntactically and semantically correct, and the generation process does not abuse scoping constructs of the target languages. Although multistage programming systems [17, 55, 59, 85], meta-programming systems [15, 16, 65], and syntax macros [5, 84] check these errors, they are deeply coupled to particular programming languages and lose the scalability of code generation interfaces. This chapter presents the *Marco* macro system, an expressive, safe, extensible, and language scalable system that composes target language compilers.

We start by introducing the *Marco* language with its essential constructs and grammar in Section 5.1. Section 5.2 describes *Marco* analysis framework that embraces language-specific analysis plug-ins. Section 5.3 and Section 5.4 respectively analyze the open fragments in *Marco* programs and detect synthetic errors and unhygienic expansions. Section 5.5 presents our choices in implementing *Marco*. Section 5.6 evaluates *Marco*.

5.1 The *Marco* Language

This section describes the *Marco* language, using examples, grammar rules, and type rules. The *Marco* language is a statically typed, imperative language. It supports macros using three constructs: code types, fragments, and blanks. We define and illustrate these constructs using the motivating example in Figure 5.1, which uses the *Marco* syntax.

```
1 Code<cpp,stmt>                                # code type
2 synch(Code<cpp,id> mux, Code<cpp,stmt> body) {
3   return
4     `cpp(stmt)[{                              # C++ fragment
5        acquireLock($mux);
6        $body                                  # blank
7        releaseLock($mux);
8     }];
9 }
```

Figure 5.1: *Marco* code for `synch` example.

The macro in Figure 5.1 ensures that lock acquires and releases are properly paired, a paradigm made popular by Java's synchronized blocks, but which C does not provide. Programmers use macros such as this one to enforce good practices. Lines 1-2 contain the signature of the *Marco* function `synch`, which takes two parameters (a C++ identifier and a C++ statement) and returns a C++ statement. The **`Code`** type constructor is parameterized by the target language and the non-terminal in the target language. Line 4 uses the back-tick operator (**`` ` ``**) to begin a *fragment*, which is a quoted piece of target-language code. Line 6 uses the dollar operator (**`$`**) *blank*, which is an escaped piece of *Marco* code embedded in a fragment. The evaluation rule for a fragment first evaluates embedded blanks, then splices their results into the fragment's target-language code:

$$\frac{\forall i \in 1 \ldots n : \mathsf{Env} \vdash e_i \longrightarrow \beta_i \qquad \gamma = \alpha_0\beta_1\alpha_1 \ldots \beta_n\alpha_n}{\mathsf{Env} \vdash \texttt{`}\mathit{lang}\texttt{(}\mathit{nonT}\texttt{)}\ \texttt{[}\alpha_0\texttt{\$}e_1\alpha_1 \ldots \texttt{\$}e_n\alpha_n\texttt{]} \longrightarrow \texttt{`}\mathit{lang}\texttt{(}\mathit{nonT}\texttt{)}\ \texttt{[}\gamma\texttt{]}} \quad \text{(E-Fragment)}$$

In rule E-Fragment, each α_i is a sequence of target-language tokens in the fragment, each $\texttt{\$}e_i$ is a blank, and each β_i is the result of evaluating a blank to a sequence of target-language tokens. The result γ is the concatenation of all the α_i and β_i.

Figure 5.2 presents the *Marco* grammar. A program is a set of functions that take zero or more formal parameters and that return a value of some type. Each function body is a sequence of statements. A statement (*stmt*) is a local variable declaration, block , expression statement, conditional, loop, or function return. An expression (*expr*) is a fragment, an attribute access, a call expression, an infix expression (with operators such as addition (+) or assignment (=)), a subscript access, or a base expression. A base expression (*baseExpr*) is a parenthesized expression, a list literal, a record literal, an identifier, or a literal for a primitive value. In the absence of parentheses, *Marco* implements the usual precedence and associativity rules.

A fragment, such as `` `cpp(stmt)[...$x...$y...] ``, consists of a head and a sequence of fragment elements. The head specifies the target language, a non-terminal, and optionally a list of captured identifiers. We use the optional list of captured identifiers to check the naming discipline in the target code (see Section 5.4). There are two kinds of fragment elements: target-language tokens to be emitted, and blanks to be filled in during evaluation. Since the fragment elements are enclosed in square brackets, the *Marco* parser must count matching square brackets in the fragment itself to find the end,

program	::= $functionDef^{+}$
functionDef	::= *type* *ID* ‘(’ $formal^{*,}$ ‘)’ *blockStmt*
formal	::= *type* *ID*
stmt	::= *localDecl* | *blockStmt* | *exprStmt*
	| *ifStmt* | *forStmt* | *returnStmt*
localDecl	::= *type* *ID* ‘=’ *expr* ‘;’
blockStmt	::= ‘{’ $stmt^{*}$ ‘}’
exprStmt	::= *expr* ‘;’
ifStmt	::= ‘**if**’ ‘(’ *expr* ‘)’ *stmt* $($‘**else**’ *stmt*$)^{?}$
forStmt	::= ‘**for**’ ‘(’ *ID* ‘**in**’ *expr* ‘)’ *stmt*
returnStmt	::= ‘**return**’ *expr* ‘;’
expr	::= *fragment* | *attrExpr* | *callExpr*
	| *infixExpr* | *subscriptExpr* | *baseExpr*
attrExpr	::= *expr* ‘.’ *ID*
callExpr	::= *ID* ‘(’ $expr^{*,}$ ‘)’
infixExpr	::= *expr* *INFIX_OP* *expr*
subscriptExpr	::= *expr* ‘[’ *expr* ‘]’
baseExpr	::= *parenExpr* | *listLiteral* | *recordLiteral* | *ID*
	| ‘**true**’ | ‘**false**’ | *INT* | *STRING*
parenExpr	::= ‘(’ *expr* ‘)’
listLiteral	::= ‘[’ $expr^{*,}$ ‘]’
recordLiteral	::= ‘{’ $(ID$ ‘=’ $expr)^{+,}$ ‘}’
fragment	::= *fragmentHead* ‘[’ $fragmentElem^{*}$ ‘]’
fragmentHead	::= ‘\`’ *language* ‘(’ *nonTerminal* (‘,’ *capture*$)^{?}$ ‘)’
language	::= *ID*
nonTerminal	::= *ID*
capture	::= ‘**capture**’ ‘=’ ‘[’ $ID^{+,}$ ‘]’
fragmentElem	::= *TOKEN* | *blank*
blank	::= ‘**$**’ *baseExpr*
type	::= *codeType* | *listType*
	| ‘**boolean**’ | ‘**int**’ | ‘**string**’
codeType	::= ‘**Code**’ ‘<’ *language* ‘,’ *nonTerminal* ‘>’
listType	::= ‘**list**’ ‘<’ *type* ‘>’
recordType	::= ‘**record**’ ‘<’ $(type\ \ ID)^{+,}$ ‘>’

Figure 5.2: *Marco* grammar.

Marco grammar. The notation $formal^{*,}$ indicates that *formal* can repeat zero or more times, separated by commas.

for example, in `` `cpp(expr)[arr[idx]] ``. However, the *Marco* parser should not count square brackets in target-language strings or comments, for example, in `` `cpp(expr)[printf("[")] ``. Since different languages have different tokens for strings and comments, *Marco* needs target-language specific lexers. However, these lexers are simple, since they only need to recognize a few key target-language tokens.

The *Marco* type system includes code types parameterized by target language and non-terminal, list types parameterized by element type, record types parameterized by attribute names and types, and the primitive types **boolean**, **int**, and **string**. *Marco* is statically typed.

```
Code<sql,query>
genTitleQueryInSQL(Code<sql,expr> pred) {
  return `sql(query)[
      select title from moz_bookmarks where $pred
    ];
}
Code<cpp,stmt>
genSwapInCpp(Code<cpp,id> x, Code<cpp,id> y) {
  return `cpp(stmt)[{
      int temp = $x;
      $x = $y;
      $y = temp;
    }];
}
```

Figure 5.3: *Marco* code for generating code in different languages.

The three macro constructs and static typing are hardly new, but *Marco* is general with respect to the target language. First, code types and fragments are parameterized by target languages and their non-terminals. For instance, Figure 5.3 presents a *Marco* program that generates SQL and C++ code:

`genTitleQueryInSQL` generates a SQL query, and `genSwapInCpp` generates a C++ statement. Second, our *Marco* analysis framework analyzes code types and fragments using target language interpreters and compilers. For example, *Marco* reports error messages for the two fragments in Figure 5.3 if they result in expressions or statements that do not conform to the grammar of SQL expressions or C++ statements. *Marco* leverages the error messages from the relational database management system and the C++ compiler. Third, *Marco* uses free and captured identifiers from each fragment as inputs to a simple data-flow analysis. This data-flow analysis ensures that identifiers from multiple macros in the host program generate consistent identifier bindings in the target language statements and expressions. These last two steps are also novel and particular to *Marco*.

The type rule for a *Marco* fragment first checks the types for each of the embedded blanks, which must result in code belonging to the same target language (*lang*). It then uses the language *lang*, the non-terminal $nonT$ of the fragment, the non-terminals $nonT_i$ of each of the blanks, and the contents of the fragment as inputs to a syntax oracle. As far as the *Marco* type system is concerned, the syntax oracle is a black-box that can either succeed or fail. If the oracle succeeds, the type of the fragment is $\texttt{Code<}lang\texttt{,}\, nonT\texttt{>}$.

$$\frac{\forall i \in 1 \ldots n : \Gamma \vdash e_i : \texttt{Code<}lang\texttt{,}\, nonT_i\texttt{>} \qquad syntaxOracle(lang)(nonT, [nonT_1, \ldots, nonT_n], \alpha_0\texttt{\$}1\alpha_1 \ldots \texttt{\$}n\alpha_n)}{\Gamma \vdash \texttt{`}lang\texttt{(}nonT\texttt{)}\ \texttt{[}\alpha_0\texttt{\$}e_1\alpha_1 \ldots \texttt{\$}e_n\alpha_n\texttt{]} : \texttt{Code<}lang\texttt{,}\, nonT\texttt{>}} \quad \text{(T-Fragment)}$$

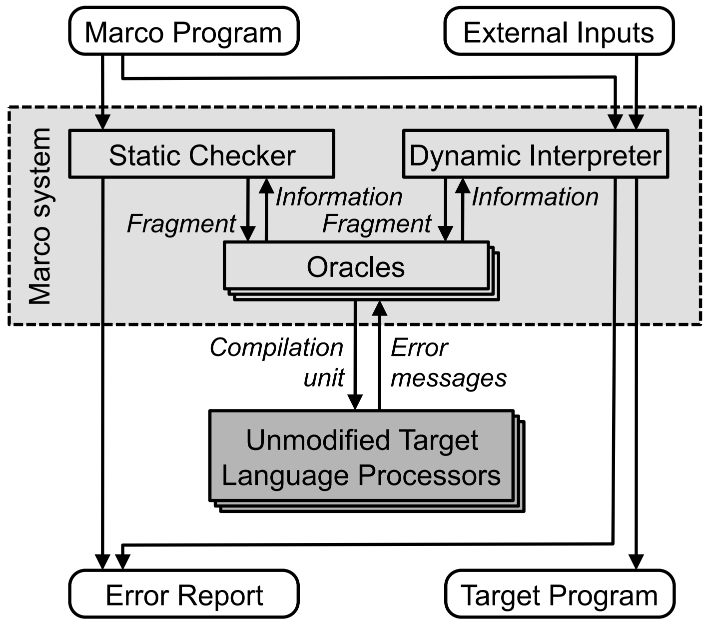


Figure 5.4: The Marco architecture.

