SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2014)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2276

Debugging mixed-environment programs with Blink

Byeongcheol Lee[1,*,†], Martin Hirzel[2], Robert Grimm[3] and Kathryn S. McKinley[4]

[1]*Gwangju Institute of Science and Technology, Gwangju, Korea*
[2]*IBM, Thomas J. Watson Research Center, Yorktown Heights, NY, USA*
[3]*New York University, New York, NY, USA*
[4]*Microsoft Research, Redmond, WA, USA*

SUMMARY

Programmers build large-scale systems with multiple languages to leverage legacy code and languages best suited to their problems. For instance, the same program may use Java for ease of programming and C to interface with the operating system. These programs pose significant debugging challenges, because programmers need to understand and control code across languages, which often execute in different environments. Unfortunately, traditional multilingual debuggers require a *single* execution environment. This paper presents a novel *composition* approach to building portable mixed-environment debuggers, in which an intermediate agent 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 (i) simple: it requires modest amounts of new code; (ii) portable: it supports multiple Java virtual machines, C compilers, operating systems, and component debuggers; and (iii) powerful: composition eases debugging, while supporting new mixed-language expression evaluation and Java native interface bug diagnostics. To demonstrate the generality of interposition, we build prototypes and demonstrate debugger language transitions with C for five of six other languages (Caml, Common Lisp, C#, Perl 5, Python, and Ruby) without modifications to their debuggers. Using real-world case studies, we show that diagnosing language interface errors require prior single-environment debuggers to restart execution multiple times, whereas Blink directly diagnoses them with one execution.

Received 22 July 2013; Revised 17 April 2014; Accepted 2 May 2014

KEY WORDS: debuggers; foreign function interface; JNI; composition

1. INTRODUCTION

Software developers resort to multiple languages to leverage legacy code and existing libraries and, when possible, use a language well suited to their needs for new code. Large programs are hard to get correct, even when written in a single language, because an individual developer is typically an expert on only a small fraction of the code. Mixed-language programs require additional developer expertise, and language interfaces add another source of errors. For example, the literature reports hundreds of mixed-language interface bugs [1–4]. Foreign function interfaces (FFIs) such as Java native interface (JNI) and Python/C consist of voluminous and complex programming rules. For example, there are 1500+ rules in JNI [5]. The effects of interface bugs are not defined, and they very often crash programs immediately or insidiously. Traditional debuggers offer little or no information that the programmer can use to correct such bugs.

*Correspondence to: Byeongcheol Lee, School of Information and Communications, Gwangju Institute of Science and Technology, 123 Cheomdangwagi-ro, Buk-gu, Gwangju 500-712, Korea.
†E-mail: byeong@gist.ac.kr

Traditional debuggers are not much help with mixed-language programs because they are limited to a *single execution environment*. For example, native programs and their debuggers (e.g., the `gdb` debugger for C, C++, and Fortran) require language implementations to use the same application binary interface (ABI). The ABI is machine dependent and thus precludes portable execution environments for *managed languages*, such as Java, C#, JavaScript, and Python. For portability, managed languages rely on virtual machine (VM) execution, using interpretation, just-in-time compilation, and garbage collection. They abstract over internal code, the stack, and data representations. Debuggers for managed languages, such as the standard Java debugger `jdb`, operate on VM abstractions, for example, through the Java Debug Wire Protocol (JDWP), but do not understand native code. Current mixed-language debuggers are limited to XDI and `dbx`, which support Java and C within a single JVM [6, 7], and the Visual Studio debugger, which supports managed and native code in the Common Language Runtime (CLR) [8]. While these debuggers understand all environments, they are behemoths that are generally not portable. The challenge when building a mixed-environment debugger is that each environment has different representations; managed debuggers operate at the level of bytecodes and objects, whereas native debuggers deal with machine instructions and memory words.

This article presents a novel *debugger composition* design for building mixed-environment debuggers that uses *runtime interposition* to control and reuse existing single-environment debuggers. An intermediate agent instruments and controls all language transitions. We show composition with interposition is sufficient to implement the three pillars of debugging functionality: execution control, context management, and data inspection [9]. The result is a simple, portable, and powerful approach to building debuggers.

We implement this approach in Blink, a debugger for Java, C, and the Jeannie programming language [10]. Because Blink reuses existing debuggers, it is simple: Blink requires 9K lines of new code, half of which implements interposition. Blink is portable: it supports multiple Java virtual machines or JVMs (Oracle and IBM), C compilers (GNU and Microsoft), and operating systems (Unix and Windows). By comparison, `dbx` works only with Oracle's JVM and XDI works only with the Harmony JVM.

We also explore how well our composition approach generalizes to other languages and what are its requirements. We implemented a simple prototype of the language interposition approach for five of the six standard debuggers for Caml, Common Lisp, C#, Perl 5, Python, and Ruby. Our prototypes implement each language's FFI to C. Because the Caml debugger lacks the ability to evaluate functions on which our interposition approach depends, it requires changes to the existing debugger to compose. We use the function evaluation in the C# debugger to implement interposition. We simply interpose on the interpreters for Common Lisp, Perl 5, Python, and Ruby. These case studies indicate that debugger composition is viable in many language settings.

Debugger composition furthermore facilitates powerful new debugging features: (i) a read–eval–print loop (REPL) that, in Blink, evaluates mixed Java and C expressions in the context of a running program and (ii) a dynamic bug checker for two common JNI problems. We implement these features in Blink. This article demonstrates this functionality using several case studies, which reproduce bugs found in real programs and compare debugging with other tools to debugging with Blink. The other tools crash, silently ignore errors, or require multiple program invocations to diagnose a bug, whereas Blink typically identifies the bug directly in a single program invocation. The result is a debugger that helps users effectively find bugs in mixed-language programs.

To summarize, the contributions of this work are as follows:

1. A new approach to building mixed-environment debuggers that composes single-environment debuggers. Prior debuggers either support only a single environment or re-implement functionality instead of reusing it.
2. Blink, an implementation of this approach for Java, C, and Jeannie, which is simple, portable, powerful, and open source [11].
3. Two advanced new debugger features: a mixed-environment interpreter and a dynamic checker for detecting JNI misuse.

DOI: 10.1002/spe

4. A prototype of language interposition for six additional languages and their debuggers which shows that our approach generalizes.
5. A discussion of the requirements and multiple mechanisms for implementing language execution environment composition.
6. Case studies showcasing the experience of debugging real-world bugs with Blink.

This article extends our prior publication on Blink [12] in two main ways: (i) it demonstrates how Blink helps diagnose real-world bugs (Section 8) and (ii) it elaborates on and reports on the implementations of several mechanisms that show debugger composition generalizes to other language environments (Section 6). Our experiences with Blink inspired us to develop another tool, called Jinn [5], to identify a wide variety of JNI automatically at runtime, but even with such tools, programmers still must discover the root cause of their bugs. Blink helps programmers answer the critical questions about their bugs, such as 'Where am I?' 'How did I get here?' and 'What variable values are the root cause?'

The remainder of the article is organized as follows. Section 2 motivates multilingual debugging with an example. Section 3 shows how to design an agent that composes debuggers, controls execution, and answers programmers queries, regardless of language context. Section 4 explains advanced debugging features, such as multilingual expression evaluation. Section 5 describes implementation details, such as tracking and reporting stack context, setting break points in C when stopped in Java code, and stepping across language interfaces. Section 6 discusses and reports on prototypes that generalize our composition approach to other languages. The evaluation in Section 7 shows that Blink is portable across architectures, composes a wide variety of vastly different debugger and language implementations, and adds little overhead to debugging. Section 8 walks through two case studies of challenging multilingual errors that crash programs and shows that Blink helps developers find these bugs quickly. Section 9 discusses related work, and Section 10 concludes.

2. MOTIVATION: A LANGUAGE INTERFACE BUG

This section uses a real-world multilingual bug to illustrate that debugging across language interfaces with current tools is at best painful and that Blink significantly improves the debugging experience.

Consider the code in Figure 1, which distills fragments from the Eclipse SWT windowing toolkit and the *java-gnome* Java binding for the GNOME desktop to illustrate a common class of JNI bugs that is due to JNI's reflection-like API [13]. Execution starts at line 6 in Java code. Line 8 calls the `dispatch` method, passing either `"mouseEvent"` or `"keyboardEvent"` as a parameter. The `dispatch` method is declared in Java (line 10) but defined in C (line 20). Line 22 calls another C function, `call_java_wrapper`, defined in line 24. Line 28 looks up the Java method identifier (`mid`) based on the parameter string. This lookup fails for `"keyboardEvent"` because of the capitalization error (line 15 expects `"keyBoardEvent"`). With the current state of the art, this bug is difficult to diagnose. For example, executing Oracle's JVM with the `-Xcheck:jni` flag results in the following output:

```
FATAL ERROR in native method:
    JNI call made with exception pending
at EventHandlerBug.dispatch(Native Method)
at EventHandlerBug.main(EventHandlerBug.java:8)
```

This call stack shows only Java line numbers and does not mention the C function `call_java_wrapper` where the error occurs. The user would at best inspect the code to find JNI calls and then re-execute the program with breakpoints potentially on all JNI operations. Existing static bug detectors do not find this problem either, because they do not currently handle the array lookup and string manipulation on line 8, which are difficult to analyze statically [2, 3, 14].

