Debug All Your Code: Portable Mixed-Environment Debugging

Byeongcheol Lee
University of Texas at Austin
bclee@cs.utexas.edu

Martin Hirzel
IBM Watson Research Center
hirzel@us.ibm.com

Robert Grimm
New York University
rgrimm@cs.nyu.edu

Kathryn S. McKinley
University of Texas at Austin
mckinley@cs.utexas.edu

Abstract

Programmers build large-scale systems with multiple languages to reuse legacy code and leverage 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 may 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 (1) relatively simple: it requires modest amounts of new code; (2) portable: it supports multiple Java Virtual Machines, C compilers, operating systems, and component debuggers; and (3) powerful: composition eases debugging, while supporting new mixed-language expression evaluation and Java Native Interface (JNI) bug diagnostics. In real-world case studies, we show that language-interface errors require single-environment debuggers to restart execution multiple times, whereas Blink directly diagnoses them with one execution. We also describe extensions for other mixed-environments to show debugger composition will generalize.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Debuggers; D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids

General Terms Languages, Design, Reliability

Keywords Foreign Function Interface, JNI, Composition

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.

1. Introduction

In an ideal world, programmers would write correct programs in a single language. In the real world, time-to-market pressures and evolving software requirements result in mixed-language programs. Software developers resort to multiple languages because (1) they can leverage legacy code and existing libraries, and (2) they can choose 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 [6, 7, 13, 29].

Unfortunately, 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#, Ruby, JavaScript, and Python, which enforce type and memory safety. For portability, these languages rely on virtual machine (VM) execution, using interpretation, just-in-time compilation, and garbage collection, while also hiding internal code, stack, and data representations. Debuggers for managed languages, such as the standard Java debugger `jdb`, operate on VM abstractions, e.g., 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 [18, 27], and the Visual Studio debugger, which supports managed and native code in the Common Language Runtime (CLR) [22]. While these debuggers understand all environments, they are behemoths that are 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 paper 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. 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 [12]. Furthermore, we identify the requirements and mechanisms for generalizing our approach to other language environments. Because Blink reuses existing debuggers, it is simple: Blink requires 9K lines of new code, half of which implement interposition. Blink is portable: it supports multiple Java virtual machines (Sun and IBM), C compilers (GNU and Microsoft), and operating systems (Unix and Windows). By comparison, `dbx` works only with Sun's JVM and XDI works only with the Harmony JVM.

Debugger composition also facilitates powerful new debugging features: (1) a read-eval-print loop (REPL) that, in Blink, evaluates mixed Java and C expressions in the context of a running program, and (2) a dynamic bug checker for two common Java Native Interface (JNI) problems. We demonstrate this functionality using several case studies, which reproduce bugs found in real programs and compare debugging with other tools to debugging with Blink. Whereas the other tools crash, silently ignore errors, or require multiple program invocations to diagnose a bug, Blink typically identifies the bug right away 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:

1. A new approach to building mixed-environment debuggers that composes single-environment debuggers. Prior debuggers either support only a single environment or reimplement 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 [9].
3. Two advanced new debugger features: a mixed-environment interpreter and a dynamic checker for detecting JNI misuse.
4. A description of the requirements and mechanisms for composing language execution environments that lays the groundwork for generalizing debugger composition.

2. Motivation: A Language Interface Bug

This section illustrates 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

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).

common class of JNI bugs that is due to JNI's reflection-like API [8]. 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 Sun'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 [7, 13, 30].