5.2 The *Marco* Analysis Framework

Marco consists of a language and a system. The previous section describes the language, and this section overviews the system. The *Marco* system provides two tools: a static checker and a dynamic interpreter (see Figure 5.4). The *Marco* static checker checks the correctness of macros at macro development time. The *Marco* dynamic interpreter detects errors and generates target-language code at runtime if there are no errors. The *Marco* static checker and the dynamic interpreter share target-specific oracles, which check for syntactic well-formedness and naming discipline in target-language fragments.

The central design goal of *Marco* is target-language independence. *Marco* is extensible, since additional target-language specific *oracles* can be added as plug-ins without changing the framework. Each oracle communicates with a black-box target-language *processor* (compiler or interpreter) to analyze fragments. The oracle generates compilation units as inputs to a target-language processor, and parses error messages in outputs from the target-language processor. The only target-language specific parts of the *Marco* framework are the target-language specific lexers (see Section 5.1) and the oracles. In particular, a key advantage of *Marco* over other safe macro systems is that it does not require new or even modified target-language processors.

To understand the *Marco* framework, consider an example of statically checking syntactic well-formedness of target-language fragments. First, the *Marco* framework parses the *Marco* program into an Abstract Syntax Tree (AST). *Marco*'s type system expresses syntactic constraints on object-language fragments. For example, the type checker walks the AST and encounters the C++ fragment `` `cpp(expr)[x = 1;] ``. The static type checker applies rule T-FRAGMENT from Section 5.1, which triggers a call to

$$\mathit{syntaxOracle}(\texttt{cpp})(\texttt{expr}, [\,], `\texttt{x = 1;}')\text{.}$$

The C++ oracle generates the following input compilation unit for the unmodified C++ compiler (i.e., gcc):

```
int query_expr() { return x = 1;; }
```

For this input, gcc reports error messages. It complains about the spurious semicolon after `x = 1`. Based on this error message, the oracle deduces that the fragment was not a syntactically well-formed non-terminal `expr`. Since the oracle failed, *Marco* type-checking fails, and the *Marco* static checker reports an error. This example is simplified, ignoring idiosyncrasies of C++ and the issue of blanks, which Section 5.3 covers in detail.

Each language-specific plug-in consists of three oracles: *syntax*, *free* *names*, and *captured names*. Section 5.5 presents the implementation details of the Java factory method idiom that we use to create plug-in extensibility. The communication interface for the oracles depends on the target language. For instance, the interface for SQL is JDBC (Java Database Connectivity) and the interface for C++ is the file system and gcc (the GNU C and C++ compiler). Although these interfaces appear different at the concrete level, they share two key characteristics. First, they receive a program as a sequence of characters: input strings for JDBC and files for gcc. We simply lower the *Marco* fragments to produce character stream input for the language-specific oracles. Second, the output of these interfaces is a string that reports syntactic and semantic errors of the input program. The concrete error reporting mechanism depends on the target language. For instance, the error message from JDBC is encapsulated in a Java exception, and the error messages from gcc are printed to standard error.

As an example for checking naming discipline, consider a first fragment f_1 that fils in a blank in a second fragment f_2. Fragment f_1 is:

```
`sql(expr)[birthYear >= 1990],
```

and fragment f_2 is:

```
`sql(query)[select name from Patrons where $pred].
```

Marco uses its free-names oracle to discover that f_1 contains the free identifier `birthYear`; *Marco* uses a data-flow analysis to discover that f_1 flows into the blank of f_2; and *Marco* uses its captured-name oracle to check whether identifier `birthYear` gets captured at blank `$pred`. whether a capture is intentional; if it is not, *Marco* reports an accidental-capture error. The next sections describe how the oracles abstract error notifications from target-language processors into information for the static checker and the dynamic interpreter.

5.3 Checking Syntactic Well-Formedness

This section describes how the *Marco* system checks whether target-language code is syntactically well-formed. The *syntax oracle* is the interface between the target-language agnostic *Marco* system and the black-box target-language processors. The signature of the syntax oracle, as embodied in type rule T-FRAGMENT from Section 5.1, is:

$$\begin{aligned} \mathit{syntaxOracle} \;:\; & \mathit{lang} \\ & \rightarrow (\mathit{nonT}, \mathbf{\texttt{list}}{<}\mathit{nonT}{>}, \alpha_0\mathbf{\texttt{\$}}1\alpha_1 \ldots \mathbf{\texttt{\$}}n\alpha_n) \\ & \rightarrow \mathbf{\texttt{list}}{<}\mathit{error}{>} \end{aligned}$$

For example, consider the following invocation of the syntax oracle:

$$\begin{aligned} \mathit{syntaxOracle}\; & (\; \texttt{sql} \;) \\ & (\; \texttt{query}, [\texttt{expr}], \text{`}\mathbf{\texttt{select}}\ \texttt{a}\ \mathbf{\texttt{from}}\ \texttt{B}\ \mathbf{\texttt{where}}\ \mathbf{\texttt{\$}}1\text{'} \;) \end{aligned}$$

In this example, the target language is SQL, the non-terminal of the fragment is `query`, and there is only one blank, whose non-terminal is `expr`. The fragment contents have the form $\alpha_0\mathbf{\texttt{\$}}1\alpha_1$, where α_0 is the sequence of tokens before the blank, **`$`**1 marks the location of the blank, and α_1 is the sequence of tokens after the blank. In other words, α_0 is '**`select`** `a` **`from`** `B` **`where`**' and α_1 is empty. The remainder of this section describes the syntax oracle algorithm for producing compilation units, interpreting the results, and iterating when necessary.

5.3.1 Syntax Oracle Algorithm

The algorithm of the syntax oracle has four steps. Recall that a *blank* is a gap in a target-language fragment where another fragment will be spliced in at runtime.

Marco type	Place-holder fragment	Completion fragment
Code<*sql*, *qlist*>	`*sql*(*qlist*)[]	`*sql*(*qlist*)[**$***orig*]
Code<*sql*, *query*>	`*sql*(*query*)[**select** *]	`*sql*(*qlist*)[**$***orig*]
Code<*sql*, *expr*>	`*sql*(*expr*)[0]	`*sql*(*qlist*)[**select $***fresh1* **from $***fresh2* **where $***orig*]
Code<*cpp*, *id*>	`*cpp*(*id*)[**$***fresh*]	`*cpp*(*cunit*)[**int $***fresh*() **return $***orig*; }]
Code<*cpp*, *type_spec*>	`*cpp*(*type_spec*)[**int**]	`*cpp*(*cunit*)[**$***orig* **$***fresh*;]
Code<*cpp*, *type_id*>	`*cpp*(*type_id*)[**int**]	`*cpp*(*cunit*)[**void $***fresh*() {*sizeof*(**$***orig*);}]
Code<*cpp*, *expr*>	`*cpp*(*expr*)[0]	`*cpp*(*cunit*)[**int $***fresh*() { **return $***orig*; }]
Code<*cpp*, *stmt*>	`*cpp*(*stmt*)[;]	`*cpp*(*cunit*)[**int $***fresh*() { **switch**(0) **$***orig* **return** 0;}]
Code<*cpp*, *fdef*>	`*cpp*(*fdef*)[**void $***fresh*(){}]	`*cpp*(*cunit*)[**$***orig*]
Code<*cpp*, *mdecl*>	`*cpp*(*mdecl*)[**int $***fresh*;]	`*cpp*(*cunit*)[**class $***fresh* { **$***orig* };]
Code<*cpp*, *decl*>	`*cpp*(*decl*)[**int $***fresh*;]	`*cpp*(*cunit*)[**$***orig*]
Code<*cpp*, *cunit*>	`*cpp*(*cunit*)[]	`*cpp*(*cunit*)[**$***orig*]

Table 5.1: Helper fragments used in the syntax oracles.

Helper fragments used in the syntax oracles. Place-holder fragments are used to fill in blanks in a fragment. Completion fragments are used to turn a fragment into a self-contained compilation unit for the target-language processor. Blanks of the form **$***fresh* are filled in with fresh identifiers, and blanks of the form **$***orig* are filled in with the original fragment.

Step 1: Fill in blanks. The syntax oracle starts by filling in each blank with a place-holder fragment. In other words, it turns the fragment with blanks into a fragment without blanks. A place-holder fragment is a fragment that is syntactically valid for a given non-terminal. In the example above, the non-terminal for blank 1 is `expr`, so the syntax oracle fills in blank 1 with the place-holder fragment for SQL expressions, which is `0`. The result is the fragment `select a from B where 0`. The middle column of Table 5.1 shows the place-holder fragments for each of the code types in *Marco*'s SQL and C++ plug-ins. The intuition why filling in blanks works is that target-languages have (more or less) context-free grammars, and that the syntax oracle can check syntactic validity even when there are semantic errors. For instance, in the example, the place-holder fragment is of type integer and the blank expects type boolean, but this semantic mismatch is irrelevant to syntactic well-formedness.

Step 2: Complete the fragment. Next, the syntax oracle completes the fragment to obtain a self-contained compilation unit for the target-language processor. In this example, the fragment is already a full query, and needs no additional completion. The right column of Table 5.1 shows the completion fragments for each of the code types in *Marco*'s SQL and C++ plug-ins. In these completion fragments, `$orig` refers to the original fragment. Besides completing the fragment to a full compilation unit, Step 2 may also generate additional boiler-plate syntax. For SQL, this step adds code to begin and then abort a transaction, in order to prevent side-effects when sending the SQL query to a live database during analysis.

Step 3: Run the target-language processor. At this point, the syntax oracle sends the completed fragment to the target-language processor, and collects error messages, if any. As discussed above, in the case of SQL, *Marco* makes a JDBC call and catches any exceptions. For C++, *Marco* generates a file with the fragment, compiles it with `gcc`, and reads any error messages from `stderr`.

Step 4: Determine the oracle results. Finally, the syntax oracle translates errors from the target-language processor into oracle results. It must distinguish syntax errors from any other errors. It only fails the syntactic well-formedness test if there are syntax errors. In C++, syntax errors may be masked by other, non-syntax, errors, so the oracle may iterate to determine if the fragment also has a syntax error, as Section 5.3.3 explains. If the syntax oracle fails, the oracle maps the line-numbers in the error message to the *Marco* code, and reports the errors to the *Marco* user.

