2015 30th IEEE/ACM International Conference on Automated Software Engineering

Mutation-Based Fault Localization for Real-World Multilingual Programs

Shin Hong*, Byeongcheol Lee†, Taehoon Kwak*, Yiru Jeon*, Bongsuk Ko†, Yunho Kim*, Moonzoo Kim*
*KAIST, South Korea
{hongshin, thkwak, podray, kimyunho}@kaist.ac.kr, moonzoo@cs.kaist.ac.kr
†GIST, South Korea
{byeong, bsk}@gist.ac.kr

***Abstract*—Programmers maintain and evolve their software in a variety of programming languages to take advantage of various control/data abstractions and legacy libraries. The programming language ecosystem has diversified over the last few decades, and non-trivial programs are likely to be written in more than a single language. Unfortunately, language interfaces such as Java Native Interface and Python/C are difficult to use correctly and the scope of fault localization goes beyond language boundaries, which makes debugging multilingual bugs challenging. To overcome the aforementioned limitations, we propose a mutation-based fault localization technique for real-world multilingual programs. To improve the accuracy of locating multilingual bugs, we have developed and applied new mutation operators as well as conventional mutation operators. The results of the empirical evaluation for six non-trivial real-world multilingual bugs are promising in that the proposed technique identifies the buggy statements as the most suspicious statements for all six bugs.**

I. Introduction

Many software systems today are written in multiple programming languages to reuse legacy code and leverage the languages best suited to the developers' needs. Over the last few decades language designers have made a variety of choices in designing the syntax and semantics of their languages. The result is a robust ecosystem where a few languages cover the most use in part due to open source libraries and legacy code while many languages exist for niche uses [33]. This ecosystem is likely to make developers write a *multilingual program* which is a non-trivial program written in more than a single language. High-level languages such as Java, Python, and OCaml provide standard libraries, which typically call legacy code written in low-level languages (e.g., C) to interface with the operating system. A number of projects for the legacy libraries that have evolved for decades provide language bindings for each language. A large scale software project employs a number of libraries written in multiple languages.

Correct multilingual programs are difficult to write in general in part due to the complex language interfaces such as Java Native Interface (JNI) and Python/C, which require the programs to respect a set of thousands of interface safety rules over hundreds of application interface functions [26], [30]. Moreover, once a bug occurs at interactions of code written in different languages, programmers are required to understand the cause-effect chains across language boundaries. Despite the advances of automated testing techniques for complex real-world programs [14], [19], [20], [21], [39], debugging multilingual bugs in real-world programs is still challenging and requires significant human effort. For instance, Bug 322222 in the Eclipse bug repository crashes JVMs with a segmentation fault in C as an effect when the program throws an exception in Java as the cause (Section VI). Locating and fixing this bug took a heroic debugging effort of more than a year from 2009 to 2010 with hundreds of comments from dozens of programmers before the patch went into Eclipse 3.6.1 in September 2010.

The existing error detectors targeting multilingual program errors [23], [26], [27], [29], [28], [41], [42], [43], [45] are not effective in debugging this case, because they can only report certain kinds of safety rule violations; they cannot indicate the *root cause* of the bug, especially when the bug does not explicitly involve any known safety rule violations. Moreover, these bug detectors do not scale well to a large number of languages and various kinds of program errors since they have to deeply analyze the semantics of each language for each kind of bug.

This paper presents a mutation-based fault localization (MBFL) technique for multilingual programs. The technique takes multilingual source code of a target program and a set of test cases including at least one failing test case as input; it then and generates a list of statements ordered by their relevance to the error (i.e., suspiciousness score). To calculate the suspiciousness score of a statement, a MBFL technique first generates diverse variants of target programs by systematically changing each statement (i.e., mutants), and then observes how testing results change if a certain statement is mutated. In addition, to improve the accuracy of localizing multilingual bugs (e.g., bugs whose causes and effects are located in code segments written in different languages), we have developed new mutation operators focusing on localizing multilingual bugs [1].

The proposed MBFL technique for real-world multilingual programs is effective (i.e., it identifies the locations of bugs precisely) and language agnostic (i.e., extensible for combination of various programming languages). Our empirical evaluation of six real-world Java/C bugs demonstrates that the proposed technique locates the bugs in non-trivial real-world multilingual programs far more precisely (i.e., the technique identifies the buggy statements as the most suspicious statements for all six bugs) than do the state-of-the-art spectrum based fault localization (SBFL) techniques (Section IV). For

[1] The proposed technique is an extension for multilingual bugs based on MUSE which targets to localize C bugs [34].

978-1-5090-0025-8/15 $31.00 © 2015 IEEE
DOI 10.1109/ASE.2015.14

example, for Bug 322222 in the Eclipse bug repository, the technique indicates the statement at which the developer made a fix as the most suspicious statement among a total of 3482 candidates (Section VI). In summary, this paper's major contributions are:

1) New mutation operators targeting multilingual program errors which are highly effective at locating multilingual bugs (Section III-C)
2) Empirical demonstration of the high accuracy of the mutation-based fault localization technique for the six real-world multilingual bugs (Section IV)
3) Detailed report on two case studies to determine why and how the proposed technique can precisely localize real-world multilingual bugs (Sections V and VI).