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

```
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. Since `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 either change the string in `EVENT_NAMES[1]` or the method name in Line 15.

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. Section 6 presents requirements and mechanisms for generalizing composition to other mixed-language environments.

3.1 Debugger Features

Our goal is to provide all the standard debugging features in a mixed environment. When a user debugs a program, she wants to find and correct a defect that results in erroneous data or control flow, which leads to erroneous output or a crash [37]. Rosenberg identifies three essential features in support of this quest [20]:

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`.

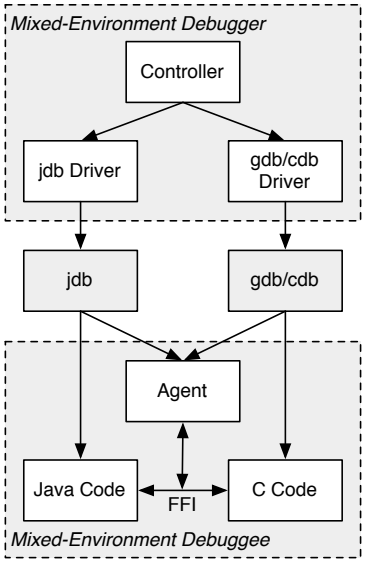

Figure 2. Agent-based debugger composition approach.

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. The 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:

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

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

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

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 track all the 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 (e.g., Java call to C, C call to Java, Java return to C, and C return to Java). The agent must also capture non-local control flow such as exceptions.

Our approach instruments all environment transitions to call agent code. For instance, in Figure 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 foreign function interface (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` 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.

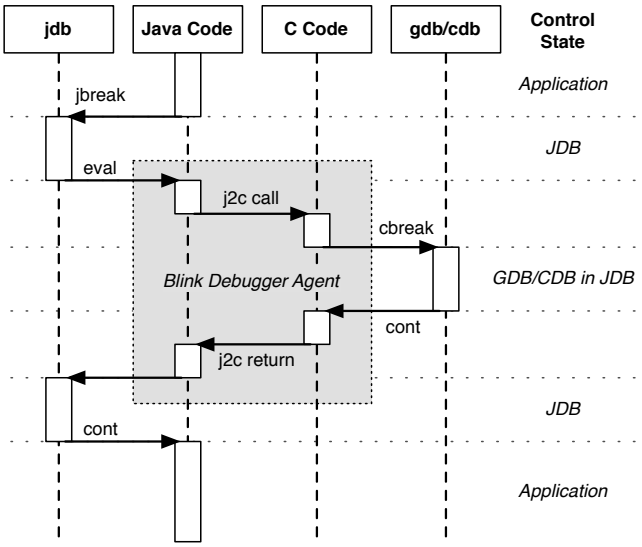

Figure 3. Debugger context switching example, using `j2c` helper function to switch from `jdb` to `gdb`/`cdb`. Blink also has a `c2j` helper function for switching in the other direction.

Figure 3 illustrates context switching on the example of switching from `jdb` to `gdb`. Each vertical line represents an execution context, with the currently active context marked by a box overlaying the line. Horizontal arrows show control transfers between execution contexts. From top to bottom, the application starts out executing Java code 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 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 (Java Debugger Wire Pro-

tocol) 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. Since the JVM is typically not reentrant (i.e., it assumes that no user code runs in the middle of a JVM service), debugger actions may now crash the JVM. For example, the JVM's allocator may temporarily leave a data structure in an inconsistent state, thus making it unsafe for the agent to allocate objects. Furthermore, even if the native breakpoint is not reachable from the JVM, JNI disallows JNI operations when exceptions are pending or garbage collection is disabled. Reentering the JVM without first clearing the exception or re-enabling garbage collection may crash or deadlock the system [13, 14].

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 entered native code through JNI, exceptions are cleared, and garbage collection is enabled. Since we already rely on language interposition, we detect whether the JVM is in a safe state. If the debugger is about to perform an action in C, but the JVM is in an unsafe state, the debugger warns the user. Instead of just warning the user, we could refuse to perform debug actions altogether. We chose a warning over refusal since unreliable information is better than no information.

4. Advanced Features

This section shows how interposition for composition facilitates two advanced debugging features: (1) identifying mixed-language interface errors, and (2) 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: (1) uncaught exceptions, and (2) unexpected null values. We leave to future work the exploration of more powerful bug finders.

Dynamic checking is complementary to static analysis: dynamic checking misses true bugs if code is not executed, whereas static analyses report some false bugs that can never occur. At the same time, dynamic checking has the advantage that it does not need to access the whole program's source code, unlike static analysis [13, 29]. At failure points, our dynamic checker prints the mixed-environment calling context and local variables, which are important clues to finding the root cause.

In the case of Java and C, the boundary is defined by the Java Native Interface (JNI), and we classify bugs as either Java-to-C or C-to-Java. We focus here on C-to-Java bugs. JNI's API is complex and brittle; consequently, it is a major source of bugs [8, 13, 29, 30]. Programmers tend to make mistakes with respect to the Java type system and do not carefully follow the JNI rules [14]. For instance, C code must correctly name fields or methods of Java objects, which cannot be easily checked statically because of dynamic class loading. Programmers must also ensure that the JVM has no pending exception when the program calls a JNI function. The JNI function specification exposes the Java type system and low-level JVM details, such as the exception model and the garbage collector, which many programmers do not understand and have no interest in.

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

4.1.1 Exception Checking

The JNI specification disallows JNI calls when an exception is pending. Since 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, since 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 [13, 29]. 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 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 and previous work reports these errors are common [14]. 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, i.e., `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);
}
```

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 she is on the right track to finding a bug. For example, `gdb` provides expression evaluation with a read-eval-print loop (REPL). An interactive interpreter evaluates arbitrary source language expressions based on the current application state. While implementing a language interpreter requires a significant engineering effort, expression evaluation makes it easier to determine whether 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 [12], which mixes Java and C code using the incantation "backtick period language", i.e., `` `.C `` and `` `.Java ``. A single backtick `` ` `` toggles when there are only two languages, as in Blink. For example, consider this native Java method declaration from the BuDDy binary decision diagram library [15]:

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

The C function implementing this Java method looks as follows:

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

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

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

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

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

4.2.1 Convenience Variables

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

4.2.2 Mixed-Environment Data Transfer

Mixed-environment data transfer is the only case where Blink must discover enough type information to treat the

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

4.2.3 Expression Evaluation (REPL)

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

Read. As suggested by Rosenberg [20], the "read" stage of Blink's REPL reuses syntax and grammar analysis code. We reuse the Jeannie grammar, which composes Java and C grammars [10, 12]. It is written in *Rats!*, a parser generator that uses packrat parsing for expressiveness and performance. The Jeannie grammar and *Rats!* are designed for composition. Section 7 discusses Jeannie in more detail.