5.3.2 Syntax Oracle Example

To understand the syntax oracle in action, consider the example fragment `` `sql(expr)[type = ] ``, which has an obvious syntax error: the right operand is missing. Type rule T-FRAGMENT invokes the syntax oracle as follows: *syntaxOracle*(`sql`)(`expr`, [ ], '`type =`'). The oracle goes through its four steps:

1. Fill in blanks. This step is a no-op, since there are no blanks.

2. Complete the fragment. The oracle consults Table 5.1 to generate the completion fragment for **`Code`**<`sql,expr`>, resulting in the fragment

select *x* **from** *T* **where** *type* =.

3. Run the target-language processor. The oracle uses JDBC to send the completed fragment to SQLite, and then catches the resulting SQLException, which contains the error message "Syntax error near '='.".

4. Determine the oracle results. Since the error from the target-language processor was a syntax error, the oracle reports this error back to the user.

Now, assume that the programmer fixes the fragment by writing `` `sql(expr)[type = 1] ``, and then runs *Marco* again.

1. Fill in blanks. This step is a no-op, since there are no blanks.

2. Complete the fragment. This step yields the completed SQL query **select** *x* **from** *T* **where** *type* = 1.

3. Run the target-language processor. If the backing database does not have a table `T` with an attribute `type`, the error message is:

   "No attribute '`type`' in table '`T`'.".

4. Determine the oracle results. Since the error from the target-language processor is not a syntax error, the oracle succeeds and indicates that the fragment is syntactically well-formed.

5.3.3 Handling Masked Syntax Errors in C++

The C programming language has a context-sensitive grammar. For example, the C code `A(*x)[4] = y;` can be parsed either as a function call or as a variable declaration. On the one hand, if `A` is a function, then the code calls the function with parameter `*x`, accesses element `[4]` of the result, and assigns `y` to it. On the other hand, if `A` is a type, then the example declares the variable `x` to be a pointer to an array of 4 elements of type `A`, and initializes `x` to `y`. Since the C++ programming language is a super-set of C, it includes this context-sensitive case. It also contains other, even more difficult cases.

As the example shows, parsing for C and C++ depends on how identifiers are declared, and may thus cause semantic errors to mask syntax errors. In other words, when gcc reports a semantic error but no syntax error, it is possible that there is a masked syntax error, which only shows up after the semantic error is resolved. Our C++ oracle automatically speculates resolutions for semantic errors by declaring additional boiler-plate code when it completes a fragment. We thus iterate. If Step 4 from Section 5.3.1 detects errors, the algorithm iterates back to Step 2, which speculatively resolves them by generating declarations for free identifiers.

The syntax oracle for C++ uses the following classification of error messages to drive its speculations and resulting actions:

A. Missing declaration. When the oracle encounters an error message of the form "... was not declared in this scope", it speculates that the identifier is a type name, a variable name, a namespace name, or function name. If a downstream iteration finds an entity-kind error or a syntax error, the oracle may backtrack the speculative declaration. If it backtracks all the speculations, the oracle flags an error in the input fragment. To backtrack speculations, it saves the iteration state before speculations.

B. Entity-kind error. When the oracle encounters an error message of the form "... is not a namespace-name" or "... cannot be used as a function", it either infers the entity of an identifier, or aborts the most recent speculation. The oracle extracts a problematic identifier from the error message. If the identifier is new to the generated declarations, the oracle declares the identifier as a namespace or a function depending on a particular error message. Otherwise, it backtracks the most recent speculative iteration.

C. Semantic error. There are many error messages in this category, for example, "too many arguments to function ..." or "invalid conversion ...". The oracle just ignores the messages about semantic errors that do not mask syntax errors.

D. Syntax error. An example of a syntax error is "expected '`;`' before ...". When the oracle encounters such an error, it either fails the most recent speculation, or it forwards the error message the user.

5.4 Checking Naming Discipline

This section describes how the *Marco* system checks that code generation does not cause accidental name capture in the target language. Accidental name capture is a typical bug when using the C preprocessor, as illustrated by Figure 5.5.

```
#define swap(v,w) {int temp=v; v=w; w=temp;}
int temp = thermometer();
if (temp<lo_temp) swap(temp, lo_temp)
```

Figure 5.5: Example of accidental name capture bug when using the C preprocessor [17, 22].

Line 1 in Figure 5.5 declares a macro `swap` that contains a local declaration of a variable `temp` (short for "temporary"). Line 2 declares a different variable `temp` (short for "temperature") that is not nested in the macro. Line 3 passes identifier `temp` as an actual parameter to the formal `v` of `swap`. The problem is that at the use of `v`, the identifier `temp` gets captured. Since the author of the code intended to use `temp` to refer to "temperature," this problem is called an accidental name capture.

More generally, accidental name capture happens when a first fragment f_1 contains a free identifier x; a second fragment f_2 unintentionally captures identifier x at blank b; and f_1 flows into b. *Marco* detects this situation as follows. The *freeNamesOracle* discovers all free identifiers in fragment f_1. The *capturedNameOracle* checks whether blank b in fragment f_2 unintentionally captures a given identifier x. *Marco* uses a forward data-flow analysis to propagate free identifiers to capturing blanks. *Marco* uses static data-flow analysis for the static checker at macro development time, and dynamic data-flow analysis for the interpreter at code generation time. The oracles are target-language specific and use the target-language processor as a black-box to generate error messages that reveal information about free and captured names. The data-flow analyses are target-language independent.

5.4.1 Free-Names Oracle

The signature of the free-names oracle is:

$$\begin{aligned} \mathit{freeNamesOracle} \ :\ & \mathit{lang} \\ & \rightarrow (\mathit{nonT}, \mathbf{\texttt{list}}\texttt{<}\mathit{nonT}\texttt{>}, \alpha_0\mathbf{\texttt{\$}}1\alpha_1 \ldots \mathbf{\texttt{\$}}n\alpha_n) \\ & \rightarrow \mathbf{\texttt{list}}\texttt{<}\mathit{ID}\texttt{>} \end{aligned}$$

For example, consider the following fragment, which contains a free name: `` `cpp(expr)[100 * (1.0 / (foo))] ``. *Marco* invokes the free-names oracle for it as follows:

$$\mathit{freeNamesOracle}(\texttt{cpp})(\texttt{expr}, [\,], \text{`}\texttt{100 * (1.0 / (foo))}\text{'})$$

A name in a fragment is free if it is not bound inside the fragment. In the example, `foo` is free, and thus, the oracle call returns the list [`foo`]. The first three steps of the free-name oracle algorithm mimic the form of the syntax oracle from Section 5.3.1. In particular, *freeNamesOracle* executes the following five steps for the example:

Step 1: Fill in blanks. This step is the same as Step 1 of the syntax oracle. Since `` `cpp(expr)[100 * (1.0 / (foo))] `` has no blanks to fill in, this case is a no-op.

Step 2: Complete the fragment. This step is the same as Step 2 of the syntax oracle. The completed example fragment is:

```
int query_expr() { return 100 * (1.0 / (foo)); }
```

Step 3: Run the target-language processor. For this query, gcc returns an error message of the form "identifier '`foo`' was not declared in this scope".

Step 4: Resolve declaration errors. The free-names oracle looks for particular error messages complaining that a name is used without definition. In the example, the message specifies the name `foo`. The oracle speculates that *foo* is free. To validate this hypothesis, it runs one more experiment. It prepends a declaration of the name *foo* to the translation unit, and sends it again to the target language processor. In the example, the test is:

```
int foo;
int query_expr() { return 100 * (1.0 / (foo)); }
```

In this case, the modification resolves the declaration error, confirming that the hypothesis is correct. Hence, the oracle adds the name *foo* to the list of free names. It repeats this process until it does not observe any more declaration errors. The insight *Marco* exploits is that a name in a fragment is free, as long as it could be bound by a declaration from an enclosing scope.

Step 5: Return free identifiers. The free-names oracle returns the list of free identifiers it found, which the *Marco* data-flow analysis will propagate and *Marco* will use these names as inputs to the captured-name oracle.

5.4.2 Captured-Name Oracle

The captured-name oracle checks, for a given fragment, blank number, and identifier, whether that identifier is captured at the blank. In other words, the captured-name oracle checks whether it is safe to fill in the blank with a fragment in which the identifier is free. The signature of the captured-name oracle is:

$$\begin{aligned} \mathit{capturedNameOracle} \;:\;& \mathit{lang} \\ & \rightarrow (\,\mathit{nonT}, \texttt{\textbf{list}}{<}\mathit{nonT}{>}, \\ & \quad\;\; \alpha_0 \texttt{\textbf{\$}} 1 \alpha_1 \ldots \texttt{\textbf{\$}} n \alpha_n, \texttt{\textbf{int}}, \mathit{ID}) \\ & \rightarrow \texttt{\textbf{boolean}} \end{aligned}$$

Here, **`int`** is the blank number, and *ID* is the free identifier whose capture is to be checked. Consider the fragment for swapping two integers: `` `cpp(stmt)[{int temp=$v; $v=$w; $w=temp;}] ``. The following oracle call checks whether blank 1 captures identifier `temp`:

```
capturedNameOracle ( cpp )
                   ( stmt, [expr, expr, expr, expr],
                     '{int temp=$1; $2=$3; $4=temp;}',
                     1, temp )
```

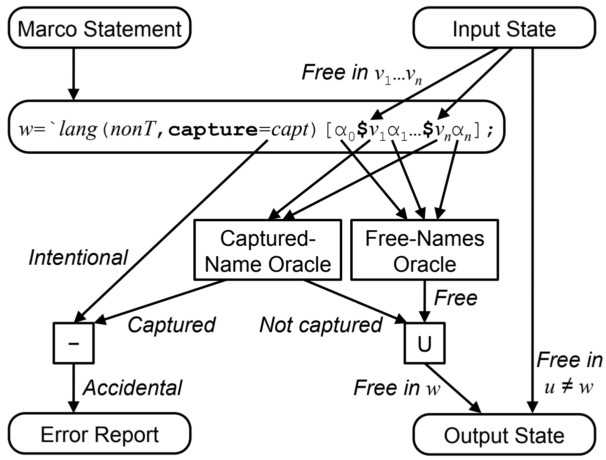
Since blank 1 in the fragment does in fact capture the name `temp`, the oracle returns **`true`**. As another example, consider fragment:

```
`sql(query)[select name from Patrons where $pred].
```

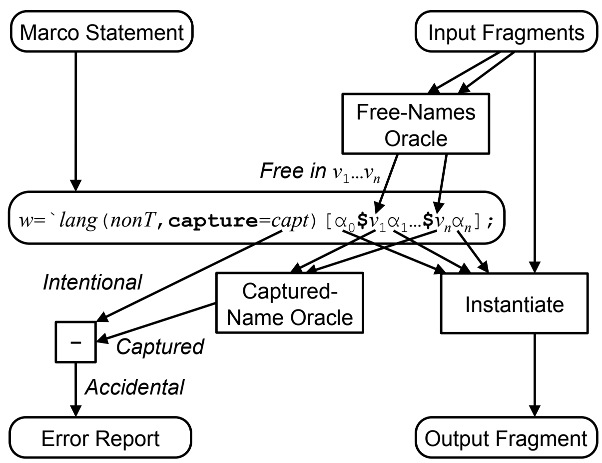
Blank 1 in this fragment captures any identifier that refers to column names in the `Patrons` table in the database. Note that SQL's scoping rules implement semantics similar to a **`with`**-statement, presenting a different challenge for naming discipline than the C++ scoping rules. The algorithm for the *capturedNameOracle* handles both target languages with the same steps:

Step 1: Fill in blanks. This step differs somewhat from Step 1 in the other oracles. Assume that *capturedNameOracle* was invoked to check whether free name x is captured at blank number i. Like the other oracles, the captured-name oracle fills in all blanks j with $i \neq j$ using the place-holders corresponding to their non-terminals from Table 5.1. However, for blank i, our analysis hypothesizes that x is captured at the blank. To find counter-evidence, it places x in the blank, wrapping it as necessary in some boiler-plate code for syntactic well-formedness.