The rest of the paper is organized as follows. Section II describes the background on multilingual debugging and fault localization techniques. Section III explains our MBFL technique. Section IV provides an overview of the empirical study on the six real-world multilingual bugs. Sections V and VI describe two case studies on real-world multilingual bugs in detail. Section VII discusses observations made through the experiment. Section VIII concludes this paper with future work.

II. Background and Related Work

A. Multilingual Bugs

A multilingual program is composed of several pieces of code in different languages that execute each others through language interfaces (e.g., JNI [30] and Python/C). These language interfaces require the multilingual programs to follow safety rules across language boundaries. Lee *et al.* [26] classifies safety rules in Java/C programs into three classes: (1) state constraints, (2) type constraints, and (3) resource constraints:

- *State constraints* ensure that the runtime system of one language is in a consistent state before transiting to/from a system of another language. For instance, JNI requires the program to not propagate a Java exception before executing a JNI function from a native method in C.
- *Type constraints* ensure that the programs in different languages exchange valid arguments and return values of expected types at a language boundary. For instance, the `NewStringUTF` function in JNI expects its arguments not to be `NULL` in C.
- *Resource constraints* ensure that the program manages resources correctly. These resource constraints are comparable to the contracts of calling the `free` function for dynamically allocated memory in C. For example, a local reference l to an Java object obtained in a native method m_1 should not be reused in another native method m_2 since l becomes invalid when m_1 terminates [30] (see Section V as an example of a multilingual bug that violates this resource constraint).

A multilingual bug is caused by violating safety rules at language interface (i.e., foreign function interface (FFI) bugs), and/or by unintended interactions of code across language boundaries. When a program breaks an interface safety rule, the program crashes or generates undefined behaviors. Multilingual programs respecting all interface safety rules still can have multilingual errors when the cause-effect chain goes through languages interfaces. For instance, a program would leak a C object referenced by a Java object that is garbage collected at some point. The cause of the memory leak is in Java at the last reference to this Java object while the effect is in C because Java code is expected to free the C object (Section III-A).

B. Debugging Multilingual Bugs

Debugging a program bug consists of the following three steps: (1) detecting an error, (2) locating the root cause of the error (i.e., buggy statements), and (3) creating a fix on the buggy statements. These three steps are more challenging for multilingual programs than for monolingual programs because interactions among different languages should be considered, which increases the complexity of debugging.

For the first step (i.e., detecting a multilingual error), there exist dozens of static and dynamic analysis techniques [23], [26], [27], [28], [29], [43], [45]. Some of these techniques provide bug-checkers that detect/predict interface safety rule violations (for example, CheckJNI which is a built-in dynamic JNI checkers in JVMs such as HotSpot and J9). The other techniques [13], [18], [48] detect an error in one programming language while the root cause would be in other languages.

Unfortunately, few techniques support the second step (i.e., fault localization of multilingual bugs). Although the aforementioned static and dynamic analysis techniques can detect/predict multilingual errors, locating the buggy statements that cause the multilingual errors is still challenging because the root cause of multilingual errors is often nontrivial and located far from the error sites (for examples, see Sections V–VI). Although multilingual debuggers may support programmers in locating the causes of the bugs manually [25], it still takes a considerable amount of time to localize a complex multilingual bug (e.g., Bug 322222 of the Eclipse bug repository).

C. Mutation-Based Fault Localization

Fault localization techniques [31], [46] aim to locate the root cause of an error in the target program (i.e., the second step of debugging) by observing test runs. Fault localization has been extensively studied for monolingual programs both empirically [17], [34], [40] and theoretically [47], [50].

Spectrum-based fault localization (SBFL) techniques infer that a code entity is suspicious for an error if the code entity is likely executed when the error occurs. Note that SBFL techniques are *language agnostic* because they calculate the suspiciousness scores of target code entities by using information on the testing results (i.e., fail/pass) of test cases and the code coverage of these test cases without complex semantic analyses. However, the accuracy of SBFL techniques is often too low to localize faults in large real-world programs.