Whereas a traditional compiler annotates the AST with types, Blink annotates the AST with: (1) the language (Java or C), and (2) whether each AST node is an r-value (read-only) or an l-value (written-to on the left-hand side of an assignment). Figure 4 shows how Blink annotates the AST for the expression "`x = $y + ‘z`", assuming that the current language is Java. Node `x` is an l-value and node `z` is a C r-value because `z`'s parent is the language toggle backtick `‘`.

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

Eval. The interpreter visits the AST in depth-first left-to-right post-order. Each node is executed exactly once and in the right order, to preserve language semantics in the presence of side effects, and to not surprise users if an exceptional condition, such as a segmentation fault, cuts expression evaluation short.

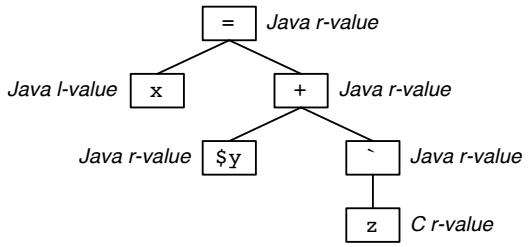

Figure 4. Reading the expression `x = $y + ‘z` when the current language is Java.

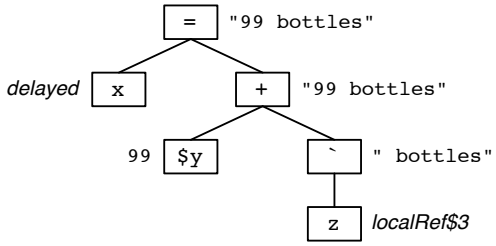

Figure 5. Evaluating the expression `x = $y + ‘z` when the current language is Java.

To evaluate an expression one AST node at a time, Blink uses temporary storage for subexpression results. For r-values, Blink evaluates the node, and then stores the result in an internal convenience variable. For l-values, Blink evaluates their children, but delays their own evaluation. These l-values are evaluated later as part of their parent, which is by definition an assignment. Figure 5 shows the example expression "`x = $y + ‘z`", assuming that the convenience variable `$y` is currently the number `99`, and the C application variable `z` is currently an opaque JNI local reference `localRef$3`. All leaves are variables, which Blink evaluates through the component debuggers' REPL. Blink directly uses any leaf literals without lookup. At inner nodes, Blink needs to perform evaluation actions. For the language toggle operator `‘`, Blink performs a mixed-environment data transfer as described in Section 4.2.2. For Figure 5, Blink discovers that the JNI reference `localRef$3` on the C side refers to the Java string `" bottles"` on the Java side. For other operators, such as `+` and `=`, Blink falls back on the REPL in the component debuggers. Note that in general, an inner node may call a user function and may thus execute arbitrary user code.

Print. When expression evaluation reaches the root of the tree, Blink prints the result. As recommended by Rosenberg, Blink disables user breakpoints for the duration of expression evaluation, because the user would probably be surprised when expression evaluation hits a breakpoint in a callee [20]. 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

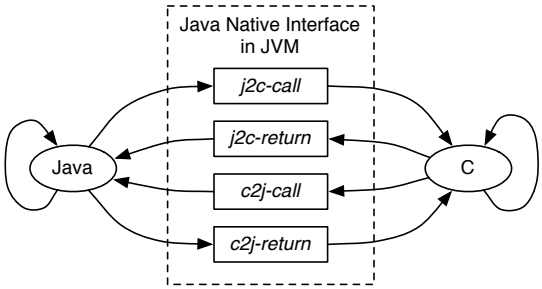

Figure 6. Transitions between Java and C.

the fault point instead. Whether expression evaluation terminates normally or abnormally, Blink always nulls out internal convenience variables for sub-results and re-enables all user breakpoints.

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 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) [26]. 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 hardcoded internal breakpoints. 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 6 shows the four possible transitions between Java and C. Java exceptions are automatically propagated by JNI, and thus do not result in additional environment transitions.

j2c call: Line 8 in Figure 7 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 7 conceptually reads:

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

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

c2j call: All calls from C to Java go through a JNI interface function, such as `CallStaticIntMethod` in Figure 7 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 7. During JVMTI initialization, Blink replaces the original function pointers by pointers to wrappers. Conceptually, the wrapper for `CallStaticIntMethod` reads:

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

```
    c2j_return(); /* interposed c2j return */
    return result;
}
```

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 above.