Step 2: Complete the fragment. This step is the same as in the other oracles.



(a) Transfer function for static data-flow analysis.



(b) Dynamic analysis piggy-backed on interpreter.

Figure 5.6: Transfer functions for the naming-discipline analysis.

Step 3: Run the target-language processor. This step is the same as in the other oracles.

Step 4: Determine oracle result. If the target-language processor reports an error message indicating that x is unknown, then the oracle concludes that x is not captured at blank i, and returns **`false`**. Otherwise, the oracle returns **`true`**.

5.4.3 Static Data-Flow Analysis

The static data-flow analysis runs as part of the static checker. The static checker first reads in the entire *Marco* program, and checks all fragments for syntactic well-formedness. If there are no syntax errors, the static checker runs the static data-flow analysis, which in turn invokes the free-names and captured-name oracles as needed.

The static data-flow analysis propagates free target-language identifiers through variables and blanks. It reports an error whenever a free identifier gets accidentally captured at a blank. To determine whether a capture is accidental or intentional, the analysis uses the optional **`capture`** annotation in the *fragmentHead* clause of the *Marco* grammar (see Figure 5.2). If an identifier is listed in the **`capture`** annotation, the analysis knows that the capture is intentional, otherwise it assumes that the capture is accidental and reports an error.

For an example of intentional capture, consider the *Marco* function `boundIf` in Figure 5.7. This function implements an if-statement that binds the value of the condition to a variable `it`, so that it can be used in the body. The annotation **`capture`**`=[it]` in Line 4 indicates that the fragment intentionally captures identifier `it`. Now, assume that blank 2 gets filled with fragment `printf("%d", it);`. Since the analysis should only report accidental capture and not intentional capture, it will not report an error for `it`

```
Code<cpp,stmt>
boundIf(Code<cpp,expr> cond, Code<cpp,stmt> body) {
  # the following fragment intentionally captures ‘it’
  return `cpp(stmt, capture=[it])[{
      int it = $cond;          #blank 1
      if (it) { $body }        #blank 2
    }];
}
```

Figure 5.7: Example for intentional name capture when using *Marco* to generate C++ code.

even though it appears free in the fragment and gets captured in blank 2.

A *Marco location* is a formal parameter of a function or a local variable in the meta-language, whereas an identifier *ID* is a token in the target-language. The analysis state at a program point is a map from *Marco* locations to lists of free target-language identifiers. In other words, the state is an element of the following lattice:

$$\mathit{analysisState} = \mathit{location} \rightarrow 2^{\mathit{ID}}$$

The *join* function, or least-upper bound, for the lattice unions the lists of free identifiers for each location. In other words, for each *Marco* location v:

$$\mathit{join}(\mathit{state}_1, \mathit{state}_2)(v) = \mathit{state}_1(v) \cup \mathit{state}_2(v)$$

As usual with data-flow analyses, the name-discipline analysis uses join functions to combine analysis state at control-flow merges. The lattice has finite height, because the state of each location is a subset of the finite set of all identifiers that are free in fragments of the program.

The naming-discipline analysis is a forward data-flow analysis, and the transfer function has the usual signature:

$$\mathit{transferFunction} : \mathit{statement} \to \mathit{analysisState} \to \mathit{analysisState}$$

The interesting statements for the data-flow analysis are statements with fragments and blanks. Figure 5.6(a) shows the transfer-function for such statements.

Given a *Marco* statement and an input analysis state *inState*, the transfer function computes an output analysis state *outState*. The analysis state only changes for the *Marco* location w assigned by the statement. The captured-name oracle from Section 5.4.2 checks whether free names from $\mathit{inState}(v_1)$ thru $\mathit{inState}(v_n)$ are captured by the fragment. If they are captured, and the capture is not intentional, the analysis reports an error. On the other hand, if they are not captured, then they are still free in w. In addition, the free-names oracle checks for free names in the constant portionsfor α_0 thru α_n of the fragment. Those free names are also free in w. The resulting output state $\mathit{outState}(w)$ uses the free names for w as discovered by the oracles. For all other locations $u \neq w$, the transfer function forwards the free names from the input state $\mathit{outState}(u) = \mathit{inState}(u)$.

We implement our data-flow analysis with a work-list algorithm. Initially, all analysis states are empty, and the work-list contains all statements that manipulate fragments. While the work-list is non-empty, the analysis removes a statement from the work-list, applies its transfer function, and if the output analysis state changes, adds all successor statements to the work-list. The analysis terminates, because analysis states grow monotonically, and because the lattice has finite height.

One pragmatic issue is how to report high-quality error messages in the case of accidental name captures. The analysis remembers which errors it has reported so far and avoids duplicates. Furthermore, the analysis tracks the originating fragment for each free identifier to more accurately report the source of accidental name captures. When the analysis detects an accidental capture, it reports both the line number of the origin and the line number of the capture in the *Marco* program.

5.4.4 Dynamic Data-Flow Analysis

The static data-flow analysis checks the naming discipline. It reports accidental name capture errors to macro authors at development time. However, a *Marco* program may also receive fragments from external input parameters. These fragments may contain free identifiers, and thus, the *Marco* dynamic interpreter checks for accidental captures at code-generation time as well. As before, a capture is accidental if the programmer has not designated it as intentional using an annotation in the fragment head.

Figure 5.6(b) shows how the dynamic interpreter piggy-backs the data-flow analysis on the execution of the *Marco* program. When the interpreter instantiates a fragment, the data-flow analysis collects the free names from each blank, using the *freeNamesOracle*. For each free name, it uses the *capturedNameOracle* to check whether the free name is captured at the blank. If the name is captured and the capture is not intentional, the analysis reports an error and aborts the program. Otherwise, the interpreter fills in the blanks with the input fragments to produce an output fragment.

5.5 Implementation

This section presents our implementation choices in adapting *Marco* to realistic programming environments. Section 5.5.1 and Section 5.5.2 respectively present a few primitive functions to express low-level operations and a foreign function interface that use legacy libraries. Section 5.5.3 shows how we adapt the factory method pattern in object-orient programming languages to build the *Marco* framework that embraces language-specific oracles.

5.5.1 Primitive Functions

Primitive functions express the low-level operations that a user-defined *Marco* function cannot perform. Such low-level operations include generating fresh identifiers (`gensym`) and concatenating strings to generate an identifier (`catid`). Our static checker does not analyze the effect of executing them. For instance, our static dataflow analyzer assumes that `catid` does not generate any free name. Our dynamic interpreter discovers the free names and ensures that they are not accidentally captured.

5.5.2 Foreign Function Interface

Foreign function interfaces enable one language to use legacy libraries in another language. The *Marco* foreign function interface to Java requires very little extension to the *Marco* system while allowing it to reuse legacy libraries in Java. For instance, some of our applications read code fragments from XML files. Writing an XML parser in *Marco* from scratch would be an unnecessary effort given that there are high-quality XML parsers that have stood the test of time. To check the safety of foreign functions, *Marco* uses the dynamic interpreter instead of the static checker. For locating the origin of errors, a programmer must specify the origin of code fragments generated through the foreign function interface. For instance, the programmer may point out an XML file and one of its source lines. Specifically, a *Marco* native method would manually create the fragment tokens with their file names and line numbers that copied from those in the XML tree generated by an XML parser.

5.5.3 Factory Method Pattern

We use a factory method pattern to abstract how language-specific analyses are created. Each factory is responsible for a target language, creating instances of the oracles for syntax, free names, and captured names of a fragment. Figure 5.8 presents the class hierarchy diagram for the factory classes for SQL and C++. OracleFactory is an abstract class that declares three methods for creating language-specific oracles. The createSyntaxOracle method creates an object of the ISyntaxOracle interface that checks the syntactic well-formedness of a fragment. The createFreeNamesOracle method creates an object of the IFreeNamesOracle interface that extracts the free names from a fragment. The createCapturedOracle method creates an object of the ICapturedNameOracle interface that checks whether or not a given name is captured at a blank in a fragment.

To map the language of a fragment to its factory class, we use a hash-table and reflection in Java. The hash-table maps from language identifiers (e.g,. `sql` or `cpp`) to factory class names (e.g., SQLOracleFactory or CPPOracleFactory). To add a new target language and its oracle factory class, a *Marco* plug-in writer edits the *Marco* property file used for populating the hash table. When a language lookup is successful, the analysis framework dynamically

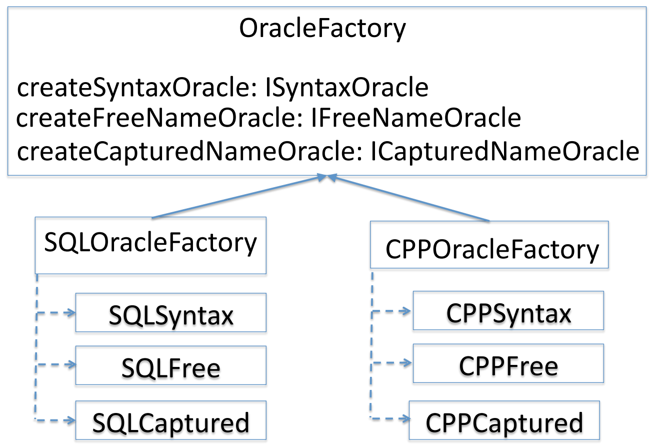


Figure 5.8: Java class hierarchy for oracle factories.

loads a factory class using the factory name. Then, it instantiates a factory object from the class using Java reflection and executes a method for creating the oracle object. For instance, the method SQLOracleFactory.createSyntax creates a syntax oracle object for an SQL fragment. This oracle checks syntactic well-formedness of the SQL fragment by repeating the process of synthesizing an SQL query, sending it to a database management system, and receiving the error messages.