Blink improves over both approaches—it detects the invalid JNI usage, automatically inserts a breakpoint, and prints the following diagnostic message:

DOI: 10.1002/spe

EventHandlerBug.java

```
 1. public class EventHandlerBug {
 2.   static { System.loadLibrary("NativeLib"); }
 3.   static final String[] EVENT_NAMES
 4.      = { "mouse", "keyboard" };
 5.   public static void main(String[] args) {
 6.     int idx = Integer.parseInt(args[0]);
 7.     assert(0 <= idx && idx < EVENT_NAMES.length);
 8.     dispatch(EVENT_NAMES[idx] + "Event");
 9.   }
10.   static native void dispatch(String m1);
11.   static void mouseEvent() {
12.     System.out.println("mouse clicked");
13.   }
14.   /* cause: 'keyboard' vs. 'keyBoard' mismatch */
15.   static void keyBoardEvent() {
16.     System.out.println("key pressed");
17.   }
18. }
```

EventHandlerBug.c

```
19. #include <jni.h>
20. void EventHandlerBug_dispatch(JNIEnv* env,
21.                              jclass cls, jstring m1) {
22.   call_java_wrapper(env, cls, m1);
23. }
24. static void call_java_wrapper(JNIEnv* env,
25.                              jclass cls, jstring jstr) {
26.   const char* cstr = (*env)->GetStringUTFChars(
27.     env, jstr, NULL);
28.   jmethodID mid = (*env)->GetStaticMethodID(
29.     env, cls, cstr, "()V");
30.   /* effect: attempted call with invalid 'mid' */
31.   (*env)->CallStaticVoidMethod(env, cls, mid);
32.   (*env)->ReleaseStringUTFChars(env, jstr, cstr);
33. }
```

Figure 1. Example bug: a typo in Java code (line 15) causes a crash in C code (line 31).

```
JNI warning:
Missing Error Checking: CallStaticVoidMethod
[1] call_by_name_wrapper (EventHandlerBug.c:31)
[2] Java_EventHandlerBug_dispatch (EventHandlerBug.c:22)
[3] EventHandlerBug.main (EventHandlerBug.java:8)
blink> _
```

This message shows the mixed C and Java stack and identifies the call at line 31 as erroneous. Because `mid` is invalid, the user would next determine that `mid` is derived from the string `cstr` and print `cstr`:

```
blink> print cstr
"keyboardEvent"
```

Variable `cstr` holds `"keyboardEvent"` instead of `"keyBoardEvent"`, but where does that value come from? Line 8, mentioned in the original stack trace, contains the expression `EVENT_NAMES[idx]+"Event"`. To examine the Java array from the C breakpoint, the user employs Blink's mixed-language expression evaluation as follows:

```
blink> print `EventHandlerBug.EVENT_NAMES[1]
"keyboard"
```

To fix the bug, the user would change either the string in `EVENT_NAMES[1]` or the method name in line 15.

DOI: 10.1002/spe

3. DEBUGGER COMPOSITION APPROACH

This section describes our approach to building mixed-environment debuggers by composing them out of single-environment debuggers. We use our implementation of Blink for Java and C as our running example. Our insight is that interposing a modest amount of functionality between language transitions suffices to reuse a substantial amount of functionality of component debuggers, creating one debugger that understands multilingual programs.

3.1. Debugger features

Our goal is to provide all the standard debugging features in a mixed environment. When a user debugs a program, he or she wants to find and correct a defect that results in erroneous data or control flow, which leads to erroneous output or a crash [15]. Rosenberg identifies three essential features in support of this quest [9].

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 `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`.

3.2. Intermediate agent

Our approach to implementing these standard debugger features for a mixed environment is to compose single-environment debuggers through an intermediate agent. Our mixed-environment debugger consists of a controller and one driver for each single-environment component debugger. Figure 2 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:

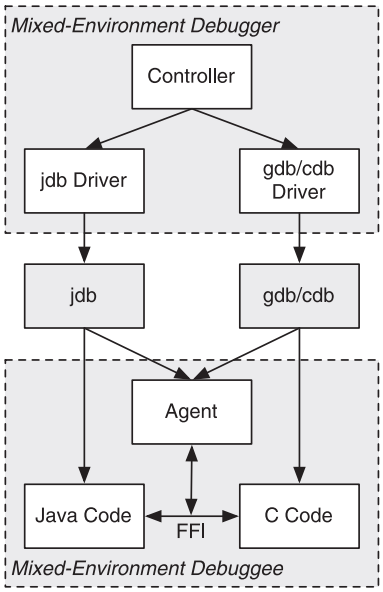

Figure 2. Agent-based debugger composition approach.

DOI: 10.1002/spe

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 as 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.

3.3. 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 stitch together different single-environment stack fragments at their seams.

Therefore, 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. For instance, in the case of JNI, those four kinds of local transitions are Java calls to C, C calls to Java, Java returns to C, and C returns 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 2, 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 FFI is wrapper based and instrument the wrappers.

3.4. 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` (which lists stack frames possibly from multiple languages; see Section 4.2) and `print` require one or more context 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 3 illustrates context switching on the example of switching from `jdb` to `gdb`. Each vertical line represents an execution context, with an 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 and hits a Java breakpoint, thus suspending itself and activating `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

DOI: 10.1002/spe

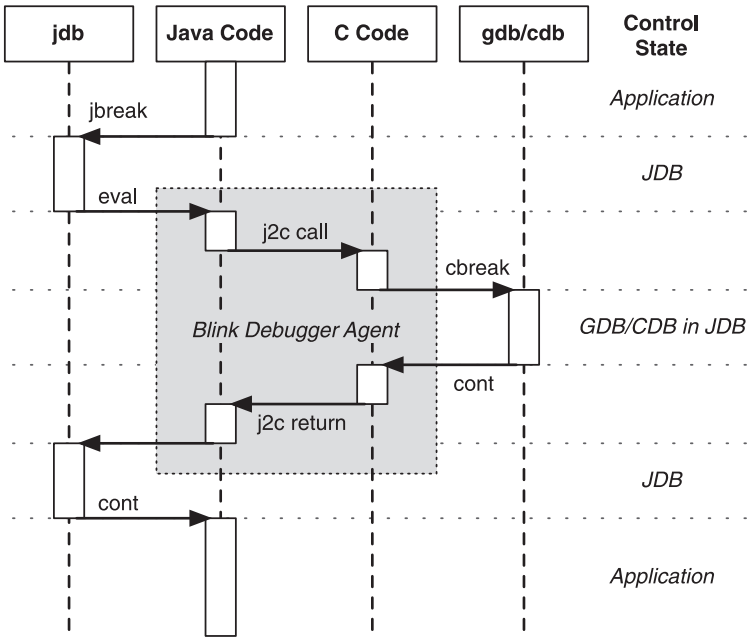

Figure 3. Debugger context switching example, using `j2c` helper function to switch from `jdb` to `gdb`/`cdb`. Blink also has a `c2j` helper function for switching in the other direction.

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`.

3.5. Soft-mode debugging

Debugger composition requires *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 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 debuggee 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. Because 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.

Blink mitigates its use of soft-mode debugging by warning users on actions that might trigger a soft-mode inconsistency. Debugging actions in C are safe as long as the program enters native code through JNI, exceptions are cleared, and garbage collection is enabled. Because 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 because unreliable information is better than no information.

DOI: 10.1002/spe

4. ADVANCED FEATURES

This section shows how interposition for composition facilitates two advanced debugging features: (i) identifying mixed-language interface errors and (ii) mixed-language expression evaluation, which helps users manipulate and examine state across multiple languages.

4.1. Environmental transition checker

Interposition makes it easy to control and compose debuggers, and it also makes it easy to add dynamic checks that find language interface bugs. This section demonstrates how to build a powerful *Environmental Transition Checker* that detects two common language interface bug classes: (i) uncaught exceptions and (ii) unexpected null values. Inspired by these cases, we subsequently built a tool called Jinn in which we codified all JNI rules in state machines, updated on language transitions to dynamically identify many classes of errors [5].

We now motivate our choice of bug classes and describe the extensions to the intermediate agent to dynamically detect missing exception checks and unexpected null values.

4.1.1. Exception checking. The JNI specification forbids JNI calls when an exception is pending [16, Chapter 10.1]. Because C does not support exceptions, users must handle them by hand. In particular, when an exception is raised, the C code must clean up resources such as acquired locks and unwind call frames until it finds an exception handler or exit. C macros and nested function calls complicate the task of writing C code that unwinds the stack and releases resources. Furthermore, because exceptions are rare, this code is hard to exercise and test, which leads to bugs. Previous work shows that programmers tend to write JNI code that incorrectly propagates exceptions [3, 4]. We thus add code to Blink that automatically detects missing *error checking*, which is key to integrating languages with and without automatic exception handling.