5.2 Context Management

One basic debugger principle from Rosenberg's book [20] 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 blind-folded 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 provides 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 7 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`

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

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

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

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

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

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

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

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

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

?(Java/C) → `cPong(0):17`

Neither `gdb` nor `cdb` understand 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 5.1. The `j2c` wrapper saves its frame pointer and program counter in a thread local variable, and the `c2j` wrapper retrieves the saved frame pointer and program counter while also overwriting its old frame pointer and return address. Modifying the processor state accordingly guides the C debuggers to skip JVM-specific native frames between j2c and c2j wrappers and yields the following C frames:

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

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

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

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

5.3 Execution Control

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

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

[1] The internal breakpoints are multiplexed for several conditions. See Section 8.3 for the performance impact of evaluating these conditions.

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

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

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

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

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

In all cases, Blink then disables its internal breakpoints, as usual for breakpoint algorithms [20].

5.4 Data Inspection

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

6. Generalization

The previous sections focus on composing debuggers for Java and C. Below, we discuss how to generalize our approach to more environments. Section 7 describes our experience with extending Blink to include the Jeannie programming language, which mixes Java and C in the same methods.

Requirement 1: Single-environment debuggers. As might be expected, debugger composition requires single-environment debuggers to compose. The single-language debuggers must support the features discussed in Section 3.1. The controller can extract these features through a command line interface (which is what we use), an API, or a wire protocol.

Requirement 2: Language transition interposition. Our approach requires instrumenting local and non-local control flow in all directions across environment boundaries. For Blink, we leverage Java's wrapper-based FFI to meet this requirement and instrument the wrappers. However, there are other viable implementation strategies for interposition. For example, 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.

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

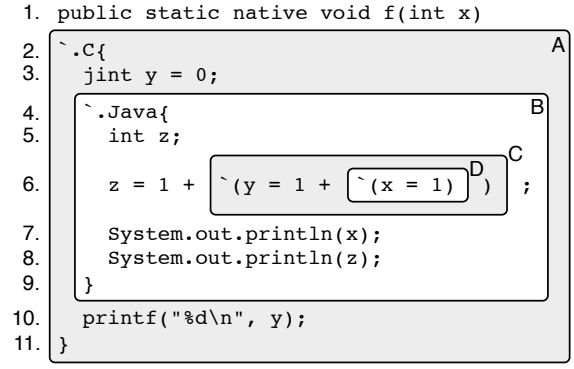

Figure 8. Jeannie line number example.

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

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

7. Language Extension Case Study: Debugging Jeannie

This section shows how composition generalizes Blink to the Jeannie programming language [12]. Jeannie programs combine Java and C syntax in the same source file. This design eliminates many language-interface errors and simplifies resource management and multi-lingual programming. The Jeannie compiler produces C and Java code that executes in a native and JVM environment, respectively. Thus adding Jeannie to Blink serves as an example of debugging more languages in Blink's mixed environment.[2]

[2] Debugging Jeannie is distinct from borrowing Jeannie's expression evaluation functionality, which Blink also does and Section 4.2 described.

Jeannie nests Java and C code in each other in the same file. Compared to JNI, Jeannie is more succinct and less brittle. For example, JNI obscures the Java type system, whereas Jeannie programs directly refer to Java fields and methods, which the Jeannie compiler type checks. In Jeannie, ‘ *.language* specifies the language. As a shortcut, backtick ‘ toggles. For example, in Figure 8, the body of Java method `f` is the block A of C code. Block A contains a nested block B of Java code, with a nested C expression C, which, in turn, nests Java expression D. The Jeannie compiler emits separate Java and C files that implement the expected nesting semantics using JNI. In the example, the Jeannie compiler separates the code for the Java method declaration and snippets B and D into a Java file and puts the code for C snippets A and C into a C file. Jeannie's design supports adding more languages, but that is beyond the scope of this paper.

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

7.1 Context Management

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

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

Calling context backtrace. Since Jeannie is a single language, Blink should show only the user-specified Jeannie methods and functions on the stack, instead of showing the generated single-language functions, which are just an implementation detail. For example, for the C source snippets A and C in Figure 8, the Jeannie compiler generates C functions `f_A` and `f_C`. For the Java snippets B and D, the Jeannie compiler generates Java methods `f_B` and `f_D`. When the application is suspended in D, the low-level call stack is:

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

but this trace is not reflected in the user's code. We changed the Jeannie compiler to generate a table mapping names of generated functions back to the original functions. Blink

uses this mapping to hide low-level call frames and instead reports source-level names, e.g., just "... → `f`".

7.2 Execution Control

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

Breakpoints. To support breakpoints in another language, the debugger needs to map from source-level lines to low-level code offsets. This requires similar debugging tables as for context management (Section 7.1), except in the opposite direction. In the case of Jeannie, there is one additional issue: Blink must delegate the breakpoint to the correct component debugger by using debugger context switching if necessary.