5.6 Results

This section experimentally validates three characteristics of *Marco*: expressiveness, safety, and scalability. Section 5.6.1 describes the methodology and tools for our experiments. To evaluate expressiveness, we implemented several macros in *Marco*, ranging from micro-benchmarks from prior work to a code-generation template for a high-performance stream processing operator. To evaluate safety, we ran *Marco* on each of the micro-benchmarks and on the streaming operator. The expressiveness and safety results are in Section 5.6.2. To evaluate scalability, Section 5.6.3 reports statistics on the implementation effort for supporting different target languages.

5.6.1 Methodology

Experimental environments. We used *Marco* r237 running on Sun HotSpot Client 1.6.0_21-ea. For the unmodified target language processors for C++ and SQL, we downloaded and built gcc 4.6.0 r164675 (20100928) and SQLiteJDBC v056 based on SQLite 3.4.14.2. We conducted all the experiment on a Pentium D T3200 with 2 GB main memory. The machine runs on Ubuntu 11.04 on the Linux 2.6.35-28 kernel.

***Marco* programs.** We used several *Marco* programs: micro-benchmarks derived from related work [17, 84] and the `Aggregate` operator derived from IBM InfoSphere Streams [18]. The micro-benchmarks are a collection of 8 small *Marco* programs that generate C++ programs without classes, namespaces, and templates. The `Aggregate` operator generates C++ declarations, statements, and expressions that exercise classes, namespaces, and templates.

Data collection methodology. To collect statistical results from analyzing fragments, we turned on *Marco*'s -pstat command-line option. To count source lines of code, we ran the sloccount utility.

5.6.2 Expressiveness and Safety

This section demonstrates expressiveness of *Marco* by presenting several *Marco* micro-benchmarks and the `Aggregate` operator written in *Marco*. This section demonstrates safety of *Marco* by running the *Marco* system on all the tests.

5.6.2.1 Micro-Benchmarks

Table 5.2 presents our micro-benchmarks. We rewrote the first four programs in the MS^2 paper by Weise and Crew [84] in the *Marco* language. These macros complement the C language with abstractions such as resource management (`paint`), dynamic binding (`dynamic_bind`), exception handling (`exception_handling`), and multiple declarations (`myenum`). The remaining three programs re-implement examples in the "macros that work" paper by Clinger and Rees [17] in *Marco*. These macros illustrate naming issues in macro expansions. All seven micro-benchmarks produce expressions, statements, and declarations in C++.

Each program contains a few fragments (Column "Fragment"). We name them using the name of the functions they appear in and the order in which they appear. Column "Code type" shows the types of the macros, which indicate the target language and the non-terminal. Column "Size" counts the number of target-language tokens and blanks. The remaining columns present statistical results from running the oracle analysis. The oracle analysis synthesizes several query programs before it concludes that the input fragment is syntactically correct. Column "Backtracks" counts how often the syntax oracle needed to backtrack before it finished. Column "Queries" counts the number of compilation units sent to the target-language processor. Column

Marco Program	Fragment	Code type	Size	Backtracks	Queries	Declarations
paint	*Painting1*	**Code**<*cpp*, *stmt*>	19	2	8	6
dynamic_bind	*dynamic_bind1*	**Code**<*cpp*, *stmt*>	13	3	9	7
exception_handling	*throw1*	**Code**<*cpp*, *stmt*>	25	2	8	6
	throw2	**Code**<*cpp*, *stmt*>	28	2	8	6
	catch1	**Code**<*cpp*, *expr*>	1	1	4	2
	catch2	**Code**<*cpp*, *stmt*>	52	1	5	3
	unwind_protect1	**Code**<*cpp*, *expr*>	1	1	4	2
	unwind_protect2	**Code**<*cpp*, *stmt*>	45	2	7	5
myenum	*myenum1*	**Code**<*cpp*, *decl*>	6	0	1	0
	myenum2	**Code**<*cpp*, *stmt*>	10	1	5	3
	myenum3	**Code**<*cpp*, *decl*>	15	0	1	0
	myenum4	**Code**<*cpp*, *stmt*>	17	2	7	5
	myenum5	**Code**<*cpp*, *decl*>	20	0	3	1
discriminant	*discriminant1*	**Code**<*cpp*, *expr*>	9	0	1	0
complain	*complain1*	**Code**<*cpp*, *stmt*>	5	0	3	1
	main1	**Code**<*cpp*, *expr*>	1	1	4	2
	main2	**Code**<*cpp*, *stmt*>	15	1	4	2
swap	*swap1*	**Code**<*cpp*, *id*>	1	1	4	2
	swap2	**Code**<*cpp*, *id*>	1	1	4	2
	swap3	**Code**<*cpp*, *stmt*>	31	5	13	11
SQLSyntax	*good1*	**Code**<*sql*, *expr*>	3	0	1	0
	good2	**Code**<*sql*, *stmt*>	6	0	1	0

Table 5.2: Oracle analysis results for the fragments in the micro-benchmarks.

"Declarations" shows the number of declarations that the oracles needed to synthesize provide evidence for syntactic well-formedness.

For fragments containing 1-52 tokens or blanks, our oracle analyzer concludes syntactic well-formedness after evaluating 1-9 query fragments. The number of queries is proportional to the number of synthesized declarations rather than the size of input fragments. This result is not surprising, because the number of C++ parsing errors for syntactically well-formed fragments would be proportional to the number of undefined symbols. About 10-20% of query programs backtrack speculations during the oracle analysis.

5.6.2.2 Aggregate Operator

In a data-stream management system (DSMS), an application is a directed graph of data streams and operators. Each stream is conceptually infinite, and each operator has its own thread of control, which continuously consumes data from input stream(s) and produces data on output streams(s). Often, there are many variants of an operator. For example, an Aggregate operator can use sum, average, maximum, etc. for aggregation, customized for various different types of streaming data, over a sliding window or a tumbling window, and so on. To implement all these variants efficiently, some commercial DSMSs allow users to write their operators as "code generation templates", in other words, as macros that generate custom code for a specific variant of an operator. One such DSMS is IBM's InfoSphere Streams [18], and one of the operators in the standard library of InfoSphere Streams is the `Aggregate` operator. We have re-implemented the `Aggregate` operator from the academic trial version of InfoSphere Streams in *Marco*.

Table 5.3 presents statistics from running the oracle analyzer over the

Code type	Count	Size	Backtracks	Queries	Declarations
Code<*cpp*,*id*>	5	1.00	0.80	4.00	2.00
Code<*cpp*,*type_spec*>	8	6.88	0.00	7.50	1.88
Code<*cpp*,*type_id*>	1	1.00	0.00	3.00	1.00
Code<*cpp*,*expr*>	12	4.75	0.08	3.25	1.58
Code<*cpp*,*stmt*>	40	14.95	1.42	6.88	5.65
Code<*cpp*,*fdef*>	11	33.18	0.64	11.09	8.18
Code<*cpp*,*mdecl*>	21	14.05	0.05	3.95	1.90
Code<*cpp*,*decl*>	13	12.38	0.00	4.77	2.00
Code<*cpp*,*cunit*>	3	7.00	0.00	3.00	1.00

Table 5.3: Oracle analysis results for the fragments in the *Aggregate* operator.

114 fragments in the *Aggregate* operator. We classify these fragments by their code type in the first column and present the number of fragments for each code type in column "Count". The remaining columns average the number of tokens and blanks ("Size"), the number of backtracks during oracle query analysis ("Backtracks"), the number of generated C++ compilation units for queries ("Queries"), and the number of helper declarations to disambiguate the C++ syntax ("Declarations").

The *Aggregate* operator exercises more C++ specific code types than the micro-benchmarks. For instance, the 12 fragments of type **Code**<*cpp*,*mdecl*>, where *mdecl* is the member-declarations non-terminal, generate fields, methods, and constructors in a C++ class. To the best of our knowledge, none of the macro systems in prior work generates members of a C++ class and checks syntactic correctness of the generated code. Due to ambiguity in the C++ grammar, our oracle analyzer backtracked its speculations 72 times over the 114 fragments. Most backtracks appeared in analyzing the fragments that generated C++ statements. These fragments contain lots of unknown identifiers in either variable declarations or expression statements. This is not surprising, because the C/C++ parser differentiates between declarations and statements based on whether or not an identifier has been declared as a type. In MS^2, the programmers must carefully write their macros such that the parsing ambiguity does not appear. Instead of asking the programmer to avoid the ambiguity, our oracle analyzer tries inferring the context, possibly with backtracking. This inference is more important in C++ than in C, because C++ has more ambiguity and types. For instance, C++ templates add more ambiguity in parsing. Even if a variable is bound to a type, the gcc parser for C++ uses backtracking internally, so it comes as no surprise that our oracle also needs to use backtracking.

While our oracle analyzer worked un-aided on 109 fragments in the *Aggregate* operator, we had to provide additional annotations to help it analyze the remaining 5 fragments. The problem was that the error messages from gcc did not provide enough information. To handle these fragments, we improved the C++ support in *Marco* with two additional annotations. The first class of annotations tell the oracle that two blanks have the same value. In our problematic fragments, the equality is guaranteed because the two blanks in the fragment are textually the same expression without side-effects. However, value equality is undecidable in general. The second kind of annotation specifies the enclosing class name for **Code**<*cpp*,*mdecl*> fragments. These fragments can generate constructors and overloaded operators that share the *decl-specifier* and *declarator* non-terminals. In order to tell these two kinds of non-terminals apart, the parser relies deeply on types of identifiers in these two non-terminals. For future work, we are planning to investigate annotation inference.

5.6.3 Scalability

To support an additional target language in a traditional safe macro system, the developer must modify the target-language processor, which is usually a large and complex piece of software. To make matters worse, the modified target-language processor is effectively a branch version, and keeping it up-to-date with the main branch requires additional engineering efforts. On the other hand, to support an additional target language in *Marco*, the developer must write a plug-in consisting of a simplified lexer and three oracles. The oracles wrap unmodified target-language processors. The effort is smaller in the *Marco* approach.

Our C++ plug-in consists of the lexical analyzer and three oracles. For the C++ lexical analysis, we define the `TOKEN` terminal in Figure 5.2. This required a few lines of regular expressions for *identifier* (1), *literal* (5), *keyword* (74), and *preprocessing-op-or-punc* (72) [69]. Most of the regular expressions were trivial, only *identifier* and *literal* (6) required any meta-level operators in regular expressions. Our C++ oracles consists of 1K+ non-blank source lines of code in Java. About half of the source lines are for describing declarations in oracle queries, and the other half are for handling error messages. The error handlers contain 52 regular expressions to classify gcc error messages.

In contrast, to quantify the code size of the gcc compiler itself, we examined the source files under the `cp` directory of gcc. These files contain the C++ front-end that includes the C++ parser [23]. The `cp` directory contains 87K+ non-blank source lines in C source files. It has a hand-written parser in the `parser.c` file that has 14K+ non-blank source lines. Compared to these line counts, our C++ plug-in is much smaller at 1K+ source lines of code. At the same time, it reuses a sophisticated, unmodified code base that has been maintained for years.