To improve the accuracy of fault localization, mutation-based fault localization techniques (MBFL) have been proposed recently; these techniques can analyze diverse program behaviors using mutants (i.e., target program versions that are generated by applying simple syntactic code changes such as replacing `if(x>10)` with `if(x<10)`). MBFL techniques are also language agnostic since they utilize only information on the testing results (i.e., fail/pass) of test cases on the original target program and its mutants. Moon *et al.* [34] demonstrate that their MBFL technique (calling it MUSE) is 6.5 times more precise than state-of-the-art SBFL techniques such as Ochiai and Op2 on the 15 versions of the SIR subjects. The key idea of MUSE is as follows. Consider a faulty program P whose execution with some test cases results in error. Let m_f be a mutant of P that mutates the faulty statement, and m_c be one that mutates a correct statement. MUSE assesses the suspiciousness of a statement based on the following two observations:

Observation 1 : *a failing test case for P is more likely to pass on m_f than on m_c.* Mutating a faulty statement is more likely to cause the tests that failed on P to pass on m_f than on m_c because a faulty program might be partially fixed by modifying (i.e., mutating) a faulty statement, but not by mutating a correct one. Therefore, the number of test cases whose results change from fail to pass will be larger for m_f than for m_c.

Observation 2 : *a passing test case for P is more likely to fail on m_c than on m_f.* A program is more easily broken by mutating a correct statement than by mutating a faulty statement. Thus, the number of the test cases whose results change from pass to fail will be greater for m_c than for m_f.

There exist a few other MBFL approaches. To localize faults precisely, Zhang *et al.* [51] measure fault-inducing changes in regression testing and Papadakis *et al.* [37], [38] measure mutant similarities. In contrast, MUSE utilizes the differences introduced by mutants for fault localization.

III. Mutation-Based Fault Localization for Real-world Multilingual Programs

To alleviate the difficulty of debugging multilingual programs, we have developed a MUtation-baSEd fault localization technique for real-world mUltilingual prograMs (MUSEUM). MUSEUM is language-independent because it generates syntactic mutants and statistical reasoning with testing results on a target program and its mutants. MUSEUM does not require special build/runtime environments but only a mutation tool and a coverage measurement tool for target programming languages. This is a great advantage over other debugging techniques which require specific infrastructure such as virtual machines or compilers.

MUSEUM targets both monolingual and multilingual bugs. To localize multilingual bugs precisely, MUSEUM utilizes conventional mutation operators and new mutation operators designed for directly mutating interactions between language interfaces. These new mutation operators (Section III-C) improve the accuracy of MBFL by generating mutants whose testing results are informative to locate multilingual bugs (Section V).

A. Motivating Example

1) Target program: Figure 1 presents a target Java/C program with a memory leak bug failing the assertion at Line 71 [2].

```
1 : /* CPtr.java */
2 : public class CPtr {
3 :   static {System.loadLibrary("CPtr");}
4 :   private final long peer;
5 :   private native long nAlloc();
6 :   private native void nFree(long pointer);
7 :   private native int nGet(long pointer);
8 :   private native void nPut(long pointer, int x);
9 :   public CPtr(){peer = nAlloc();}
10:   public int get(){return nGet(peer);}
11:   public void put(int x){nPut(peer, x);}
12:   public void dispose(){nFree(peer);} }
13:
14: /* CPtr.c */
15: #include <jni.h>
16: #include <stdlib.h>
17: jlong Java_CPtr_nAlloc(JNIEnv *env,jobject o){
18:   jint *p;
19:   p =(jint *)malloc(sizeof (jint)); /*Mutant m1*/
20:   return (jlong)p;
21: }
22: void Java_CPtr_nFree(JNIEnv *env,jobject o,jlong p){
23:   free((void *)p);
24: }
25: jint Java_CPtr_nGet(JNIEnv *env,jobject o,jlong p){
26:     return *(jint *)p;
27: }
28: void Java_CPtr_nPut(JNIEnv *env,jobject o,jlong p,
29  jint x){
30:     *((jint *)p) = x;
31: }
32:
33: /* Client.java*/
34: public class Client {
35:   CPtr m = null;
36:   void add(int x){
37:     m = new CPtr(); /*Mutant m2*/
38:     m.put(x);
39:   }
40:   int remove(){
41:     int x = m.get();
42:     m.dispose();
43:     m = null;
44:     return x;  /*Mutant m3*/
45: } }
46:
47: /* ClientTest.java */
48: import java.util.*;
49: public class ClientTest {
50:   static final List pinnedObj=new LinkedList();
51:   public static Object pinObject(Object o){
52:     pinnedObj.add(o);
53:     return o;
54:   }
55:   void passingTest(){ // passing test case
56:     try {
57:       Client d = new Client() ;
58:       d.add(1) ;
59:       assert d.remove() == 1;
60:     } catch(VirtualMachineError e) {
61:       assert false; /*potential memory leak in C*/
62:     }
63:   }
64:   void failingTest(){ // failing test case
65:     try {
66:       Client d = new Client() ;
67:       d.add(1) ;
68:       d.add(2) ;
69:       assert d.remove() == 2;
70:     } catch (VirtualMachineError e) {
71:       assert false; /*potential memory leak in C*/
72:     }
73: } }
```

Fig. 1: A Java/C program leaking memory in C after garbage collection in Java

[2] This example is a simplified version of a real-world bug found in Azureus 3.0.4.2 (Bug1 in Table II).

The program is composed of source files in C and Java defining three Java classes: CPtr, Client, and ClientTest.