Single stepping. Stepping in Jeannie adds the challenge that a single source line may involve multiple languages. Line 6 in Figure 8 is an example. As discussed in Section 7.1, the Jeannie compiler tracks original line numbers even when code ends up in different source files. Blink implements Jeannie stepping by inspecting line numbers and iterating: it keeps stepping until the source line differs from the starting source line. For step-over, Blink records the current stack depth and then iterates, stepping until stack depth is less than or equal to the initial depth.

7.3 Data Inspection

Data inspection helps users determine if the current state is correct or infected. The compiler for each language must generate a table that maps source-level variable names to underlying variable access expressions in the generated code. The Jeannie compiler stores local variables in explicit environment records [12]. We extended the Jeannie compiler to provide the necessary mapping information through a separate symbol file, which Blink reads on demand.

8. Evaluation

This section evaluates our claim that debugger composition is an economical way to build mixed-environment debuggers and that the resulting debuggers are powerful. We show that Blink is relatively concise, new development cost is low, the space and time overheads are low, and the resulting tool is portable. Through the use of case studies, Section 8.4 demonstrates that Blink helps programmers to quickly find mixed-language interface bugs.

8.1 Methodology

We rely on single-environment debuggers, JVMs, C compilers, and operating systems. We use JDK 1.6 as implemented by Sun and IBM. For the debuggee running on Linux/IA32 machines, we use Sun's Hotspot Client 1.6.0_10 [25] and IBM's J9 1.6.0 (build pxi3260-20071123_01) [1]. We also use Sun's `javac` 1.6.0_10 and `gcc` 4.3.2 with the `-g` option. For Windows, we use Sun's Hotspot Client 1.6.0_10, Sun's `javac` 1.6.0_10, and Microsoft's C/C++ compiler (`cl.exe`) 15.00.21022.08. We use Sun's JDK 1.6.0 `jdb` and Microsoft's `cdb` 6.9.0003.113 debuggers, and GNU `gdb` 6.8 debugger running on Cygwin 1.5.25, a Unix compatibility layer for Windows.

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

Table 1. Debugger SLOC (source lines of code).

8.2 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.

8.2.1 Construction Effort

Table 1 shows the code sizes of Blink, `jdb`, `gdb`, and their components. The `jdb` line counts are for the `jdb` 1.6 sources in `demo/jpda/examples.jar` of Sun's JDK 1.6.0-b105. The JDI line counts are for the JDI implementation in the Eclipse JDT. The JDWP and JVMTI line counts are for the corresponding subdirectories of the Apache DRLVM. Blink adds a modest 9,481 SLOC to integrate 1,103,648 SLOC from the Java and C debuggers. The SLOC of the existing debugger packages are 9 to 107 times larger than Blink's. Although other researchers show how to build single-environment debuggers more economically than `gdb` [19, 21], Blink adds modestly to this effort. Blink only adds new code for interposing on environment transitions and for controlling the individual debuggers. Blink otherwise reuses existing debuggers for intricate platform-dependent features such as instruction decoding for single-stepping or code patching for breakpoints.

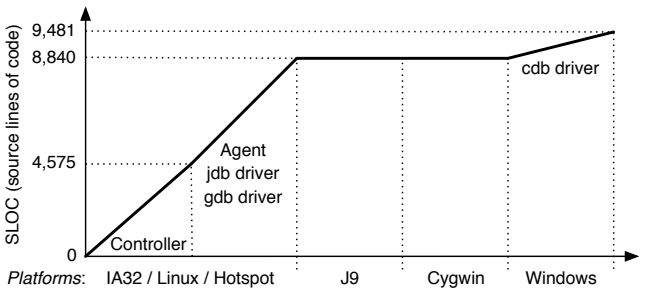

Figure 9. Blink portability and SLOC.

8.2.2 Portability

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

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

Figure 9 plots the cumulative SLOC for the Blink controller; then the code for supporting Hotspot, `jdb`, and `gdb` on Linux; then the code for supporting J9 on Linux; then `gcc` and `gdb` under Cygwin on Windows; and finally Microsoft's C and `cdb` on Windows. As shown in the figure, Blink requires no additional code to support IBM's J9 and Cygwin. Furthermore, it requires only 640 SLOC to support `cdb` on Windows. These results show that Blink's debugger composition is effective and requires only small amounts of code when adding more operating systems, JVMs, C compilers, and component debuggers.

8.2.3 Portability Tests

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

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

Execution control. This test first sets a breakpoint at the `main` method of the mutual recursion example in Figure 7. 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 [9].) When the application hits the breakpoint, the test evaluates a variety of expressions, covering primitive and compound data, pure expressions and assignments, language transitions, and user function calls.

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

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

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

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

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`.