Our SQL plug-in consists of the C++ lexical analyzer and the set of three SQL oracle analyzers containing 400 more source lines of code. As a direct consequence of using the C++ lexical analyzer, *Marco* recognizes a subset of SQL tokens. On the other hand,We argue that SQLite has about 1K source lines in the `parser.y` file written in LALR(1) specification. While the SQL plug-in and the SQLite parser have comparable source lines, this result does not necessarily devalue our scalability claim. SQLite has been maintained, adapted, and tested widely for over a decade. It is better to use a proven parser than to reinvent a new one. Furthermore, *Marco* uses not just the SQLite parser, but also other components of SQLite for checking naming discipline.

Discussion: Extensible lexical analysis. We are working on an oracle for lexical analysis that wraps target-language lexical analyzers, such as the Antlr parser generator [58] . The extensible oracle analyzer would call plug-in scanners on the fly. The oracle will recognize *Marco* tokens including back tick. Whenever it finds a back tick and the following target language identifier, it will delegate the scanning work to a plug-in scanner. Each target-language plug-in would count opening and closing brackets in addition to recognizing their own tokens. Whenever it finds the end of a code fragment, it will return to the main driver analyzer.

5.7 Summary

Any program in one language can communicate with any program in another language if both languages have a string type and their programs are represented as strings. Specifically, a program generates another program as a string value and sends it to a compiler or an interpreter to execute. While this programming practice requires no extension to any language or its compiler or interpreter, there is no guarantee that the generated programs are syntactically and semantically correct, and the generation process is hygienic.

To bridge the gap between scalability and safety in code generation interfaces, *Marco* raises the level of abstraction from a string type to a code type. The *Marco* code types are parametrized by a target language and a phrase type. In our code type system, an open fragment represents a set of expressions and statements in target language. Our analyzer synthesizes oracle queries from the open fragment, analyzes the error messages from target language compilers, and infers information about the input fragment. Using information from the oracle analyzers, our static and dynamic checker reports errors in *Marco* programs. In summary, *Marco* presents a scalable analysis for scalable code generation interfaces for any language.

Chapter 6

Related Work

This chapter compares our multilingual tools with previous work that avoids and detects the bugs at foreign function interfaces in Section 6.1, at code generation interfaces in Section 6.2, and across language interfaces in Section 6.3.

6.1 Foreign Function Interfaces Safety

FFI programming is challenging because programmers must reason about multiple languages and their semantic interactions. For example, Chapter 10 of the JNI manual identifies fifteen pitfalls [49]. We list the most serious of these in Table 6.1, using Liang's numbering scheme, and include "bad critical region" from Section 3.3.1 as a 16th pitfall. We created small JNI programs to exercise each pitfall and executed them with HotSpot and J9. Columns two and three show that JNI mistakes cause a wide variety of crashes and silent corruption. The two JVMs behave differently on four of the pitfalls. Columns six and seven show the JVMs are not much better with built-in JNI checking (turned on by the `-Xcheck:jni` command-line flag).

Table 6.1 also compares language designs, static analysis tools, and our *Jinn* implementation. An empty entry indicates that we are not aware of a language feature or static analysis that handles this pitfall. We fill in entries based

	Default Behavior		Language	Static	Dynamic Analysis		
JNI Pitfall	**HotSpot**	**J9**	**Design**	**Analysis**	**HotSpot**	**J9**	***Jinn***
1. Error checking	running	crash	[35], [74]	[43], [48]	warning	error	exception
2. Invalid arguments to JNI functions	running	crash	[35], [74]	[26], [76]	running	crash	exception
3. Confusing jclass with jobject	crash	crash	[35], [74]	[26]	error	error	exception
6. Confusing IDs with references	crash	crash	[35]	[26]	error	error	exception
8. Terminating Unicode strings	running	NPE	[35], [74]		running	NPE	running/NPE
9. Violating access control rules	NPE	NPE	[35], [74]		NPE	NPE	exception
11. Retaining virtual machine resources	leak	leak	[35], [74]	[43]	running	warning	exception
12. Excessive local reference creation	leak	leak			running	warning	exception
13. Using invalid local references	crash	crash	[35]	[43]	error	error	exception
14. Using the JNIEnv across threads	running	crash	[35]		error	crash	exception
16. Bad critical region	deadlock	deadlock	[35]	[43]	warning	error	exception

Table 6.1: JNI pitfalls

JNI pitfalls [49]. *Running*: continues to execute in spite of undefined JVM state. *Crash:* aborts without diagnosis. *Warning*: prints diagnosis and continues. *Error*: prints diagnosis and aborts. *NPE*: raises a null pointer exception. *Exception*: raises a *Jinn* JNI exception.

on our reading of the literature [26, 35, 43, 48, 74, 76]. We did not execute the static tools. Language designs cover the widest class of JNI bugs [35, 74], but new languages require developers to rewrite their code. Static analysis catches some, but not all, pitfalls. For example, statically enforcing non-nullness without language support (e.g., a @NonNull annotation) is undecidable. At the same time, dynamic and static FFI analysis are complementary. Dynamic analysis misses unexercised bugs, whereas static analysis reports false positives.

The last column shows that *Jinn* detects all but one of these serious and common errors. Pitfall 8 depends on how C code uses character buffers and requires analysis or instrumentation of a program's entire C code, which is beyond the scope of our more targeted dynamic analysis. Consequently, the program exhibits the same behavior as a production run without *Jinn*, i.e., it either keeps on running (HotSpot) or signals a null pointer exception (J9). When *Jinn* detects any of the other errors, it throws a JNI failure exception and stops execution to help programmers debug. *Jinn* works out-of-the-box on unmodified JNI which makes it practical for use on existing programs. It systematically finds more errors than all the other approaches.

6.1.1 Safe Interface Languages

Two language designs propose to replace the JNI. SafeJNI [74] combines Java with CCured [57], and Jeannie safely and directly nests Java and C code into each other using quasi-quoting [35]. Both SafeJNI and Jeannie define their language semantics such that static checks catch many errors and both add dynamic checks in translated code for other errors. From a purist perspective, preventing FFI bugs while writing code is more economical than spending time to fix them after the fact. Another approach generates language bindings for annotated C and C++ header files [8, 38]. Ravitch et al. reduce the annotations required for generating idiomatic bindings [62]. *Jinn* is more practical than these approaches, because it does not require developers to rewrite or annotate their code in a different language.

6.1.2 Static FFI Bug Checkers

A variety of static analyses verify foreign function interfaces [25, 26, 43, 48, 75, 76]. All static FFI analysis approaches suffer from false positives because the specification includes dynamic properties, such as non-null reference parameters, valid Java class and method names in string parameters, and less than 16 local references. Static analysis cannot typically guarantee these properties. For instance, J-Saffire reports false positives and warnings [26]; Tan et al. report a false positive rate of 15.4% [48]; and BEAM reports a false positive, while missing the bug in Section 3.1.1. In contrast, *Jinn* never generates false positives but only finds bugs actually triggered during program execution. Furthermore, whereas prior static analyses for JNI require the native library to be written in C and available in source form, *Jinn* is neither restricted to C nor does it require source code access. For instance, *Jinn* found FFI bugs in the Subversion Java binding written in C++. In summary, static analysis finds a subset of FFI bugs without executing the program but suffers from false positives. In comparison, *Jinn* finds more FFI bugs but only when they are exercised; suffers from no false positives; and requires no source code access.

6.1.3 Dynamic FFI Bug Checkers

Some JVMs provide built-in dynamic JNI bug checkers, enabled by the `-Xcheck:jni` command-line flag. While convenient, these error checkers only cover limited classes of bugs, and JVMs implement them inconsistently. NaturalBridge's BulletTrain ahead-of-time Java compiler performed several ad-hoc JNI integrity checks on language transitions [56].

Jinn covers a larger class of JNI bugs, works consistently with any JVM that implements the JVM Tools Interface (JVMTI), and explicitly throws an exception at the point of failure. Exceptions provide a principled and language supported approach to software quality — for example, enabling a GUI-based program to report the bug in a dialog instead of relying on the user to sift through the system log. Furthermore, when the exception's error message and calling context do not suffice to identify the cause of the failure, programmers can rerun the program with both *Jinn* and a Java debugger. The debugger then catches the exception, and the programmer can access the detailed program state at the point of failure.

6.1.4 State Machine Specifications

Several programmable bug checkers take state machine specifications, and report errors when state machines reach error states. Dwyer et al. survey state-machine driven static analyses [20]. For instance, Metal [21] and SLIC [7] for general program properties are languages for specifying state machines that are then used to find bugs through static analysis. On the dynamic side, Allan et al. turn FSMs into dynamic analyses by using aspect-oriented programming [1]; Chen and Rosu synthesize dynamic analyses from a variety of specification formalisms, including FSMs [14]; and Arnold et al. implement

FSMs for bug detection in a JVM, controlling the runtime overhead by sampling [2]. While in principle these specification languages are expressive enough to describe many FFI constraints, in practice none of them address the unique challenges of multi-lingual software. Also, unlike *Jinn*, most of them require source code access.

6.2 Code Generation Interface Safety

In using code generation interfaces, the generator programs should respect the rules of syntax, scope, and semantics in the target languages. Safe programming systems verify these rules at the cost of being deeply coupled to their target languages (Section 6.2.1). Language-agnostic systems do not check safety, or they do not provide good error messages (Section 6.2.2). Language-agnostic syntax embedding systems do not take advantage of the engineering efforts in compilers and interpreters for target languages (Section 6.2.3). *Marco* bridges the gap between safety, language agnostics, and engineering efforts by analyzing the error messages from production compilers and interpreters (Section 6.2.4).

6.2.1 Language-Specific Safe Macro Systems

Some programming languages check safety of generated code by deeply coupling meta-languages and their target languages. Multi-stage programs dynamically generate and execute safe code in Scheme, ML, C, and Java [17, 55, 59, 85]. There are meta-programming systems for ensuring safe generation of HTML documents and SQL queries [15, 16, 65]. C++ *concepts* add constraints to template declarations to produce error messages before fully expanding templates and discovering errors [19, 29]. Researchers added types to C macros [53, 84]. Like *Marco*, these systems check safety in macros, but unlike *Marco*, they are language-specific.

Syntax. The problem of respecting syntax rules is well-recognized in general-purpose imperative language communities and in the web programming area [3, 4, 53, 84]. Syntax analysis faces huge challenges when the target languages have rich syntax and when macros are dynamic. For target language with rich syntax, MS^2 abstracted the level of macro processing from tokens to abstract syntax trees [84]. ASTEC proposed a refactoring approach for legacy C programs [53]. For dynamic code generation, Minamide statically checks the syntax of the generated HTML pages from a PHP program by finding a regular expression that over-approximates these HTML pages [54]. Apollo explores a dynamic random-testing approach [3, 4]. All these systems directly recognize the syntax of their target languages only, while the *Marco* system leverages unmodified target language compilers.