CPtr (Lines 2–31) characterizes the peer class idiom [30, p. 123] of wrapping native data structures, which is widely used in language bindings for legacy C libraries. The peer field (Line 4) is an opaque pointer from Java to C to point to a dynamically allocated integer object in C. The CPtr constructor (Line 9) executes the nAlloc native method (Lines 17–21) to allocate an integer object in C and stores the address of the integer object in peer. While JVMs automatically reclaim a CPtr object once the object becomes unreachable in the Java heap, the clients of CPtr are required to dispose manually the integer object by executing dispose (Line 12) on the CPtr object. If the client does not dispose an CPtr object before it becomes unreachable, the peer integer object becomes a unreachable memory leak in C.

Client (Lines 34–45) is a client Java class of using CPtr. The m field (Line 35) holds a reference to a CPtr object. add (Lines 36–39) and remove (Lines 40–45) write/read a value to/from the CPtr object respectively. add instantiates a CPtr object, assigns the reference of the new object to m, and then writes a value to the object. remove reads the value of the CPtr object pointed by m, disposes the CPtr object, deletes the reference to the object, and returns the value of the CPtr object.

ClientTest (Lines 48–73) is a Java class of driving test cases directly for Client and indirectly for CPtr. It contains one passing test passingTest (Lines 55–63) and one failing test failingTest (Lines 64–73). The testing oracle validates a program execution by using (1) the assertion statements (Lines 59 and 69) and (2) the exception handler statements (Lines 61 and 71). The assertion statements at Line 59 and Line 69 validate the program state after executing a sequence of add and remove by checking if remove correctly returns the last value given by add. On the other hand, the exception handler statements at Line 60 and Line 70 detect failures at arbitrary locations. For instance, runtime monitors such as QVM [10] and Jinn [26] would throw an asynchronous Java exception either at GC safe points or at language transitions.

2) Passing test: passingTest executes successfully. It satisfies the assertion statement at Line 59 because both the CPtr object and the peer integer object in Java and C are reachable, and remove at Line 59 returns 1 stored at Line 58. The runtime monitor does not throw any Java exception indicating a memory leak in C because the native integer object is released in the call to remove.

3) Failing test: failingTest fails at Line 71 because the runtime monitor throws an exception due to a memory leak in C. The test case creates one Client object (Line 66) and two CPtr objects (Lines 67–68), and two native integer objects. The first native peer integer object is a leak in C heap while all the other objects are reclaimed automatically by garbage collectors and manually by C memory deallocator (i.e., dispose). The first CPtr object and its peer integer object are created in a call to add at Line 67. Both become unreachable after the second call to add at Line 68. The CPtr object would be garbage collected while the program does not manually execute dispose on the unreachable native integer peer object. The runtime monitor would perform a garbage collection and find out the native integer peer object is a unreachable memory leak. This memory leak bug appears because add does *not* call dispose if m already points to a CPtr object. Thus, we indicate Line 37 as the buggy statement.

4) Our approach: MUSEUM generates mutants each of which is obtained by mutating one statement of the target code. Then, MUSEUM checks the testing results of the mutants to localize buggy statements. For example, suppose that MUSEUM generates the following three mutants m_1, m_2, and m_3 by mutating each of Lines 19, 37, and 44.

m_1, *a mutant obtained by removing Line 19*
This mutation resolves the memory leak as the mutant will not allocate any native memory. However, both test cases fail with the mutant because an access to p raises an invalid memory access (at nGet/nPut of CPtr).

m_2, *a mutant obtained by inserting a statement of pinning the Java reference before Line 37* [3]
This mutation inserts a statement of pinning the object: ClientTest.pinObject(m); before Line 37, where pinObject stores the Java reference m into a global data structure pinnedObj. This mutation intends to prolong the lifetime of the Java object referenced by m to the end of the program run. This mutation resolves the memory leak in failingTest because the first CPtr object will not be reclaimed and, thus, will not leak its peer native integer object. The two test cases pass with the mutant because the mutation does not introduce any new bug.

m_3, *a mutant obtained by replacing the return value with 0 in Line 44*
This mutation replaces the variable x with an integer constant 0 at Line 44. This mutation fails the assertion at Lines 59 and 69 since the return value of remove is always 0.

From these testing results, MUSEUM concludes that Line 37 is more suspicious than Line 19 and Line 44 because the failing test case passes only with m_2 and the passing test case fails with m_1 and m_3 (see Step 4 of Section III-B).

Locating the root cause of this memory leak poses challenges in runtime monitoring and fault localization techniques. Memory leak detectors [18], [49] locate memory leaks and their allocation sites not the cause of the leaks in general. While some leak chasers [10], [11], [48] locate the cause of memory leak, they do not scale well across language boundaries since they do not track opaque pointers and their staleness values across languages. SBFL techniques cannot localize the bug because both passingTest and failingTest cover the same branches/statements in their executions. Consequently, SBFL techniques cannot indicate any code element that is more correlated with the failure than the others.