8.3 Time and Space Overhead

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

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

To measure the performance impact of interposition in the intermediate agent, we ran several large Java programs with the Blink agent. We measured runtime and dynamic transition counts with Sun Hotspot 1.6.0_10 running on a Linux/IA32 machine on the SPECjvm98 and DaCapo Java v.2006-10 Benchmarks [23, 3]. These Java benchmarks exercise C code inside the standard Java library. The initial heap size was 512MB, and the maximum heap size was 1GB. The experiments used a Pentium D 2 GHZ running Linux 2.6.27. Each benchmark iterated once. The results are the median of 16 trials.

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

Table 2. Performance characteristics of the Blink debug agent with Hotspot VM 1.6.0_10.

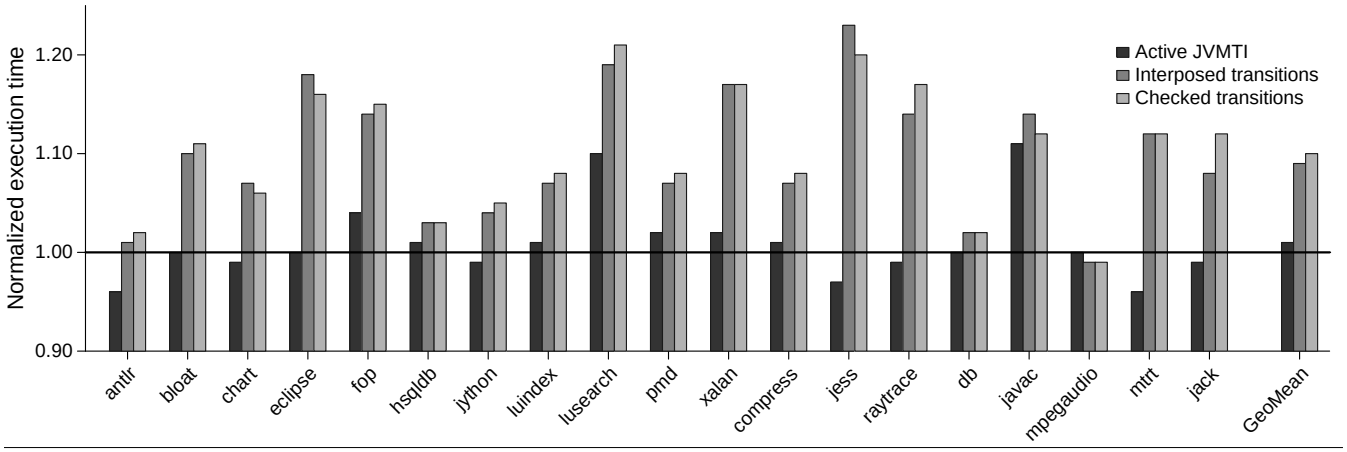

Figure 10. Time overhead of the Blink debug agent with Hotspot VM 1.6.0_10. Note the vertical axis starting at 0.9.

Table 2 shows the results. Column *environmental transition counts* shows the number of dynamic transitions between Java and C. The following columns show execution times in the four configurations—*Base*, *Active JVMTI*, *Interposed transitions*, and *Checked transitions*—and normalized execution times for the debugger configurations. The Base configuration represents production runs without any debugging-related overhead. In contrast, the fully functional agent needs to activate JVMTI, interpose transitions, and check transitions.

Figure 10 illustrates Blink's runtime normalized to the production runs. JVMTI, interposition, and transition checking add 1%, 8%, and 1% overhead, respectively. There are a few counter-intuitive speedups because the JIT and GC add non-determinism to the runtime. On average, Blink's total overhead is 10%. Figure 11 shows that the overhead is sublinear to the total dynamic transition counts. Although the agent overhead is linearly proportional to the dynamic transition counts in theory, it is less in practice because environmental transitions contribute little to overall execution time. For an interactive tool, a 10% overhead is modest.

Space Overhead. The space overhead of running Blink is mostly due to additional code loaded into the debuggee. In particular, on Linux/IA32, the intermediate agent itself requires 388 KB, and the 229 JNI function wrappers introduce 174 KB of constant space overhead. Additionally, each na-

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

Table 3. Studied JNI Bugs. The two JNI bugs in `UnitTest` and `gconf.BasicGConfApp` are found when running these two programs with Blink. `BadErrorChecking` models exception handling bugs reportedly common in both user- and system-level JNI code [13, 29].

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

Table 4. Impact of JNI bugs under different configurations. *Running*: continue executing with undefined state. *Crash*: abort the JVM with a fatal error (e.g., segmentation fault). *Error*: exit JVM with error message. *Fault*: suspended by debugger due to an error inside the JVM, which becomes inoperable. *Breakpoint*: suspended by debugger, while JVM remains operable.