Some meta-programming systems eliminate security vulnerabilities in web applications [12, 32, 40, 83]. Static analyzers deeply track the string values and their operations in host-language programs to find out the code and inputs that breach security policy [32, 40, 83]. `StringBorg` eliminates these vulnerabilities by raising the abstraction of inputs from strings to syntax trees [12]. *Marco* abstracts the input values and their operations to the lexical level that is sufficient to eliminate these vulnerabilities.

Scope. The problem of respecting scope rules has been addressed by work on *hygiene* in the functional language community [41, 42]. Kohlbecker et al. introduced hygienic expansion [42]. Clinger and Rees presented an improved algorithm for renaming identifiers to guarantee hygiene [17]. In contrast, *Marco* does not automatically rename identifiers, but rather reports errors when identifiers are accidentally captured. Kim et al. reached a complete system that formally characterizes both accidental captures and intentional captures [41]. All these systems depend on the syntax and scope rules in their specific target language. On the other hand, the *Marco* system indirectly recognizes scopes by querying target language compilers.

Semantics. Some systems check whether or not all the expanded fragments will pass type checking in their target languages. C++ concepts add contracts to templates [29]. MorphJ verifies some contracts statically so that expanded code will not have name-resolution conflicts [37]. Quail checks types between SQL queries and the database system [77]. Target-language agnostic type checking is an open problem that has not been addressed by any of these systems, and that we have not addressed it in *Marco* either.

All the safe macro systems are tightly coupled to specific target languages. Instead of modifying the target language compilers, *Marco* relies on compilation error messages to infer all these properties. Production-level compiler writers are strongly motived to generate high-quality error messages to serve their users. Based on this assumption, we believe that our weakly coupled analysis approach is the right direction for safely handling many programming languages.

6.2.2 Language-Agnostic Unsafe Macro Systems

Language-agnostic macros are quite common in practice, because most programming languages have string data types that can represent both well-formed and ill-formed programs in any target languages. A JSP web program generates SQL queries and HTML/JavaScript pages to talk to back-end database systems and front-end web browsers, respectively. The C preprocessor does not incorporate much information about its target language, because it takes tokenized streams as input and output. Unfortunately, none of these language-agnostic macro systems provide safety checks. Ernst et al. present an empirical study that finds that programmers often break safety rules when using the C preprocessor [22]. Reading ill-formed generated-code and locating the erroneous code is hard and tedious.

Compared to these systems, our *Marco* system adds safety checks while remaining expressive and language-agnostic. The *Marco* system relies on high-quality error messages from its target language compilers, while these unsafe system assume nothing. We believe that our assumption aligns well with compiler writers who want to give good explanations for compilation failures. The *Marco* language raise the abstraction level of target programs from character strings or token sequences to fragments. A fragment's type constrains both its target-language and its non-terminal, so that *Marco* can check syntactic well-formedness for each fragment in isolation. Furthermore, *Marco* checks for naming discipline.

6.2.3 Language-Agnostic Syntax Embedding Systems

Language-agnostic syntax embedding systems offer extensible grammars to embed guest-language fragments into host-language programs. `MetaB-`

`org` [13] and `StringBorg` [12] propose scannerless generalized LR parsing to extend grammars (embedding) and define their transformation rules (assimilation). `Metafront` reduces the overhead from scannerless parsing [11]. While *Marco* shares the goal of language agnostics and error checking, it relies on the guest-language compilers and interpreters using oracle queries. This approach adds a few strengths over these systems. *Marco* takes advantage of the highly tuned parsers, error reports, and scope analyses in guest-language compilers and interpreters. Unlike these systems, *Marco* resolves context-sensitivity and grammatical ambiguity in C++, produces human readable error reports for fragments, and enforces naming disciplines in expanding the blanks in fragments.

6.2.4 Using Messages from Black-Box Compilers

A few systems consume error messages from compilers and interpreters for a variety of reasons. SEMINAL takes error messages from the OCaml and `g++` compilers and suggest changes for ill-formed programs [46]. Autoconf macros generates C/C++ programs, and send them to C/C++ compilers. They check error messages to determine whether or not some header files and some preprocessor symbols are available in the build host environment. The HelpMeOut system mines IDE logs to discover common fixes, and then proposes them to programmers based on which error messages are displayed [34]. Like *Marco*, each of these systems runs unmodified language compilers, and then inspects their error messages for clues. Unlike *Marco*, none of these systems is a macro system. To our knowledge, *Marco* is the first system that mines error messages from black-box compilers for safe code generation.

6.3 Multilingual Debuggers

While programmers have adapted high-level languages such Java, JavaScript, and Python with managed runtime environments in addition to the legacy native C environment, debuggers do not recognize them completely. One contribution of this thesis is an implementation of the most portable and powerful debugger for Java and C to date. Blink's power and portability derives from composing existing powerful and portable debuggers.

6.3.1 Mixed-Environment Debuggers

The closest work to mixed-environment debugging is by White, who describes a manual technique for mixed-environment debugging for Java and C that attaches single-environment debuggers to the same process [86, 87]. The resulting system is limited because it cannot examine a mixed stack, step into cross-environment calls or set breakpoints in one environment when stopped in the other, all of which Blink supports.

We are aware of three mixed-environment debuggers (`dbx`, XDI, and the Visual Studio debugger) that are practical but, unlike *Blink*, do not use a compositional approach. These debuggers are not easily extended nor are they portable.

The `dbx` debugger extends an existing C debugger for Java [73]. XDI extends an existing Java debugger for C [60]. Both XDI [60] and `dbx` [73] are powerful but they are less portable than Blink. XDI works only with the Harmony JVM, which is a non-standard JVM. `Dbx` only works with Sun's JVM on Solaris, and, with limited functionality, on Linux. Because we use composition, Blink is more portable; it supports multiple JVMs (HotSpot and J9) and C debuggers (`cdb` and `gdb`) on both Linux and Windows.

The Visual Studio debugger debugs C#, C, C++, and other .NET languages in the CLR (Common Language Runtime) [67]. It is also extensible through debug engines [82]. However, in contrast to Blink, where multiple debuggers attach to a single mixed-environment program, each Visual Studio's debug engine is responsible for one program. The CLR provides two debugging APIs: one native and one managed. To handle a mixed-environment program, a debug engine must use both APIs. Given two CLR debuggers, one for the native API and one for the managed API, our compositional approach would yield a mixed-environment debugger.

6.3.2 Single-Environment Multilingual Debuggers

Some multilingual debuggers require all the languages to implement a single interface in the same environment [10, 52, 64]. For example, the GNU debugger, `gdb`, can debug C together with a subset of Java statically compiled by `gcj` [10]. Many real-world Java applications, however, exceed the `gcj` subset and require a full JVM to run. Compared to these approaches, ours is the only one that leverages independently developed debuggers.

6.3.3 Portable Debuggers

Portability of debuggers depends on their construction mechanisms: *reverse engineering* or *instrumentation*. In the reverse engineering model, debuggers interpret machine-level state with symbol tables emitted by compilers, and generalize the symbol table formats to add more platforms. For instance, `dbx`, `gdb`, and `ldb` recognize portable symbol table formats including dbx "stabs" [52], DWARF [24], and even PostScript [61]. In the instrumentation model, a debuggee process executes its debugger code. By construction, the instrumentation-based debuggers are as portable as the languages of the in-process debuggers. For instance, TIDE [80], `smld` [79], and Hanson's machine-independent debugger [33] do not need any extra effort for additional platforms. However, instrumentation causes a factor 3–4 slowdown, which may impede adoption.

Blink leverages portability of its component debuggers, and the construction mechanisms are portable. For reverse engineering, the symbol table for Jeannie discussed in Section 4.6 is platform-independent. For instrumentation, the intermediate agent has only 10–20 lines of low-level assembly code.

6.3.4 Mixed-Language Interpreters

One contribution of this dissertation is Blink's read-eval-print loop (REPL) for mixed Java and C expressions. Debuggers that support multiple languages, such as `gdb`, often include an interpreter for expressions in each language. Blink is novel in that it interprets expressions by delegating subexpressions to the appropriate single-language debuggers. Blink's REPL uses a syntax for embedding Java in C (and vice versa) that was developed in an earlier paper on Jeannie [35]. The Jeannie paper described the language and its compiler but did not describe an interpreter, let alone a debugger.

Chapter 7

Conclusion

Programs are increasingly written in a variety of languages as programmers take advantage of new innovative languages and legacy libraries. Although multilingual programming is inevitable in any real-world software projects, it requires lots of expertise to write correct multilingual programs. As a direct consequence, multilingual programs are full of bugs, and debugging is notoriously tedious and painful.

We showed that multilingual programming tools can be built with relatively low effort by combining single-language tools. Our tooling experience and experiment results indicate that tool composition is scalable and effective. We avoid re-implementing what single-language tools provide. The composed tools help programmers to debug and fix multilingual programs. The insight is that multilingual programming interfaces define language boundaries and their correctness conditions. We next apply the principle of interposition to composing tools that recognize language boundaries and use single-language tools. Our tools include a dynamic checker for foreign function interfaces (*Jinn*), an interactive debuggers (*Blink*), and a safe macro language for code generation interfaces (*Marco*).

As part of this dissertation, we introduced a taxonomy for describing multilingual programming interfaces. Foreign function interface rules capture key language differences in thread state, types, and resources. Code generation interface rules respect language constructs in syntax, scope, and semantics. This dissertation also offers following contributions and tools:

1. The first complete dynamic JNI analysis, a partial specification of Python/C, and an approach that automatically generates them from the specifications and mapping functions.

2. The first fully functional Java and C debugger.

3. The first system for agnostically analyzing code generators for code generation interfaces.

Our compositional approach will influence language designers, tool developers, and programmers. Language designers would document FFI rules using our classification when they introduce innovative programming languages. Tool developers would begin composing multilingual programming tools by following our approach. Programmers will write more correct programs using composable multilingual programming tools.

Bibliography

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 345–364, October 2005.

[2] Matthew Arnold, Martin Vechev, and Eran Yahav. QVM: An efficient runtime for detecting defects in deployed systems. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 143–162, October 2008.

[3] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Practical fault localization for dynamic web applications. In *International Conference on Software Engineering (ICSE)*, pages 265–274, May 2010.

[4] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in dynamic web applications. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 261–272, July 2008.

[5] Jonthan Bachrach and Keith Playford. The java syntactic extender (JSE). In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 31–42, October 2001.

[6] Chris Bailey. Java technology, IBM style: Introduction to the IBM developer kit. http://www.ibm.com/developerworks/java/librar urly/j-ibmjava1.html, May 2006.

[7] Thomas Ball and Sriram K. Rajamani. SLIC: a specifcation language for interface checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research, January 2002.