B. Fault Localization Process of MUSEUM

Figure 2 describes how MUSEUM localizes faults. MUSEUM takes the target source code and the test cases of the target program as input, and returns the suspiciousness scores

[3] See Pin-Java-Object mutation operator in Table I

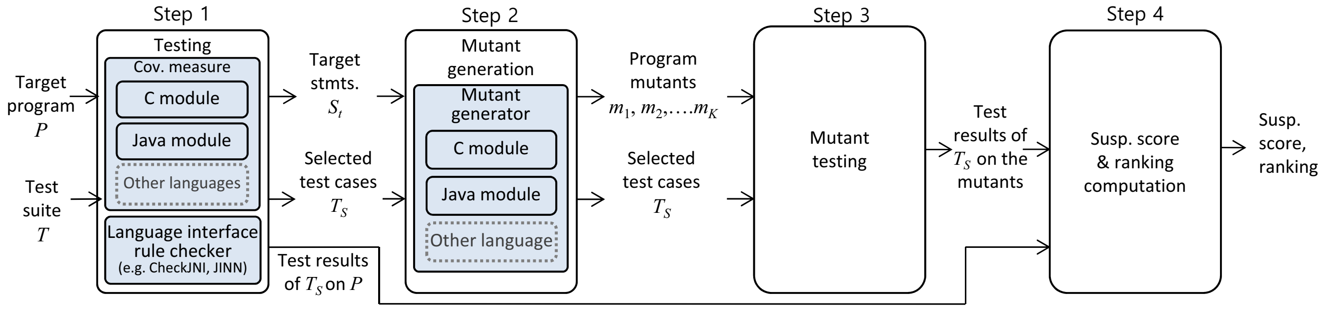
Step 1
Testing
Cov. measure
C module
Java module
Other languages
Language interface rule checker (e.g. CheckJNI, JINN)
Target program P
Test suite T
Target stmts. S_t
Selected test cases T_S
Test results of T_S on P
Step 2
Mutant generation
Mutant generator
C module
Java module
Other language
Program mutants $m_1, m_2, \ldots m_K$
Selected test cases T_S
Step 3
Mutant testing
Test results of T_S on the mutants
Step 4
Susp. score & ranking computation
Susp. score, ranking

Fig. 2: Fault localization process of MUSEUM

of the target code lines as output. MUSEUM has the following basic assumptions on a target program P and test suite T:

1. Existence of test oracles
   A target program has explicit or implicit test oracle mechanism (i.e., user-specified `assert`, runtime failure such as null-pointer dereference, and/or runtime monitor such as Jinn [26]) which can detect errors clearly.
2. Existence of a failing test case
   A target program has test cases, at least one of which violates a test oracle.

MUSEUM operates in the following four steps:

Step 1: MUSEUM receives P and T and selects target statements S_t and test cases T_s. S_t is the set of the statements of P that are executed by at least one failing test case in T. MUSEUM selects S_t as target statements for bug candidates. Also, MUSEUM selects and utilizes a set of test cases T_s, each of which covers at least one target statement because the other test cases may not be as informative as test cases in T_s for fault localization. To select S_t and T_s, MUSEUM first runs P with T while storing the test results and the test coverage for each test case. Testing results are obtained from the user given assert statements, runtime failures, and multilingual bug checkers such as CheckJNI, Jinn [26], and QVM [10] (Section II-A).

Step 2: MUSEUM generates mutant versions of P (i.e., $m_1, m_2, ...m_k$) each of which is generated by mutating each of the target statements. MUSEUM may generate multiple mutants from a single statement since one statement may contain multiple mutation points [9]. [4]

Step 3: MUSEUM tests all generated mutants with T_S and records the testing results. Since a mutation may induce an infinite loop, we consider a test fails if the testing time exceeds a given time limit.

Step 4: MUSEUM compares the test results of T_S on P with the test results of T_S on all mutants. Based on these results, MUSEUM calculates the suspiciousness scores of the target statements of P as follows.

For a statement s of P, let $f(s)$ be the set of tests that covers s and fails on P, and $p(s)$ the set of tests that covers s and passes on P. Let $mut(s) = \{m_1, \ldots m_k\}$ be the set of all mutants of P that mutates s.