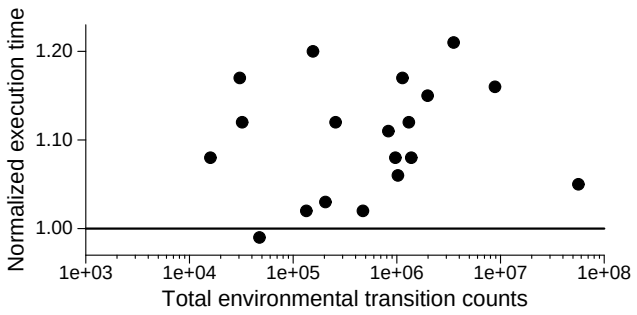

Figure 11. Environmental transitions and time overhead for the Blink debug agent with Hotspot VM 1.6.0_10. Note the logarithmic horizontal axis.

tive method incurs 11 bytes space overhead for its wrapper, instantiated from an assembly code template. Finally, each thread requires 156 bytes of thread-local storage used by the intermediate agent, and less than 160 bytes for each wrapper activation on the stack for an environment transition. We do not measure total space overhead in a live system, since it is small by design.

8.4 Feature Evaluation

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

We examine three common mixed-environment errors: one artificially recreated and two found in JNI programs in the wild. Table 3 lists the programs, lines of code, bug types, and bug sites. Blink directly identifies the two JNI bugs in `UnitTest` and `gconf.BasicGConfApp`. We also recreated an exception-handling bug in `BadErrorChecking`, which is reported as common in both user- and system-level JNI code [13, 29]. For each of these bugs, Table 4 compares Blink to production runs of Hotspot and J9, with runtime checking in Hotspot and J9 (configured with the `-Xcheck:jni` command line option), and with `jdb` and `gdb`.

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

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

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

Null Parameters. Semantics for JNI functions are undefined when their arguments are `(jobject)0xFFFFFFFF` or `NULL` [14]. Hotspot ignores these errors and J9 crashes in `gconf.BasicGConfApp` and `UnitTests`, which pass

NULL to the NewStringUTF JNI function (see Table 4). 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." [14]

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 [24]. J9 crashes and presents a low-level error message with register values and a stack trace. The error message does not include any clue to the cause of the bug. JVM runtime checking does improve the error message.

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

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

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

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

At the call to Java in Line 21, the target Java method foo may raise an exception and then continue with the C code in Line 22, while the JVM has a pending exception. JNI rules require that the C code either returns immediately to the most recent Java caller or invokes the ExceptionClear JNI function. Consequently, the call to the JNI function GetMethodID in Line 22 leaves the JVM state undefined. In fact, Hotspot keeps running while J9 crashes. This rule applies to 209 JNI functions out of 229 functions in JNI 6.0.

Writing the corresponding error checking code is tedious and error-prone. Previous work [13, 29] reports hundreds of bugs in JNI glue code. We briefly inspected the Java-gnome 4.0.10 code base and found two cases of missing error checking. One case never happens unless the JVM implements one JNI function incorrectly. The other case happens only when the JVM is running out of memory, throwing an OutOfMemoryError exception, which is rare and thus hard to find and test. For these reasons, we created the BadErrorChecking micro benchmark.

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

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

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 [35, 36]. 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 [27]. XDI extends an existing Java debugger for C [18]. Both XDI [18] and dbx [27] are powerful but they are less portable than Blink. XDI works only with the Harmony JVM, which is a non-standard JVM. Dbx only works with Sun's JVM on Solaris, and, with limited functionality, on Linux. 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 (Common Language Runtime) [22]. It is also extensible through debug engines [34]. However, in contrast to Blink, where multiple debuggers attach to a single mixed-environment program, each Visual Studio's debug engine is responsible for one program. The CLR provides two debugging APIs: one native and one managed. To handle a mixed-environment program, a debug engine must use both APIs. Given two CLR debuggers, one for the native API and one for the managed API, our compositional approach would yield a mixed-environment debugger.

9.2 Single-Environment Multi-Lingual Debuggers

Some multi-lingual debuggers require all the languages to implement a single interface in the same environment [4, 16, 21]. For example, the GNU debugger, `gdb`, can debug C together with a subset of Java statically compiled by `gcj` [4]. 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" [16], DWARF [5], and even PostScript [19]. 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` [32], and Hanson's machine-independent debugger [11] do not need any extra effort for additional platforms. However, instrumentation causes a factor 3–4 slowdown, which may impede adoption.

Blink leverages portability of its component debuggers, and the construction mechanisms are portable. For reverse engineering, the symbol table for Jeannie discussed in Section 7 is platform-independent. For instrumentation, 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 read-eval-print loop (REPL) for mixed Java and C expressions. Debuggers that support multiple languages, such as `gdb`, often include an interpreter for expressions in each language. Blink is novel in that it interprets expressions by delegating subexpressions to the appropriate single-language debuggers. Blink's REPL uses a syntax for embedding Java in C and vice versa that was developed in an earlier paper on Jeannie [12]. The Jeannie paper described the language and its compiler, but did not describe an interpreter, let alone a debugger.