[8] David M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *USENIX Tcl/Tk Workshop*, pages 129–139, July 1996.

[9] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 169–190, October 2006.

[10] Per Bothner. Compiling Java with GCJ. http://www.linuxjournal url.com/article/4860, January 2003.

[11] Claus Braband, Michael I. Schwartzbach, and Mads Vanggaard. The `metafront` system: Extensible parsing and transformation. Technical Report BRICS RS-03-7, BRICS, February 2003.

[12] Martin Bravenboer, Eelco Dolstra, and Eelco Visser. Preventing injection attacks with syntax embeddings. In *Generative Programming and*

Component Engineering (GPCE), pages 3–12, October 2007.

[13] Martin Bravenboer and Eelco Visser. Concrete syntax for objects. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 365–383, October 2004.

[14] Feng Chen and Grigore Rosu. MOP: An efficient and generic runtime verification framework. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 569–588, October 2007.

[15] Adam Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Operating systems design and implementation (OSDI)*, October 2010.

[16] Adam Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 122–133, June 2010.

[17] William Clinger and Jonathan Rees. Macros that work. In *Principles of Programming Languages (POPL)*, pages 155–162, January 1991.

[18] IBM Corporation. InfoSphere Streams: Stream processing system. `http://www-01.ibm.com/software/data/infosphere/streams/`.

[19] Gabriel Dos Reis and Bjarne Stroustrup. Specifying C++ concepts. In *Principles of Programming Languages (POPL)*, pages 295–308, October 2006.

[20] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *International Conference on Software Engineering (ICSE)*, pages 411–420, May 1999.

[21] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Operating systems design and implementation (OSDI)*, October 2000.

[22] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, December 2002.

[23] Free Software Foundation, Inc. GNU compiler collection (GCC) internals. `http://gcc.gnu.org/onlinedocs/gccint/`.

[24] Free Standards Group. DWARF 3 debugging information format. `http://www.dwarfstd.org/Dwarf3.pdf`, December 2005.

[25] Michael Furr and Jeffrey S. Foster. Checking type safety of foreign function calls. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 62–72, June 2005.

[26] Michael Furr and Jeffrey S. Foster. Polymorphic type inference for the JNI. In *European Symposium on Programming (ESOP)*, pages 309–324, March 2006.

[27] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.

[28] David Greenfieldboyce and Jeffrey S. Foster. Type qualifier inference for Java. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 321–336, October 2007.

[29] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: Linguistic support for generic programming in C++. pages 291–310, October 2006.

[30] Robert Grimm. xtc — eXTensible C. `http://www.cs.nyu.edu/rgrimm/xtc/`.

[31] Robert Grimm. Better extensibility through modular syntax. In *Programming Language Design and Implementation (PLDI)*, June 2006.

[32] William G. J. Halfond, Ro Orso, and Panagiotis Manolios. Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *Foundation of Software Engineering (FSE)*, pages 175–185, November 2006.

[33] David R. Hanson. A machine-independent debugger—revisited. *Software: Practice and Experience (SPE)*, 29(10):849–862, 1999.

[34] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. What would other programmers do: suggesting solutions to error messages. In *ACM Conference on Human Factors in Computing Systems (CHI)*, pages 1019–1028, April 2010.

[35] Martin Hirzel and Robert Grimm. Jeannie: Granting Java native interface developers their wishes. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 19–38, October 2007.

[36] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 32–43, June 1992.

[37] Shan Shan Huang and Yannis Smaragdakis. Expressive and safe static reflection with MorphJ. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 79–89, June 2008.

[38] Alan Kaplan, John Bubba, and Jack C. Wileden. The Exu approach to safe, transparent and lightweight interoperability. In *IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 393–394, October 2001.

[39] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, April 1988.

[40] Adam Kieyzun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *International Conference on Software Engineering (ICSE)*, pages 199–209, May 2009.

[41] Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. A polymorphic modal type system for Lisp-like multi-staged languages. In *Principles of Programming Languages (POPL)*, pages 257–268, January 2006.

[42] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *ACM Conference on LISP and Functional Programming (LFP)*, pages 151–161, August 1986.

[43] Goh Kondoh and Tamiya Onodera. Finding bugs in Java native interface programs. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 109–118, July 2008.

[44] Jay A. Kreibich. *Using SQLite*. O'Reilly Media, Inc., 1st edition, 2010.

[45] Byeongcheol Lee, Martin Hirzel, Robert Grimm, and Kathryn S. McKinley. Debug all your code: Portable mixed-environment debugging. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 207–226, 2009.

[46] Benjamin Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. Searching for type-error messages. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2007.

[47] Xavier Leroy. The Objective Caml System Release 3.12. `http://caml.inria.fr/distrib/ocaml-3.12/ocaml-3.12-refman.pdf`, April 2010.

[48] Siliang Li and Gang Tan. Finding bugs in exceptional situations of JNI programs. In *ACM conference on Computer and communications security (CCS)*, pages 442–452, November 2009.

[49] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, 1999.

[50] Jorn Lind-Nielsen. BuDDy. `http://buddy.sourceforge.net/`.

[51] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, September 1996.

[52] Mark A. Linton. The evolution of Dbx. In *Usenix Technical Conference*, 1990.

[53] Bill McCloskey and Eric Brewer. ASTEC: A new approach to refactoring C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 21–30, September 2005.

[54] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *International World Wide Web Conference (WWW)*, pages 432–441, May 2005.

[55] Eugenio Moggi, Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming Languages and Systems (ESOP)*, pages 193–207, March 1999.

[56] NaturalBridge. BulletTrain JNI Checking Examples. `http://web.archive.org/web/*/http:///www.naturalbridge.com/jnichecking.html`, January 2001.

[57] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Principles of Programming Languages (POPL)*, pages 128–139, January 2002.

[58] Terence Parr. *The Definitive ANTLR Reference*. The Pragmatic Programmers, May 2007.

[59] Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. ‘C and tcc: A language and compiler for dynamic code genera-

tion. *Transactions on Programming Languages and Systems (TOPLAS)*, 21(2):324–369, March 1999.

[60] Vitaly Providin and Chris Elford. Debugging native methods in Java applications. In *EclipseCon User Conference*, March 2007.

[61] Norman Ramsey and David R. Hanson. A retargetable debugger. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 22–31, June 1992.

[62] Tristan Ravitch, Steve Jackson, Eric Aderhold, and Ben Liblit. Automatic generation of library bindings using static analysis. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 352–362, June 2009.

[63] Jonathan B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture*. John Wiley & Sons, 1996.

[64] Sukyoung Ryu and Norman Ramsey. Source-level debugging for multiple languages with modest programming effort. In *International Conference on Compiler Construction (CC)*, 2005.

[65] Anders Sandholm and Michael I. Schwartzbach. A type system for dynamic web documents. In *Principles of Programming Languages (POPL)*, pages 290–301, January 2000.

[66] Joseph Siefers, Gang Tan, and Greg Morrisett. Robusta: taming the native beast of the jvm. In *ACM conference on Computer and communications security (CCS)*, pages 201–211, November 2010.

[67] Mike Stall. Mike Stall's .NET debugging blog. `http://blogs.msdn.com/jmstall/default.aspx`.

[68] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.

[69] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 2000.

[70] Sun Microsystems, Inc. Bug database Bug 4207056 was opened 1999-01-29. `http://bugs.sun.com`.

[71] Sun Microsystems, Inc. Java SE HotSpot at a glance. `http://java.sun.com/javase/technologies/hotspot/`.

[72] Sun Microsystems, Inc. JVM™ tool interface, version 1.1. `http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html`, 2006.

[73] Sun Microsystems, Inc. Debugging a Java application with dbx. `http://docs.sun.com/app/docs/doc/819-5257/blamm?a=view`, 2007.

[74] Gang Tan, Andrew W. Appel, Srimat Chakradhar, Anand Raghunathan, Srivaths Ravi, and Daniel Wang. Safe Java native interface. In *IEEE International Symposium on Secure Software Engineering (ISSSE)*, pages 97–106, 2006.

[75] Gang Tan and Jason Croft. An empirical security study of the native code in the JDK. In *Usenix Security Symposium*, pages 365–377, July 2008.

[76] Gang Tan and Greg Morrisett. ILEA: Inter-language analysis across Java and C. In *ACM Conference on Object-Oriented Programming Systems and Applications (OOPSLA)*, pages 39–56, October 2007.

[77] Zachary Tatlock, Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. Deep typechecking and refactoring. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 37–52, October 2008.

[78] The GNOME Project. GNOME bug tracking system. Bug 576111 was opened 2009-03-20. `http://bugzilla.gnome.org`.

[79] Andrew P. Tolmach and Andrew W. Appel. Debugging standard ML without reverse engineering. In *LISP and Functional Programming (LFP)*, 1990.

[80] Mark van den Brand, Bas Cornelissen, Pieter Olivier, and Jurgen Vinju. TIDE: A generic debugging framework — tool demonstration. *Electronic Notes in Theoretical Computer Science*, 141(4), 2005.

[81] Guido van Rossum and Fred L. Drake. *Python/C API Manual - PYTHON 2.6: Python documentation MANUAL Part 4*. CreateSpace, Paramount, CA, 2009.

[82] Visual studio debugger extensibility. `http://msdn.microsoft.com/en-us/library/bb161718(VS.80).aspx`.

[83] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 32–41, June 2007.

[84] Daniel Weise and Roger Crew. Programmable syntax macros. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 156–165, June 1993.

[85] Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. Mint: Java multi-stage programming using weak separability. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 400–411, 2010.

[86] Matthew White. Debugging integrated Java and C/C++ code. `http://web.archive.org/web/20041205063318/www-106.ibm.com/developerworks/java/library/j-jnidebug/`, November 2001.

[87] Matthew White. Integrated Java technology and C debugging using the Eclipse platform. In *JavaOne Conference*, 2006.

[88] Ben Wiedermann, Ali Ibrahim, and William R. Cook. Interprocedural query extraction for transparent persistence. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 19–36, October 2008.

[89] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging.* Morgan Kaufmann, October 2005.

[90] Craig Zilles. Accordion arrays: Selective compression of unicode arrays in Java. In *ACM International Symposium on Memory Management (ISMM)*, pages 55–66, June 2007.

Vita

Byeongcheol Lee was born in Jeonju, S. Korea on August 30, 1976, the second of two sons and one daughter of Dr. Sanghyun Lee and Gymok Chang. He received a double B.E. in Electronic & Electrical Engineering and Computer Science & Engineering, *magna cum laude* from POSTECH, S. Korea in August 2004. In September 2004, he entered the Department of Computer Sciences at The University of Texas at Austin. In May 2006, he received a Master of Arts degree in Computer Science. Then, he continued onto this Ph.D. program.

Permanent address: 103-304, Seogok Hyundai APT, Hyoja3-ga,
Wansan-gu, Cheunju, Cheonbuk, South Korea

This dissertation was typeset with LaTeX[†] by the author.

[†] LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.