To detect missing error checking, Blink adds the checks to the intermediate agent, which instruments and interposes on all JNI function calls. For example, Blink wraps `CallStaticIntMethod` as follows:

```
int wrapped_CallStaticIntMethod(JNIEnv* env, ...) {
  if (jvm_ExceptionCheck(env))
     cbreak(env, "Missing JNI Error Check!");
  return jvm_CallStaticIntMethod(env, ...);
}
```

The agent changes the pointer `CallStaticIntMethod` to refer to `wrapped_CallStaticIntMethod` instead of the original `jvm_CallStaticIntMethod`. The wrapper checks if the JVM has a pending exception. If it does, it executes `cbreak`, a native breakpoint set during agent initialization, which reports a breakpoint hit to the native component debugger and, in turn, to Blink, which displays the error message to the user together with the current calling context.

4.1.2. Null checking. The JNI specification requires that some function arguments must be non-null pointer values [16, Chapter 10.2]. If JNI functions receive unexpected arguments, the JVM may crash or silently produce incorrect results. Neither outcome is desirable, and the programmer should inspect and correct all these errors. Blink dynamically detects obviously invalid arguments to JNI functions, that is, `NULL` or `(jobject)0xFFFFFFFF`. We extend Blink's intermediate agent interposition on every JNI function call to check that the arguments are valid as in the following example function:

```
jstring wrapped_NewStringUTF(JNIEnv* env, char* utf) {
  if ( (utf == NULL) || (utf == 0xFFFFFFFF) )
     cbreak(env, "Invalid JNI Argument!");
  return jvm_NewStringUTF(env, utf);
}
```


DOI: 10.1002/spe

So, when C passes `NULL` as the `utf` argument, the agent calls the C breakpoint function `cbreak` and reports an error message and the current stack. At this point, the user probably needs to examine variables and expressions from both languages to determine the root cause of the invalid argument. We therefore provide mixed-language expression evaluation, as described in the next section.

4.2. Jeannie mixed-environment expressions

The more powerful a debugger's data inspection features, the easier it is for the user to determine whether he or she is on the right track to finding a bug. For example, `gdb` provides expression evaluation with an 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 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-environment expressions, which nest subexpressions from multiple languages and environments with a language specification operator. 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-environment expressions written in the Jeannie programming language syntax [10], which mixes Java and C code using the incantation 'backtick period language', that is, `` `.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 [17]:

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

The C function implementing this Java method is 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. But the mixed-environment expression `` `.Java((`.C 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.

Blink decomposes a Jeannie expression into language-specific subexpressions and delegates the tasks of evaluating each subexpression to language-specific component debuggers. The delegation task is based on two primitive commands parameterized by languages: `backtick` and `print`.

backtick. `backtick` implements the semantics of the backtick operator in Jeannie in the following form:

`backtick`(*source*, *target*) *expression*

source and *target* are programming languages, and *expression* is an expression in the *source* language. `backtick` evaluates *expression*, transfers the result value from the *source* language to the *target* language, and returns the name of a fresh variable in the *target* language holding the transferred value.

print. `print` generalizes expression evaluation in component debuggers by taking a language as a parameter as well as an expression in the following form:

`print`(*lang*) *expression*

DOI: 10.1002/spe

lang is a programming language, and *expression* is an expression in the *lang* language.

After Blink decomposes a Jeannie expression into single-language subexpressions, it executes a sequence of the two primitive commands. For instance, `'.Java((`.C arr).length)` is decomposed into the following three primitive commands.

```
backtick(c, java)  arr            => _v0
backtick(java, c)  _v0.length     => _v1
print(c)           _v1
```

The first `backtick` command evaluates the `arr` expression in C holding an opaque reference to a Java array, transfers the opaque reference to a Java reference, and returns a fresh variable in Java holding the Java reference. The second `backtick` command evaluates the Java expression containing the fresh variable from the first command, obtains the length of the Java array, transfers the integer value from Java to C, and returns a fresh C variable that holds the C integer value. The last `print` command evaluates the C variable `_v1`, thus printing the length of the array.

4.2.1. Implementing the backtick command. To convert and store values across languages in a backtick command, we add two features: mixed-environment data transfer and convenience variables.

Mixed-environment data transfer. Mixed-environment data transfer is constrained by the kinds and mappings of data types and values offered by the FFI. For instance, the set of exchangeable types in JNI includes a few Java primitive types (e.g., `jint`) and dozens of opaque reference types (e.g., `jobject`). The mechanisms of transferring data across languages are baked in the implementations of FFIs. For instance, a call from Java to C converts a Java reference into an opaque reference, whereas a call from C to Java converts an opaque reference to a Java reference.

To transfer data values across languages in general, Blink delegates the value conversion task to the intermediate agent executing calls between Java and C inside the debuggee process. For instance, Blink converts an opaque reference to a Java reference by generating and evaluating a C expression that contains the JNI reference as a parameter to a call to the agent's function. The agent transfers the opaque reference to a Java reference and stores the result in a Java global variable. Blink next generates and evaluates a Java expression to extract the Java reference in the global variable.

One complication in this meta-programming approach of generating and evaluating expressions is the static typing in Java and C that expects all generated expressions to be correctly typed. In the case of C values, `gdb` provides exactly what Blink needs: the `whatis` command finds the type of an expression without executing it. For instance, consider a backtick command `backtick(c, java) p` where `p` is an opaque reference. The `whatis` command discovers that the type of `p` is `jobject`, and Blink chooses the agent function `c2java_jobject`, which transfers an opaque reference in C into the reference in Java. Blink then generates and evaluates `c2java_jobject(p)` as a correctly typed C expression by construction. If the type of `p` is `jint`, Blink would choose the agent function `c2java_jint` that expects an argument of the `jint` type. All values of opaque reference types (e.g., `jstring` and `jarray`) go through `c2java_jobject`. The eight primitives types (e.g., `jboolean` and `jint`) have their specialized agent functions (e.g., `c2java_jboolean` and `c2java_jint`). These agent functions share the `c2java` prefix and return the location of a Java global variable of type `Object` containing the transferred value. Primitive values are wrapped into Java objects (e.g., `Integer`). To extract the value in a variable `o`, Blink adds either type casts or method calls to the variable. For instance, `((Integer)o).intValue()` is the expression for extracting the integer value stored in the wrapper object.

In case of Java values, `jdb` lacks the necessary functionality, and our workaround takes advantage of method overloading in Java and `jdb`. The Blink agent declares several overloaded `java2c` methods in Java for the argument types of `Object` and the eight primitive types. `jdb` chooses one of the overloaded methods by matching the types of formal and actual arguments. For instance, `jdb`

DOI: 10.1002/spe

chooses the overloaded method of receiving a Java reference if the Java expression has reference type instead of other overloaded methods receiving a value of primitive types.

Convenience variables. Convenience variables store the results of a (sub)expression evaluation in temporary 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. Because `gdb` provides convenience variables (written '`$var`'), Blink reuses them to store C values. Because `jdb` and `cdb` lack this feature, Blink implements convenience variables in the debugger agent, using a table to map names to values and languages.

4.2.2. Read–eval–print loop. This section explains how Blink evaluates expressions.

Read. As suggested by Rosenberg [9], the Read stage of Blink's REPL reuses syntax analysis code. We reuse the Jeannie grammar, which composes Java and C grammars [10, 18]. 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. Blink uses abstract syntax tree (AST) implementations from the xtc compiler framework, which is integrated with *Rats!* and provides generic tree walking support.

Eval. The Eval stage of Blink's REPL evaluates the AST in two passes. Both passes use depth-first left-to-right tree traversals. The first pass annotates each AST node with its language (Java or C). Figure 4 shows how Blink annotates the AST for the expression '`x = $y + ` z`', assuming that the current language is Java.

The second pass does the actual evaluation. The evaluation pass uses the `backtick` and `print` commands discussed earlier for language transitions and for the root of the AST, respectively. That leaves only AST nodes for operators in the languages being debugged. Rather than eagerly evaluating such nodes one by one, the evaluation pass builds up expression strings corresponding to maximum single-language subtrees. Evaluation of those expression strings is delayed as far as possible and is only forced at AST nodes for backtick or print. Figure 5 illustrates this. For example, the evaluator delimits single-language subtrees in Java and C at `backtick` and creates a convenience variable `_vj` as a representative of the subtree below `backtick`. Rather than eagerly computing the result of `$y + _vj` at the + node, the evaluator merely computes an expression string at

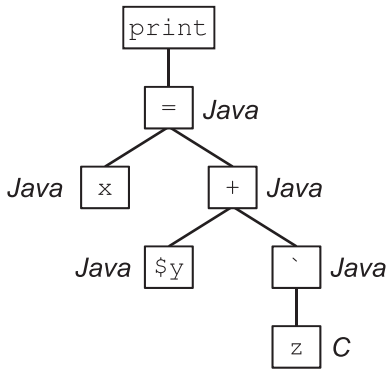

Figure 4. Reading and annotating the expression `x = $y + ` z` when the current language is Java.