For each mutant $m_i \in mut(s)$, let f_{m_i} and p_{m_i} be the set of failing and passing tests on m_i respectively. And let $f2p$ and $p2f$ be the numbers of changed test result from fail to pass and vice versa between P and all mutants of P. The suspiciousness metric of MUSEUM is defined as follows:

$$Susp(s) = \frac{1}{|mut(s)|}\sum_{m_i \in mut(s)}\left(\frac{|f(s) \cap p_{m_i}|}{f2p} - \frac{|p(s) \cap f_{m_i}|}{p2f}\right)$$

The first term, $\frac{|f(s) \cap p_{m_i}|}{f2p}$, reflects the first observation: it is the proportion of the number of tests that failed on P but now pass on a mutant m_i that mutates s over the total number of all failing tests that pass on a some mutant (the suspiciousness of s increases if mutating s causes failing tests to pass). Similarly, the second term, $\frac{|p(s) \cap f_{m_i}|}{p2f}$, reflects the second observation, being the proportion of the number of tests that passed on P but now fail on a mutant m_i that mutates s over the total number of all passing tests that fail on a some mutant (the suspiciousness of s decreases if mutating s causes passing tests to fail). After dividing the sum of the first term and the second term by $|mut(s)|$, $Susp(s)$ indicates the probability of s to be a faulty statement based on the changes of test results on P and $mut(s)$. [5]

C. New Mutation Operators for Multilingual Bugs

In addition to the conventional mutation operators, MUSEUM utilizes new mutation operators to effectively localize multilingual bugs because these mutation operators can directly mutate interactions between language interfaces. We have made 11 new mutation operators, which change semantics of a target program regarding the JNI constraints based on the previous JNI bug studies [6], [12], [26], [30], [44]. Table I shows the list of the new mutation operators. The description of the new mutation operators are as follows:

1. `Clear-pending-exceptions` clears a pending exception by inserting
   `(*env)->ExceptionClear(env);`
   immediately after every JNI function invocation in C
   (i.e., `(*env)->`$<JNIFunction>$`(...);`).
   This mutation operator is created based on a best practice in JNI programming [12].

[4] MUSEUM can localize a bug spanning on multiple statements (not limited for locating a single-line bug). This is because mutating a part of a bug (i.e., one statement among multiple statements that constitute a bug) can still change a failing test case into passing one, which will increase the suspiciousness of the statement constituting the bug [34].

[5] If a target statement has no mutant (i.e., $|mut(s)|$=0), $Susp(s)$ is defined as 0. MUSEUM defines the first term as 0 if $f2p$ is 0. Similarly, the second term is defined as 0 if $p2f$ is 0. For a concrete example of how to calculate the suspiciousness score of MBFL, see Section II.C of Moon *et al.* [34].

2. `Propagate-pending-exceptions` propagates an exception immediately by inserting

```
if((*env)->ExceptionOccurred(env)) return;
```

after every JNI function invocation in C.

3. `Type-cast-to-jboolean` explicitly converts an integer expression to `JNI_TRUE` or `JNI_FALSE` when the expression is assigned to a `jboolean` variable. [6] In other words, `Type-cast-to-jboolean` changes an assignment `jbool_var = int_expr;` with

```
jbool_var=int_expr?JNI_TRUE:JNI_FALSE;
```

This mutation operation is motivated by the common pitfall of JNI programming [30, pp.132–133].

4. `Type-cast-to-superclass` changes a JNI call to get the reference of a class with the JNI call to get the reference of its superclass by mutating `jclass cls = (*env)->GetObjectClass(env,obj);` with

```
jclass cls = (*env)->GetSuperclass(env,
((*env)->GetObjectClass(env,obj)));
```

This mutation operator is motivated by a report of a real-world bug found in Eclipse 3.4 [26].

5–10. These mutation operators increase or decrease the life time of a reference to a Java object (and probably the life time of the referenced Java objects too). For example, `Make-global-reference` increases the life time of a local reference *l* by making the reference as a global one. In other words, `Make-global-reference` inserts the following statement after an assignment statement to a local reference *l* (i.e., *l = expr*):

```
l = (*env)->NewGlobalRef(env,l);
```

In contrast, `Remove-global-reference` decreases the life time of a global reference *g* (and probably the referenced Java object too) by inserting the following statement for a global reference *g*:

```
(*env)->DeleteGlobalRef(env,g);
```

We have developed four other mutation operators for local references and weak global references. These mutation operators are related to a bug fix pattern regarding reference errors in native code [6].

11. `Pin-Java-object` prevents garbage collectors from reclaiming a Java object by placing a Java reference to the object into a class variable in Java before a reference to the object is removed by an assignment statement. Before an assignment statement `x = obj;`, the mutation operator inserts a statement:

```
Test.pinnedObjects.add(x) ;
```

where `Test.pinnedObjects` is a Java class variable of a list container type. The Java object pointed by `x` is transitively reachable from the class variable, and Java garbage collectors cannot reclaim the object. This mutation operator intends to extend the lifetime of Java objects in a target program and influence interactions of Java and native memory managements. This mutation operator is inspired by a safe memory management scheme of SafeJNI [44].