9.4.2 Mixed-Language Bug Checkers

Another contribution of this paper is Blink's dynamic error checker for Java native interface (JNI) calls. The closest related work is the `-Xcheck:jni` flag, which turns on dynamic error checking in Sun's and IBM's JVMs. Table 3 in [13] 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 class of bugs. J-Saffire infers and checks Java types from C code [8]. Kondoh and Onodera check type-state properties on JNI code based on BEAM [13]. Tan and Croft use static analyses to study security issues in Java's standard library [29]. Of course, static multi-lingual bug checkers are not restricted to Java and C. For example, Quail performs string analysis for Java/SQL safety [31]. Static analysis is complementary to dynamic debugging, which helps find some bugs static analysis misses. Furthermore, the hundreds of bugs in widely-used libraries reported by these papers motivate our work.

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 [2] generates stubs for C to be called from Tcl. SafeJNI [28] combines Java with CCured [17] instead of C. Jeannie [12] provides a type-checked syntactic embedding for Java and C. While these approaches are the right long term solution, they may never be adopted because they require substantial code rewrites. Blink is synergistic since it supports debugging JNI and Jeannie. Furthermore, Blink is an enabling technology for transitioning to better languages.

10. Conclusions

Debugging is one of the most time-consuming tasks in software development. It requires a knack for formulating the right hypotheses about bugs and the discipline to systematically confirm or reject hypotheses until the cause of the bug is found [37]. Single-environment developers have long had

good tools to help them navigate the debugging task systematically. But mixed-language developers have been left in the dark. We propose and evaluate a new way to build cross-environment debuggers more easily using scalable composition. We use our compositional approach to develop Blink, a debugger for Java, C, and Jeannie. The open-source release of Blink is available as part of the xtc package [9]. Blink is full-featured and portable across different JVMs, operating systems, and C debuggers. Furthermore, Blink includes an interpreter (read-eval-print loop) for cross-environment expressions, thus providing users with a powerful tool not just for debugging, but also for testing, program understanding, and prototyping.

Acknowledgments

This work is supported by the National Science Foundation (CCF-0429859, CNS-0448349, CNS-0615129, CNS-0719966, and CCF-0811524), 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] C. Bailey. Java technology, IBM style: Introduction to the IBM developer kit. `http://www.ibm.com/developerworks/java/library/j-ibmjava1.html`, May 2006.

[2] D. M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *USENIX Tcl/Tk Workshop (TCLTK)*, 1996.

[3] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2006.

[4] P. Bothner. Compiling Java with GCJ. `http://www.linuxjournal.com/article/4860`, Jan. 2003.

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

[6] M. Furr and J. S. Foster. Checking type safety of foreign function calls. In *Programming Language Design and Implementation (PLDI)*, 2005.

[7] M. Furr and J. S. Foster. Polymorphic type inference for the JNI. In *European Symposium on Programming (ESOP)*, 2006.

[8] M. Furr and J. S. Foster. Checking type safety of foreign function calls. *Transactions on Programming Languages and Systems (TOPLAS)*, 30(4), 2008.

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

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

[11] D. R. Hanson. A machine-independent debugger—revisited. *Softw. Pract. Exper.*, 29(10):849–862, 1999.

[12] M. Hirzel and R. Grimm. Jeannie: Granting Java native interface developers their wishes. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

[13] G. Kondoh and T. Onodera. Finding bugs in Java native interface programs. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2008.

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

[15] J. Lind-Nielsen. BuDDy. `http://buddy.sourceforge.net/`.

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

[17] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Principles of Programming Languages (POPL)*, 2002.

[18] V. Providin and C. Elford. Debugging native methods in Java applications. In *EclipseCon User Conference*, Mar. 2007.

[19] N. Ramsey and D. R. Hanson. A retargetable debugger. In *Programming Language Design and Implementation (PLDI)*, 1992.

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

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

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

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

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

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

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

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

[28] G. Tan, A. W. Appel, S. Chakradhar, A. Raghunathan, S. Ravi, and D. Wang. Safe Java native interface. In *International Symposium on Secure Software Engineering (ISSSE)*, 2006.

[29] G. Tan and J. Croft. An empirical security study of the native code in the JDK. In *Usenix Security Symposium (SS)*, 2008.

[30] G. Tan and G. Morrisett. ILEA: Inter-language analysis

across Java and C. In *Object-Oriented Programming Systems and Applications (OOPSLA)*, 2007.

[31] Z. Tatlock, C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. Deep typechecking and refactoring. In *Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, 2008.

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

[33] M. van den Brand, B. Cornelissen, P. Olivier, and J. Vinju. TIDE: A generic debugging framework — tool demonstration. *Electronic Notes in Theoretical Computer Science*, 141(4), 2005.

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

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

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

[37] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, Oct. 2005.