DOI: 10.1002/spe

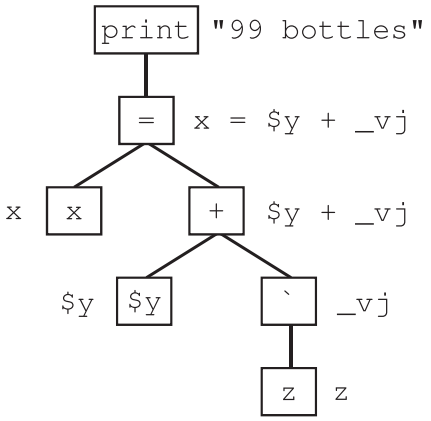

Figure 5. Evaluating the expression `x = $y + ‘z` when the current language is Java, and the current values of `$y` and `z` are the integer `99` and a C reference to the Java string `" bottles"`, respectively.

```
1.  interpret(astNode):
2.    lang ← astNode.language
3.    if astNode.operator == BACKTICK:
4.      source ← astNode.children[0].language
5.      subExpression ← interpret(astNode.children[0])
6.      tmpVarName ← backtick(source, lang) subExpression
7.      return tmpVarName
8.    else if astNode.operator == PRINT:
9.      subExpression ← interpret(astNode.children[0])
10.     print(lang) subExpression
11.     return null
12.   else: //single-language operator of one of the concrete languages
13.     for each i in range(0, astNode.children.length):
14.       subExpressions[i] ← interpret(astNode.children[i])
15.     op ← astNode.operator
16.     expression ← generate(lang, op, subExpressions)
17.     return expression
```

Figure 6. Pseudo-code for the Eval stage of Blink’s REPL.

that point. That string is further incorporated into the generated string for the parent `=` node and ultimately evaluated as part of the `print` node.

Figure 6 shows the algorithm for the expression evaluation pass. The `interpret` function visits an AST node and returns a subexpression to be incorporated by the evaluation of the parent node. Lines 3–7 handle backtick nodes in the AST by invoking the `backtick` function described earlier. The `backtick` function evaluates an expression in the source language, transfers the result to the target language, stores it in an internal convenience variable, and returns the name of that temporary. Lines 8–11 handle print nodes by invoking the `print` function described earlier to display a result to the user. Because print nodes are always at the root of the AST, this case returns **`null`**. Finally, lines 12–17 handle all other AST nodes, which correspond to normal C or Java operators or user-visible debugger convenience variables. For such nodes, the interpreter first computes expression strings of children (e.g., `$y` and `_vj` in Figure 5) and then combines them with the appropriate operator syntax (e.g., `+` is an infix operator, so the combined expression is `$y + _vj`).

In our conference paper [12], Blink used to take a ‘one node at a time’ approach for expression evaluation. This choice kept the engineering effort reasonable and increased our confidence that Blink was accurate. However, it lead to a large number of calls to component debuggers. More importantly, Blink had to special-case l-values by delaying their evaluation and building up larger expressions. We subsequently changed Blink, so it delays all single-language evaluation, not just that of l-values. This choice simplifies the overall design and speeds up the interpreter, because it makes fewer round trips to component debuggers.

DOI: 10.1002/spe

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 [9]. But 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 subresults and re-enables all user breakpoints.

5. BLINK IMPLEMENTATION

While previous sections described debugger composition and the advanced features it enables at a high level, this section explains Blink's implementation in detail.

5.1. Controlling component debuggers

In order to control component debuggers, Blink employs event-driven programming. The event sources include the drivers for each component debugger, and the event sink is the controller. This event-driven programming framework decouples the controller from the component debuggers as well as allowing Blink to issue appropriate actions based on the sequence of events from the component debuggers. The drivers communicate with the component debuggers through Posix pipes by sending and receiving textual messages. Each driver recognizes some patterns of the textual output when it sends a command, waits for the completion of the command, and generates an event containing the result. This choice of the textual interface is driven by the command line interface available in the single-environment debuggers such as `jdb`, `gdb`, and `cdb`, but our event-driven framework could also accommodate any kinds of interface to component debuggers.

5.2. 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) [19]. 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 3. The helper functions `j2c` and `c2j` are part of the Blink debugger agent and contain breakpoints set during initialization. These internal breakpoints force the application to surrender control to the respective debugger.

Runtime transition interposition. The Blink agent interposes on all environment transitions to report full mixed runtime stack traces and to control single stepping between environments. Figure 7 shows the four possible transitions between Java and C. (Exceptions do not add additional cases, because at the boundary between environments, they are indistinguishable from returns.) The program in Figure 8 contains mutual recursion between Java and C to exercise the four kinds of transitions. `jPing` in Java and `PingPong_cPong` in C call each other until the integer argument reaches zero.

`j2c` call: Line 8 in Figure 8 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 8 conceptually reads as follows:

DOI: 10.1002/spe

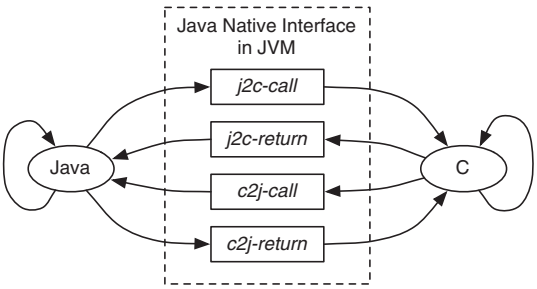

Figure 7. Transitions between Java and C.

PingPong.java

```
 1. class PingPong {
 2.   static { System.loadLibrary("PingPong"); }
 3.   public static void main(String[] args) {
 4.     jPing(3);
 5.   }
 6.   static int jPing(int i) {
 7.     if (i > 0)
 8.       cPong(i - 1);
 9.     return i;
10.   }
11.   static native int cPong(int i);
12. }
```

PingPong.c

```
13. #include <jni.h>
14. jint PingPong_cPong(
15.   JNIEnv* env, jclass cls, jint i
16. ) {
17.   if (i > 0) {
18.     jmethodID mid = (*env)->GetStaticMethodID(
          env, cls, "jPing", "(I)I");
19.     (*env)->CallStaticIntMethod(env,cls,mid,i-1);
20.   }
21.   return i;
22. }
```

Figure 8. JNI mutual recursion example.

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

Because Blink cannot know all native methods at start-up, and part of the wrapper is specific to the C function being wrapped, Blink needs to generate code on the fly. On the other hand, most of the wrapper is general, just passing arguments to and results from the original native method implementation while also invoking the debugger agent. Therefore, instead of generating code for the entire wrapper, Blink just generates a small trampoline in assembly code. For instance, consider the assembly code template for IA32 in AT&T syntax:

```
movl $descriptor, %eax
jmp  j2c_wrapper
```


DOI: 10.1002/spe

`$descriptor` is a parameter that identifies a native method. The agent dynamically generates two instructions for the native method, saves the address of the native method, and uses JVMTI's `NativeMethodBind` to replace the address of the native method with the address of the generated code. When the host JVM activates the native method, the first instruction stores the identity of the native method into the scratch register `%eax`, and the second instruction jumps to the generic wrapper `j2c_wrapper`. The generic wrapper receives the identity of the native method through the scratch register, extracts the arguments, and calls the original native method.

This approach is simple and general; that is, it 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 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 earlier. The return looks just like an ordinary function return, and, in fact, the same C function can return to Java or to C.

`c2j` call: All calls from C to Java go through a JNI interface function, such as `CallStaticIntMethod` in Figure 8 on line 19. 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 8. During JVMTI initialization, Blink replaces the original function pointers by pointers to wrappers. Conceptually, the wrapper for `CallStaticIntMethod` reads as follows:

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

Note that, for demonstration purposes, Section 4.1.1 showed a different wrapper for `CallStaticIntMethod`. In the actual implementation, the wrapper also performs the check for pending exceptions.