TABLE I: New mutation operators of MUSEUM

No.	Mutation operator	Corresponding language interface rule (Section II-A)
1	`Clear-pending-exceptions`	JVM state constraints
2	`Propagate-pending-exceptions`	
3	`Type-cast-to-jboolean`	Type constraints
4	`Type-cast-to-superclass`	
5	`Make-global-reference`	Resource constraints
6	`Remove-global-reference`	
7	`Make-weak-global-reference`	
8	`Remove-weak-global-reference`	
9	`Make-local-reference`	
10	`Remove-local-reference`	
11	`Pin-Java-object`	

D. Implementation

We have implemented MUSEUM targeting programs written in Java and C (support for other languages will be added later). MUSEUM is composed of the existing mutation testing tools for C and Java, together with the fault localization module that analyzes testing results and computes suspiciousness scores. MUSEUM consists of 1,500 lines of C/C++ code and 1,802 lines of Java code.

MUSEUM uses gcov and PIT [1] to obtain the coverage information on C code and Java code of a target program, respectively. MUSEUM uses existing mutation tools Proteum[32] and PIT, together with the 11 new mutation operators for multilingual bugs (Section III-C). Proteum implements 107 mutation operators defined in Agrawal *et al.* [9] which mutate C code in source level. Among the 107 mutation operators, MUSEUM uses 75 mutation operators that change only one statement. To reduce the runtime cost of the experiments, MUSEUM generates only one mutant for every applicable operator at each mutation point [7]. MUSEUM generates Java mutants by using PIT which mutates Java bytecode. MUSEUM uses all 14 mutation operators of PIT. Among the 11 new mutation operators, 10 new mutation operators for C code are implemented with Clang, and the one new mutation operator for Java (i.e., `Pin-Java-object`) is built with the ASM bytecode engineering tool.

IV. Empirical Evaluation

This section evaluates MUSEUM on the six bugs in four real-world multilingual programs to demonstrate its effectiveness. Section IV-A describes the experiment setup, and Section IV-B presents the fault localization results. [8]

A. Experiment Setup

1) Real-world multilingual program bugs: Table II presents the six multilingual bugs in four real-world programs with

[6] `jboolean` is an 8 bit integer type. If a 32 bit integer value is assigned to a `jboolean` variable, the variable can have an unintended Boolean value due to the truncation (e.g., `jboolean_var = 256` will make `jboolean_var` as false).

[7] For example, `if(x+2>y+1)` has one mutation point (`>`) for ORRN (mutation operator on relational operator) and two points (`2` and `1`) for CCCR (mutation operator for constant to constant replacement) [9]. MUSEUM generates only one mutant like `if(x+2<y+1)` using ORRN and only `if(x+0>y+1)` and `if(x+2>y+0)` using CCCR. The selection of a mutant to generate using a mutation operator depends on the Proteum implementation. Note that MUSEUM generates multiple mutants for a code location when multiple mutation operators are applied to the code location .

[8] The full experiment data and the target program code are available at http://swtv.kaist.ac.kr/data/museum.zip.

TABLE II: Target multilingual Java/C bugs, sizes of the target code, the number of test cases used, and references

Bug	Target program	Symptom	Size of target program				# of TC used	Bug report or bug-fixing revision
			Java		NativeC			
			Files	LOC	Files	LOC		
Bug1	`Azureus` 3.0.4.2	Memory leak in C	2,705	340.6K	N/A	N/A	8	CVS revision 1.64 of ListView.java [2]
Bug2	`sqlite-jdbc` 3.7.8	Assertion violation in Java	20	4.6K	3	1.8K	150	Issue 16 [7]
Bug3	`sqlite-jdbc` 3.7.15	Assertion violation in Java	19	4.2K	2	1.7K	159	Issue 36 [8]
Bug4	`java-gnome` 4.0.10	Invalid JNI reference in C	1,097	64.2K	496	65.6K	170	Bug 576111 in Bugzilla database [3]
Bug5	`java-gnome` r-658	Double free in C	1,134	67.1K	514	69.2K	184	Subversion revision 659 [4]
Bug6	`SWT` 3.7.0.3	Segmentation fault in C	582	118.7K	29	20.7K	50	Bug 322222 in the Eclipse bug repo. [5]

TABLE III: Overview of the experiment data

	Bug1	Bug2	Bug3	Bug4	Bug5	Bug6
# of the target lines (executed by the failing test case)	1,939	299	443	186	186	3,482
# of mutants	2,861	676	942	701	364	8,553
# of lines which have a mutant	1,575	219	327	130	103	2,468
# of mutants that make a passing test case fails (breaking)	305	462	681	364	311	3,044
# of mutants that make a failing test case passes (partial fix)	1	3	7	2	51	32
Time cost (min)	12	64	52	95	60	246

their programs, symptoms, line of code (LOC) in Java and C, the number of the test cases used to localize the fault, and bug reports or bug-fixing revisions of the target programs. As described in the assumption 1 for fault localization (Section III-B), the bug reports and commit logs in the last column describe the symptoms of the target bugs so that our test oracle detects test failures. A corresponding bug report indicates both buggy version and its fixed version. We have applied MUSEUM to Java code and native C code of the target program, not library code nor external system code. All target programs are written in Java and C except for `Azureus`. While `Azureus` is a pure Java program, it triggers a memory leak in C when it misuses the application program interface of the Eclipse SWT library written in Java and C.