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

5.3. Context management

One basic debugger principle from Rosenberg's book [9] 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 8 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 → jPing(3):7
```

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

DOI: 10.1002/spe

`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, and 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, the single-environment debuggers only know about a part of the stack each, because each environment uses its own calling conventions. 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 5.2. 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 later, the user can also inspect data from both languages at a breakpoint.

5.4. Execution control

With context as the torch, 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.[‡] After it initializes

[‡]The internal breakpoints are multiplexed for several conditions. See Section 7.3 for the performance impact of evaluating these conditions.

DOI: 10.1002/spe

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 states 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 through the program, 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, because accurate Java single stepping requires interpreting the semantics of all bytecodes and accurate C single stepping requires platform-dependent disassembly. Blink therefore relies on the component debuggers for this functionality.

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.

1. The component debugger's single step remains in the same environment. Blink performs no further action.
2. There is an environment transition and consequently an internal breakpoint intercepts it. Blink steps from the internal breakpoint to the next line.
3. An exceptional condition, such as a segmentation fault, occurs. Blink abandons single stepping.

In all cases, Blink then disables its internal breakpoints, as usual for breakpoint algorithms [9].

5.5. 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. However, if the variable is declared in a different environment, the user would add a backtick to the variable so that Blink uses the other component debugger.

DOI: 10.1002/spe

6. GENERALIZATION

The previous sections focus on composing debuggers for Java and C. This section discusses how to generalize our approach to more environments, starting with the three requirements for composing debuggers.

Single-environment debuggers: As should be expected, debugger composition requires single-environment debuggers to compose. The single-language debuggers must support execution control, context management, and data inspection (as discussed in Section 3.1). The controller can extract these features through a command line interface (which is what we use), an API, or a wire protocol.

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, for 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.

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. On the other hand, if the single-environment debugger does not support direct function invocation, such as in Caml, we must call the helper function through other means, for example, using an agent helper thread, adding functionality, or using a lower-level API underlying the single-environment debuggers.

Composing environments. Most language implementations interoperate with C and implement interoperating with other languages using C. Given two environments where one environment is the native C environment, it is easy to satisfy the aforementioned criteria. For instance, Perl, Python, and Ruby have debuggers and FFIs to C. We can thus satisfy the three requirements as follows: (i) reuse the `perldebug`, `pdb`, or `ruby-debug` single-environment debuggers and their interfaces; (ii) extend the runtime systems to interpose calls to native methods; and (iii) use `perldebug`, `pdb`, or `ruby-debug` to evaluate calls to native methods that trigger a C breakpoint. We implement all of these as a proof of concept, as described later.

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, because most environments implement FFIs to C. Using C as a bridge environment, all the essential requirements are satisfiable: (i) N single-environment debuggers handle their corresponding N environments; (ii) interposition captures transitions between the N environments and C, because every transition goes through C; and (iii) debugger context switching to any environment also goes through the bridging C environment.

Feasibility of composition. We conducted an experimental study to investigate how our approach generalizes beyond JNI. Table I summarizes the results of this feasibility study of our debugger composition. We implement prototype language interposition for five of the six languages in the table. These languages offer a variety of FFIs, execution environments, and single-language debuggers.

For each language, we spend one day to build a prototype intermediate agent in dozens of lines of code (LOC). We perform a simple test of executing a debugee program written in Common Lisp, C#, Perl 5, Python, and Ruby languages. The programs transition between the original language and C. We attach both `gdb` and the single-language debugger to the debuggee program, load the intermediate agent, and trigger debugger context switching at both C and the other language. Successful debugger context switching indicates very strong evidence that are our debugger composition approach will be successful in this language settings. Five of the six prototypes successfully per-

DOI: 10.1002/spe

Table I. Feasibility of debugger composition using the lines of code (LOC) in the prototype intermediate agents over additional programming languages, foreign function interfaces, and single-language debuggers.

Language	Foreign function interface	Debugger	Feasibility	LOC
Caml	OCaml/C	`ocamldebug`	Weak	
Common Lisp	CLISP foreign function call facility	top level loop	Strong	21
C#	P/Invoke	`sdb`	Strong	49
Perl 5	XS	`perldebug`	Strong	48
Python	Python/C	`pdb`	Strong	58
Ruby	Ruby C extension	`ruby-debug`	Strong	27

form the context switches between languages. Caml's debugger (`ocamldebug`) does not satisfy the requirement of evaluating expressions containing function applications, and we could not find any workaround without adding this functionality to the debugger. For the five other debuggers for Common LISP, C#, Perl 5, Python, and Ruby, we trigger debugger context switching successfully.

Given that a significant fraction of single-language debuggers evaluate expressions containing calls to the helper functions in agents or the programming execution environment uses interpretation, we conclude that the requirements for debugger composition are very often met in practice.

7. 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.

7.1. Methodology

We rely on single-environment debuggers, JVMs, C compilers, and operating systems. We use JDK 1.7 as implemented by Oracle and IBM. For the debuggee running on Linux/IA32 machines, we use Oracle Hotspot Client 1.7.0_25 [20] and IBM's J9 1.7.0 SR6 [21]. `jdb` 1.6 and `gdb` 7.4 inspect code and data produced by Oracle's `javac` 1.7.0_25 and `gcc` 4.6.3 with the `-g` option. For Windows, we use Oracle's Hotspot Client 1.7.0_45, Oracle's JDK 1.7 `jdb`, Microsoft's C/C++ compiler (`cl.exe`) 16.00.30319.01, and Microsoft's `cdb` 6.12.0002.633 debugger. For MinGW, an open-source development environment for native Microsoft Windows applications, we use `gcc` 4.8.1 and `gdb` 7.6.1.

7.2. 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.

7.2.1. Construction effort. Table II shows the code sizes of Blink, `jdb`, `gdb`, and their components. The line counts for `jdb`, JDI, JDWP, and JVMTI are for the source lines of OpenJDK 1.7.0_25. The JDI line counts are for the JDI implementation under `jdk/src/share/classes/com/sun/tools/jdi` and three others. JDWP line counts are for the source files under `jdk/src/share/back` that are compiled into the JDWP agent. The JVMTI line counts are from the source files under `hotspot/src/share/vm/prims`. Blink adds a modest 10,295 SLOC to integrate 1,478,818 SLOC from the Java and C debuggers. The SLOC of the existing debugger

DOI: 10.1002/spe

Table II. Debugger source lines of code (SLOC).

Debugger		SLOC	Files
Blink		10,295	45
	Controller (front-end)	4,924	18
	jdb driver (back-end)	384	1
	gdb driver (back-end)	514	1
	cdb driver (back-end)	533	1
	Agent - Java (back-end)	1,717	11
	Agent - C (back-end)	2,223	13
Java debugger – `jdb` 1.6		99,016	473
	jdb (user-interface)	19,314	119
	JDI (front-end)	13,617	208
	JDWP Agent (back-end)	16,668	79
	JVMTI (back-end)	49,417	67
C debugger – `gdb` 7.4		1,379,802	3,817
	gdb	599,400	2,878
	include	41,271	284
	bfd	353,887	430
	opcodes	385,244	225

packages are 9–134 times larger than Blink's. Although other researchers show how to build single-environment debuggers more economically than `gdb` [22, 23], 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 and code patching for breakpoints.

7.2.2. Portability. To evaluate the effort required for porting Blink to multiple platforms, we measure the amount of platform-independent and platform-dependent code and examine how it relies on architectures and operating systems.

The basic composition framework requires 4,924 SLOC. Blink needs an additional 4,838 SLOC in the agent, `jdb` driver, and `gdb` driver to support our initial configuration, which uses Oracle Hotspot JVM, `jdb`, and `gdb` running on Linux/IA32. Out of Blink's total 10,295 SLOC, approximately 1,500 SLOCs implement platform-specific code in the agent and debugger drivers, representing about 15% of Blink's code base. Our native agent contains a small amount of non-portable platform-specific 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.

Dozens of source lines in the agent rely on architecture and OS. The architecture-dependent code is composed of an assembly template to wrap native methods and some inline assembly code to stitch frames of native methods and JNI functions. The OS-dependent code abstracts process identifiers and synchronization primitives. Specifically, the agent obtains the process identifier of a debugee to attach itself to a native debugger and uses mutexes to protect some data structures keeping track of the set of wrapped native methods and allocating memory for the generated machine code from an assembly code template.

7.2.3. 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.

DOI: 10.1002/spe

Context management. This test sets two breakpoints, at `jPing` (`PingPong.java:7`) and `cPong` (`PingPong.c:17`) in Figure 8. During execution, the application hits each of these breakpoints twice and issues the `backtrace` command each time.

Execution control. This test first sets a breakpoint at the `main` method of the mutual recursion example in Figure 8. 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 [11].) 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. All these and other functionality tests succeed for the following configurations on IA32:

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

The 'MinGW' case uses Windows with the GNU C compiler, instead of Microsoft's C compiler. We performed cursory testing of Blink with Microsoft's C compiler and Microsoft's C debugger on all the key debugger functionality:

$$\text{Oracle JVM} + \text{Windows} + \texttt{cdb}$$

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`.

7.3. Time and space overheads

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, because 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 compared the execution times with and without the agent. We measured runtime with Oracle Hotspot 1.7.0_25 running on a Linux/IA32 machine on the SPECjvm98 and DaCapo Java v.2006-10 Benchmarks [24, 25]. These Java benchmarks exercise C code inside the standard Java library. The initial heap size was 512 MB, and the maximum heap size was 1 GB. The experiments used an Intel Xeon E5-2665 2.4 GHz running Linux 3.2.0-48. Each benchmark iterated once. We took the median of 16 trials and normalized the execution time with the agent by the time without the agent. On average, Blink's total overhead is 2%.

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 about 286 KB for machine code and data. Additionally, each native method incurs 52+ bytes space overhead for its wrapper, instantiated from an assembly code template and descriptor containing the type signature of the native method. Finally, each thread requires 40+ bytes of thread-local storage used by the intermediate agent and 32+ bytes for each wrapper activation on the stack for an environment transition. We do not measure total space overhead in a live system, because it is small by design.

DOI: 10.1002/spe

8. DEBUGGING MIXED-ENVIRONMENT PROGRAMS

This section presents our experience of debugging real-world mixed-environment programs with Blink. We classify mixed-environment bugs into two classes: (i) bugs with cause–effect chain across execution environments and (ii) language interface bugs.

8.1. Bugs with a cause–effect chain across execution environments

Bug 322222 in the Eclipse Bugzilla database illustrates a heroic debugging effort for a critical bug. The bug crashed JVMs frequently enough to cause more than 14 duplicate bug reports: 261627, 283024, 285749, 291128, 299732, 300637, 303389, 316527, 318623, 319609, 320590, 321929, 323107, and 325238. It had survived more than a year from 2009 to 2010 with more than 100 comments from dozens of programmers before the patch went into Eclipse 3.6.1 in September 2010. Debugging was painful because the erroneous execution included two events in different environments: an exception in Java and a segmentation fault in C. Figure 9 presents a program in Java derived from the test case in Bug 323107 to show that a Java exception triggers a segmentation fault in C. The only difference between `SWTBug322222.success` and `SWTBug322222.fail` is the call-back listener that throws a Java exception. `SWTBug322222.fail` creates a table widget object, sets the number of entries, and computes the size of the widget object. This process contains extensive cooperative calls between Java and native code. The native code determines how a table entry will be graphically shown, and Java code determines what data will be displayed for the table entry.

Figure 10 simplifies the activation tree from `SWTBug322222.fail` where the control flow goes across Java and C. On a successful run, it would have created a table data entry for a table widget in the call to `gtk_tree_view_column_cell_set_cell_data` and estimated the size of the table widget in the call to `gtk_tree_view_column_cell_get_size`. The program fails to create the table data entry because `SWTBug322222.handleEvent` throws a Java exception. The JVM discards several Java stack frames, and `CallIntMethodV` returns to `callback` with a pending Java exception in the current thread. The JNI requires the native code not to execute any JNI functions except for several ones that check a pending Java exception and clean up resources. `callback` follows this exception state rule in the second activation by querying the exception state with `ExceptionOccured` and deleting the two pointers to Java objects with `DeleteLocalRef`. Unfortunately, `pango_layout_new` is oblivious of the pending exception,

```
import org.eclipse.swt.*; import org.eclipse.swt.widgets.*;

public class SWTBug322222 {
  static void main(String[] args) {
    success();
    fail();
  }
  static void success() {
    final Shell shell = new Shell(Display.getDefault());
    final Table tableSuccess = new Table(shell, SWT.VIRTUAL);
    tableSuccess.setItemCount(100);
    tableSuccess.computeSize(SWT.DEFAULT, SWT.DEFAULT);
  }
  static void fail() {
    final Shell shell = new Shell(Display.getDefault());
    final Table tableFail = new Table(shell, SWT.VIRTUAL);
    tableFail.setItemCount(100);
    tableFail.addListener(SWT.SetData, new Listener() {
        public void handleEvent(Event e) {
          throw new RuntimeException("Acceptable");
        }
      });
    tableFail.computeSize(SWT.DEFAULT, SWT.DEFAULT);
  }
}
```

Figure 9. A test case for Bug 322222 in the Eclipse Bugzilla database. `SWTBug322222.success` has no problem, whereas `SWTBug322222.fail` triggers a segmentation fault in the native environment. The difference between the two methods shows that the cause of the fault is the Java exception.

DOI: 10.1002/spe

```
Java SWTBug322222.main
Java   SWTBug322222.fail
Java     org.eclipse.swt.widgets.Control.computeSize
C          Java_gtk_widget_size_request
C            gtk_tree_view_column_cell_set_cell_data
C              callback
JNI              ExceptionOccurred
JNI              CallIntMethodV
Java               org.eclipse.swt.widgets.Display.cellDataProc
Java                 org.eclipse.swt.widgets.Widget.sendEvent
Java                   SWTBug322222.handleEvent
Java                     <<<Cause: an exception in Java>>>
C            gtk_tree_view_column_cell_get_size
C              pango_layout_new
C                g_object_new
C                  callback
JNI                ExceptionOccurred
JNI                DeleteLocalRef
JNI                ExceptionOccurred
JNI                DeleteLocalRef
C              <<<Effect: a segmentation fault in C>>>
```

Figure 10. Simplified activation tree from `SWTBug322222.fail` for the test case in Figure 9. `Java`, `C`, and `JNI` denote Java methods, C functions, and JNI functions, respectively.

and it tries to dereference the return value from `callback`, which is `NULL`. The program ends up with a segmentation fault.

The fix was made to Eclipse 3.6.1, and it replaced the second `callback` with a routine in C that does not call any routine in Java. Specifically, the target of the second `callback` was rewritten from Java to C. From the segmentation fault to the fix, the programmer must discover several facts: (i) the origin of `NULL` in `pango_layout_new`; (ii) characteristics of the pending Java exception; and (iii) the target Java method at the second `callback`.

The origin of `NULL`. Blink runs the program and reports the segmentation fault:

```
blink> run
Signal received: SIGSEGV
```

The programmer would examine the calling context, source lines, and variables by executing a few commands:

```
blink> where
  [1] pango_layout_new  (pango-layout.c:271)
  [2] gtk_tree_view_column_cell_get_size  (gtktreeviewcolumn.c:2646)
  [3] Java_gtk_widget_size_request  (os.c:16216)
  [4] Control.computeSize  (Control.java:467)
  [5] SWTBug322222.fail  (SWTBug322222.java:26)
  [6] SWTBug322222.main  (SWTBug322222.java:7)
blink> list
268
269   layout = g_object_new (PANGO_TYPE_LAYOUT, NULL);
270
271   layout->context = context;
blink> print layout
====> (PangoLayout *) 0x0
```

The statement at line 271 triggers the segmentation fault when it fails to dereference `layout`, which is `NULL`. In order to find the origin of `NULL`, the programmer has no choice but to rerun the program, set a breakpoint at line 269, and single step to `callback` because the call path from `g_object_new` to `callback` goes through indirect calls to the low-level machine code generated on the fly. In other words, the programmer cannot reveal the low-level dynamic control flow by examining the source lines from `g_object_new`. After setting up a breakpoint at line 269, the programmer must select one breakpoint event out of multiple ones because the program calls `pango_layout_new` several times. The programmer can discover the breakpoint by executing `where` to print mixed-environment calling contexts:

DOI: 10.1002/spe

```
blink> break pango-layout.c:269
blink> run
blink> where
 [1] pango_layout_new (pango-layout.c:269)
 [2] gtk_tree_view_column_cell_get_size (gtktreeviewcolumn.c:2646)
 [3] Java_gtk_widget_size_request (os.c:16216)
 [4] Control.computeSize (Control.java:467)
 [5] SWTBug322222.success (SWTBug322222.java:14)
 [6] SWTBug322222.main (SWTBug322222.java:6)
blink> continue
blink> where
 [1] pango_layout_new (pango-layout.c:269)
 [2] gtk_tree_view_column_cell_get_size (gtktreeviewcolumn.c:2646)
 [3] Java_gtk_widget_size_request (os.c:16216)
 [4] Control.computeSize (Control.java:467)
 [5] SWTBug322222.fail (SWTBug322222.java:26)
 [6] SWTBug322222.main (SWTBug322222.java:7)
```

`SWTBug322222.fail` in the second calling context tells the programmer that the program in execution is close to the origin of `NULL` because it appeared in the calling context at the segmentation fault. The programmer would single step through an indirect call to the generated machine code to reach the origin of `NULL` at line 1265 in `callback`:

```
blink> step
...
blink> list
  1162 jintLong callback(int index, ...)
  1163 {
  ...
  1257  done:
  ...
  1259      if ((ex = (*env)->ExceptionOccurred(env))) {
  1260          (*env)->DeleteLocalRef(env, ex);
  ...
=>1265          result = callbackData[index].errorResult;
  ...
  1272       }
  ...
  1285   return result;
  1286 }
```

Characteristics of the pending Java exception. At the origin of `NULL`, the programmer would like to find out what has activated the statement at line 1265. The source lines around line 1265 reveal that a pending Java exception is responsible. The programmer would examine the Java exception. To conclude whether or not the exception is acceptable, the programmer would examine what is in the exception by evaluating a Jeannie expression:

```
blink> print jstr2cstr(`((`((*env)->ExceptionOccurred(env))).getMessage()))
====> "Acceptable"
```

The target Java method at the second `callback`. If the Java exception is feasible, the programmer is interested in finding the target of the Java method from `callback`. The programmer can reverse the control decision at line 1210 in `callback` by clearing the pending exception with a Jeannie expression:

```
blink> list
  1162 jintLong callback(int index, ...)
  1163 {
  ...
=>1210    if ((ex = (*env)->ExceptionOccurred(env))) {
  1211      (*env)->DeleteLocalRef(env, ex);
  1212      goto done;
  1213   }
  ...
blink> print (*env)->ExceptionClear(env)
```


DOI: 10.1002/spe

The programmer would execute the single-stepping command several times and reach the target of `callback`:

```
blink> list
=>1244   result = (*env)->CallIntLongMethodV(env, object, mid, vl);
blink> step
  2957 int pangoLayoutNewProc (...) {
=>2958 ...
blink> where
 [1] org.eclipse.swt.widgets.Display.pangoLayoutNewProc (Display.java:2958)
 [2] CallIntMethodV
 [3] callback (callback.c:1244)
 [4] g_object_new (gobject.c:1542)
 [5] pango_layout_new (pango-layout.c:269)
 [6] gtk_tree_view_column_cell_get_size (gtktreeviewcolumn.c:2646)
 [7] Java_gtk_widget_size_request (os.c:16216)
 [8] SWTBug322222.fail (SWTBug322222.java:26)
 [9] SWTBug322222.main (SWTBug322222.java:7)
```

`org.eclipse.swt.widgets.Display.pangoLayoutNewProc` was the target method in Java, and it was rewritten in C in the bug fix made to Eclipse 3.6.1.

8.2. Language interface bugs

Languages interface bugs happen if a program violates one of the programming rules defined by FFIs such as JNI and Python/C. These voluminous and complex programming rules (e.g., 1500+ rules in JNI [5]) are so hard for programmers that real-world multilingual programs are full of language interface bugs. The effect of an interface bug is not defined, and it crashes a program immediately or insidiously. Our subsequent work on dynamic bug detectors classifies all programming rules into a modest number of classes and detects all interface bugs completely [5]. Blink detects some of these bugs and suspends the program. This section focuses on how Blink helps programmers diagnose the source of language interface bugs.

Table III presents interface bugs along with the program, bug type, and bug site in a few open-source programs written in Java and C. `PrepStmtTest`, `UnitTests`, `SWTExceptionState`, and `UDFTest` are from the source packages. `SWTExceptionState` is an artificially created driver program to activate a bug inside Eclipse SWT 3.6.1. `BadErrorChecking` is an artificially created standalone program to show that an exception state bug can crash JVMs.

Table IV compares Blink to production runs of Hotspot and J9 for each of the bugs in Table III. We use runtime checking in Hotspot and J9 by configuring them with the `-Xcheck:jni` command line option. Blink uses `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 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

Table III. Blink finds JNI bugs.

Main Java class	Program	Bug type	Bug site (source file:line)
`PrepStmtTest`	sqlite-jdbc 3.6.0	Null parameter	NativeDB.c:434
`UnitTests`	Java-gnome 4.0.10	Null parameter	Environment.c:48
`gconf.BasicGConfApp`	libgconf-2.16.2	Null parameter	org_gnu_gconf_ConfClient.c:425
`SWTExceptionState`	Eclipse SWT 3.6.1	Exception state	os_structs.c:1691
`UDFTest`	sqlite-jdbc 3.6.0	Exception state	NativeDB.c:184
`BadErrorChecking`	Blink-testsuite 1.14.3	Exception state	BadErrorChecking.c:21

Blink finds the four JNI bugs in `PrepStmtTest`, `UnitTests`, `gconf.BasicGConfApp`, and `UDFTest`. Blink finds the JNI bug in `SWTExceptionState` when running the test case under Eclipse SWT 3.6.1. `BadErrorChecking` exhibits exception handling bugs reportedly common in both user-level and system-level JNI code [3, 4].

DOI: 10.1002/spe

Table IV. Blink detects JNI bugs and hits a breakpoint whereas other tools often crash or ignore the bugs.

	Production run		Runtime checking (-Xcheck:jni)		Debugging with J9		
					Single environment		Mixed environment
Main Java class	Hotspot	J9	Hotspot	J9	jdb	gdb	Blink
`PrepStmtTest`	Running	Crash	Running	Crash	Crash	Fault	Breakpoint
`UnitTests`	Running	Crash	Warning	Warning	Crash	Fault	Breakpoint
`gconf.BasicGConfApp`	Running	Crash	Running	Crash	Crash	Fault	Breakpoint
`SWTExceptionState`	Running	Running	Warning	Warning	Running	Running	Breakpoint
`UDFTest`	Running	Running	Warning	Warning	Running	Running	Breakpoint
`BadErrorChecking`	Running	Crash	Warning	Error	Crash	Fault	Breakpoint

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.

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 in two categories: (i) null parameters and (ii) exception state checking.

Null parameters. Semantics for JNI functions are undefined when their arguments are `(jobject)0xFFFFFFFF` or `NULL` [16]. Hotspot ignores these errors, and J9 crashes in `gconf.BasicGConfApp` and `UnitTests`, which pass `NULL` to the `NewStringUTF` JNI function (Table IV). `NewStringUTF` takes a C string and creates an equivalent Java string. Returning `NULL` for a `NULL` input may improve reliability but violates the specification of `NewStringUTF`:

> Returns `NULL` if and only if an invocation of this function has thrown an exception. [16]

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` [26]. 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. The 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 `PrepStmtTest`, 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
blink> list
434     return (*env)->NewStringUTF(
435         env, (const char*)sqlite3_column_text(toref(stmt), col));
blink> where
[1] Java_org_sqlite_NativeDB_column_1text (NativeDB.c:434)
[2] org.sqlite.RS.getString (RS.java:241)
[3] org.sqlite.PrepStmtTest.set (PrepStmtTest.java:196)
```


DOI: 10.1002/spe

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 (JNIEnv *env, jobject obj)
18. {
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 [3, 4] 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 runs 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> _
```

9. RELATED WORK

This section describes how our paper advances the state of the art in building mixed-environment debuggers and how Blink compares to prior work.

9.1. Mixed-environment debuggers

One contribution of this paper 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. In retrospect, this idea may seem obvious, but we believe that it was previously unclear whether composition could provide fully featured debugging across language environments.

The closest work to compositional 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 [27, 28]. The resulting system is limited because it cannot examine a mixed stack, cannot step into cross-environment calls, and cannot set breakpoints in one environment when stopped in the other, all of which Blink supports.

There are three mixed-environment debuggers (`dbx`, XDI, and the Visual Studio debugger) that are practical but, unlike this paper, 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 [7]. XDI extends an existing Java debugger for C [6]. Both XDI and `dbx` are powerful, but they are less portable than Blink. XDI works only with the Harmony JVM, which is a non-standard JVM.

DOI: 10.1002/spe

`Dbx` works only with Oracle's JVM on Solaris and on Linux with limited functionality. 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# and C in the CLR [8]. It is also extensible through debug engines [29]. However, in contrast to Blink, where multiple debuggers attach to a single mixed-environment program, each Visual Studio 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.

9.2. Single-environment multilingual debuggers

Some multilingual debuggers require all the languages to implement a single interface in the same environment [23, 30, 31]. For example, the GNU debugger, `gdb`, can debug C together with a subset of Java statically compiled by `gcj` [30]. 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.

9.3. 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' [31], DWARF [32], and even PostScript [22]. 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 [33], `smld` [34], and Hanson's machine-independent debuggers [35] 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. The evaluation of Jeannie mixed-environment expressions in Section 4.2 is platform independent. The intermediate agent has only 10–20 lines of low-level assembly code.

9.4. Advanced mixed-language debugger features

The following subsections discuss work related to Blink's advanced debugger features.

9.4.1. Mixed-language interpreters. One contribution of this paper is Blink's 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 [10]. The Jeannie paper described the language and its compiler but did not describe an interpreter or a debugger.

9.4.2. Mixed-language bug checkers. Another contribution of this paper is Blink's dynamic error checker for JNI calls. The closest related work is the `-Xcheck:jni` flag, which turns on dynamic error checking in Oracle's and IBM's JVMs. Table 3 in [3] summarizes how each JVM behaves for a variety of bugs with and without this flag. For example, the flag traps uses of invalid local references, or double-frees of resources. Blink provides similar functionality in a JVM-independent way and, as an added benefit, provides a stack trace and breakpoint for debugging the problem.

There are various static bug checkers for Java and C. Static analyses are a valuable asset for detecting bugs early. However, they suffer from false positives: not every reported bug is an actual bug. As a dynamic checker, Blink has no false positives. Each existing static JNI bug checker is designed to look only for some class of bugs, and some yield false negatives even for their chosen

DOI: 10.1002/spe

class of bugs. J-Saffire infers and checks Java types from C code [13]. Kondoh and Onodera check type-state properties on JNI code based on BEAM [3]. Tan and Croft use static analyses to study security issues in Java's standard library [4]. Of course, static multilingual bug checkers are not restricted to Java and C. For example, Quail performs string analysis for Java/SQL safety [36]. Static analysis is complementary to dynamic debugging, which helps find some bugs static analysis misses.

An alternative to finding bugs in mixed-language programs is to rewrite those programs in a language that prevents some bugs from occurring in the first place. For example, SWIG [37] generates stubs for C to be called from Tcl. SafeJNI [38] combines Java with CCured [39] instead of C. Jeannie [10] provides a type-checked syntactic embedding for Java and C. While these approaches provide good long-term solutions, they also require substantial code rewrites.

10. CONCLUSIONS

Debugging is difficult and time consuming and requires good tooling support. While there are many interactive debuggers for single-environment programs, there is a lack of debuggers for multi-environment programs. Unfortunately, most programs in modern languages, such as Java, require multiple environments, because they use legacy languages, such as C, for low-level code. This paper presents a language interposition approach and implements it in Blink, a debugger for Java and C. Blink is constructed by composing existing single-environment debuggers for Java and C. We show that interposing a modest amount of functionality between language transitions is sufficient to leverage the execution control, context management, and data inspection of component debuggers, creating one debugger that understands multilingual programs and their environments. We prototype the interposition approach and show it works for five of six commonly used languages. Our compositional approach makes it easier to implement, port, and even add advanced features. This paper describes the design and implementation of Blink, an advanced, fully functional multilingual debugger for Java and C, and some practical experiences using Blink on real-world bugs.

ACKNOWLEDGEMENTS

Much of this work was performed while Lee and McKinley were at the The University of Texas at Austin and supported by the National Science Foundation (SHF-0910818), Samsung Foundation of Culture, Microsoft, and CISCO. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

REFERENCES

1. Furr M, Foster JS. Checking type safety of foreign function calls. *ACM Programming Language Design and Implementation (PLDI)*, Chicago, IL, USA, 2005; 62–72.
2. Furr M, Foster JS. Polymorphic type inference for the JNI. *European Symposium on Programming (ESOP)*, Vienna, Austria, 2006; 309–324.
3. Kondoh G, Onodera T. Finding bugs in Java native interface programs. *ACM International Symposium on Software Testing and Analysis (ISSTA)*, Seattle, WA, USA, 2008; 109–118.
4. Tan G, Croft J. An empirical security study of the native code in the JDK. *Usenix Security Symposium (SS)*, San Jose, CA, USA, 2008; 365–377.
5. Lee B, Wiedermann B, Hirzel M, Grimm R, McKinley KS. Jinn: synthesizing dynamic bug detectors for foreign language interfaces. *ACM Programming Language Design and Implementation (PLDI)*, Toronto, ON, Canada, 2010; 36–49.
6. Providin V, Elford C. Debugging native methods in Java applications. *EclipseCon User Conference*, Santa Clara, CA, USA, March 2007. Available from: http://www.eclipsecon.org/2007/index59af.html?page=sub/&id=4129.
7. Sun Microsystems Inc. Debugging a Java application with dbx, 2007. Available from: http://docs.oracle.com/cd/E19205-01/819-5257 [last accessed 26 May 2014].
8. Stall M. Mike Stall's.NET debugging blog. Available from: http://blogs.msdn.com/jmstall/default.aspx [last accessed 26 May 2014].
9. Rosenberg JB. *How Debuggers Work: Algorithms, Data Structures, and Architecture*. John Wiley & Sons: New York, NY, USA, 1996.
10. Hirzel M, Grimm R. Jeannie: granting Java native interface developers their wishes. *ACM Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Montreal, Quebec, Canada, 2007; 19–38.

DOI: 10.1002/spe

11. Grimm R. xtc — eXTensible C. Available from: http://www.cs.nyu.edu/rgrimm/xtc/ [last accessed 26 May 2014].
12. Lee B, Hirzel M, Grimm R, McKinley KS. Debug all your code: portable mixed-environment debugging. *ACM Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Orlando, FL, USA, 2009; 207–226.
13. Furr M, Foster JS. Checking type safety of foreign function calls. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 2008; **30**(4):18:1–18:63.
14. Tan G, Morrisett G. ILEA: inter-language analysis across Java and C. *ACM Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Montreal, Quebec, Canada, 2007; 39–56.
15. Zeller A. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann: San Francisco, CA, USA, October 2005.
16. Liang S. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley: Boston, MA, USA, 1999.
17. Lind-Nielsen J. BuDDy, July 2004. Available from: http://buddy.sourceforge.net/ [last accessed 26 May 2014].
18. Grimm R. Better extensibility through modular syntax. *ACM Programming Language Design and Implementation (PLDI)*, Ottawa, Ontario, Canada, 2006; 38–51.
19. Oracle Corporation. JVM™ tool interface, version 1.1, 2006. Available from: http://docs.oracle.com/javase/6/docs/platform/jvmti/jvmti.html [last accessed 26 May 2014].
20. Oracle Corporation. Java SE HotSpot at a glance. Available from: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html [last accessed 26 May 2014].
21. Bailey C. Java technology. IBM style: introduction to the IBM developer kit, May 2006. Available from: http://web.archive.org/web/20110214002653/http://www.ibm.com/developerworks/java/library/j-ibmjava1.html [last accessed 26 May 2014].
22. Ramsey N, Hanson DR. A retargetable debugger. *ACM Programming Language Design and Implementation (PLDI)*, San Francisco, CA, USA, 1992; 22–31.
23. Ryu S, Ramsey N. Source-level debugging for multiple languages with modest programming effort. *International Conference on Compiler Construction (CC)*, Edinburgh, UK, 2005; 10–26.
24. Blackburn SM, Garner R, Hoffmann C, Khang AM, McKinley KS, Bentzur R, Diwan A, Feinberg D, Frampton D, Guyer SZ, Hirzel M, Hosking A, Jump M, Lee H, Eliot J, Moss B, Moss B, Phansalkar A, Stefanović D, VanDrunen T, von Dincklage D, Wiedermann B. The DaCapo benchmarks: Java benchmarking development and analysis. *ACM Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Portland, OR, USA, 2006; 169–190.
25. Standard Performance Evaluation Corporation. *SPEC JVM98 Benchmarks*, March 1999. Available from: http://www.spec.org/jvm98 [last accessed 26 May 2014].
26. Sun Microsystems Inc. Bug database Bug 4207056 was opened 1999-01-29. Available from: http://bugs.java.com/ [last accessed 26 May 2014].
27. White M. Debugging integrated Java and C/C++ code, November 2001. Available from: http://web.archive.org/web/20041205063318/www-106.ibm.com/developerworks/java/library/j-jnidebug/ [last accessed 26 May 2014].
28. White M. Integrated Java technology and C debugging using the Eclipse platform. *JavaOne Conference*, San Francisco, CA, USA, 2006.
29. Visual Studio debugger extensibility. Available from: http://msdn.microsoft.com/en-us/library/bb161718(VS.80).aspx [last accessed 26 May 2014].
30. Bothner P. Compiling Java with GCJ, January 2003. Available from: http://www.linuxjournal.com/article/4860 [last accessed 26 May 2014].
31. Linton MA. The evolution of Dbx. *Usenix Technical Conference*, Anaheim, CA, USA, 1990; 211–220.
32. Free Standards Group. DWARF 3 debugging information format, December 2005. Available from: http://www.dwarfstd.org/doc/Dwarf3.pdf [last accessed 26 May 2014].
33. van den Brand M, Cornelissen B, Olivier P, Vinju J. TIDE: a generic debugging framework — tool demonstration. *Electronic Notes in Theoretical Computer Science* 2005; **141**(4):161–165.
34. Tolmach AP, Appel AW. Debugging standard ML without reverse engineering. *ACM LISP and Functional Programming (LFP)*, Nice, France, 1990; 1–12.
35. Hanson DR. A machine-independent debugger—revisited. *Software Practice Experience* 1999; **29**(10):849–862.
36. Tatlock Z, Tucker C, Shuffelton D, Jhala R, Lerner S. Deep typechecking and refactoring. *ACM Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Nashville, TN, USA, 2008; 37–52.
37. Beazley DM. SWIG: an easy to use tool for integrating scripting languages with C and C++. *USENIX Tcl/Tk Workshop (TCLTK)*, Monterey, CA, USA, 1996; 15–15.
38. Tan G, Appel AW, Chakradhar S, Raghunathan A, Ravi S, Wang D. Safe Java native interface. *International Symposium on Secure Software Engineering (ISSSE)*, Washington, DC, USA, 2006; 97–106.
39. Necula GC, McPeak S, Weimer W. CCured: type-safe retrofitting of legacy code. *ACM Principles of Programming Languages (POPL)*, Portland, OR, USA, 2002; 128–139.

DOI: 10.1002/spe