2) Test Cases: Regarding test cases, we have used the test cases maintained by the developers of the target programs. We utilize the test cases of the fixed version, at least one of which reveals the target bug in the buggy version (see the assumption 2 in Section III-B). If the fixed version does not have a test case that fails on the buggy version (e.g., `Azureus` memory leak bug), we create a failing test case based on the bug report. In addition, to localize a fault precisely, we focus to localize one bug at a time by building a new test suite out of the original test suite. The new test suite consists of one failing test case and all passing test cases that cover at least one statement executed by the failing test case.

3) System Platform: The experiments were performed on the 30 machines equipped with Intel i5 3.4 GHz with 8 GB main memory (we performed experiment on one core per machine). All machines run Ubuntu 8.10 32-bits, gcc 4.3.2, and OpenJDK 1.6.0. MUSEUM distributes tasks of testing each mutant to the 30 machines. [9]

B. Experiment Results

Table III reports the experiment data on the six bugs. The second row counts the number of the source target lines which are executed by the failing test case (see Step 1 of Section III-B). The third row shows the total number of the mutants generated by MUSEUM, and the forth row describes the total number of the target lines on which at least one mutant is generated. The fifth and sixth rows show the number of the mutants on which testing results have changed. The last row describes the runtime cost. For example, to localize the fault in Bug4 (an invalid JNI reference in C), we built a test suite containing one failing test case and 169 passing test cases out of the original test suite (see the eighth column of the fifth row at Table II). For Bug4, MUSEUM generated 701 mutants with at least one mutant for 70% of the target lines (=130/186). Among the 701 mutants, there are two mutants on which the failing test case passes (see the sixth row of Table III). [10] We call such mutants as "partial fix" because the failing test case passes on the mutant (but passing test cases may fail on these mutants). The table shows that only 0.14% of the mutants are partial fixes (=2/701). However, these mutants contribute significantly to localize a fault precisely because a partial fix increases the first term of the suspiciousness metric formula much (see the metric formula in the Step 4 of Section III-B).

Regarding time cost, although MUSEUM consumes large amount of computing resources to test a large number of mutants, the overall elapsed time can be modest. This is because tasks of testing mutants can be distributed to a large number of machines (e.g., Amazon EC2) as these tasks are independent to each other. For example, it takes around 90 minutes (=(12+64+52+95+60+246)/6) to localize each bug of the six multilingual bugs on average by utilizing 30 machines.

Table IV compares the fault localization results of MUSEUM and the cutting-edge SBFL techniques including Jaccard [16], Ochiai [36], and Op2 [35]. Each entry reports the suspiciousness score ranking which is the maximum number of statements to examine until finding a faulty statement described in the bug report. The percentage number in the parentheses indicates the normalized rank of the faulty statement out of the total target statements (i.e., $\frac{\text{rank}}{\text{\# of the target statements}}$). The second row of the table clearly shows that MUSEUM precisely identifies the buggy statement. MUSEUM ranks the buggy statements in Bug1, Bug3, and Bug4 as the most

[9] We set the time limit (10 seconds) at each test run on a mutant to avoid the infinite loop problem caused by mutation. Time taken to execute a test run was less than one second on the six subjects on average.

[10] The number of mutants that make the failing test case pass is equal to $f2p$ since the test suite contains only one failing test case in our experiments.

TABLE IV: The ranks of the buggy line identified by MUSEUM and other SBFL techniques

	Bug1	Bug2	Bug3	Bug4	Bug5	Bug6
MUSEUM	1 (0.1%)	2 (0.7%)	1 (0.2%)	1 (0.1%)	8 (4.3%)	3 (0.2%)
Jaccard	80 (4.1%)	4 (1.3%)	5 (1.1%)	83 (44.6%)	61 (32.8%)	3,482 (100.0%)
Ochiai	80 (4.1%)	4 (1.3%)	5 (1.1%)	83 (44.6%)	61 (32.8%)	3,482 (100.0%)
Op2	80 (4.1%)	4 (1.3%)	5 (1.1%)	83 (44.6%)	61 (32.8%)	3,482 (100.0%)

suspicious statements (i.e., the first rank). Even for Bug2 and Bug5, MUSEUM identifies the buggy statement as the most suspicious statement with the other one and seven statements together (e.g., for Bug5, the suspiciousness scores of the eight statements including the buggy statement are equal). Thus, from these experiments, we conclude that MUSEUM localizes a multilingual bug precisely.

In contrast, SBFL techniques fail to localize multilingual bugs precisely. For Bug6, Op2 ranks the buggy statement as the 3,482nd among the 3,482 target statements (see the fifth row of Table III), which means that a developer has to examine all target statements (i.e., 100%) to identify the bug. One main deficiency of traditional SBFL techniques is the low resolution in fault localization (i.e., all statements in the same branch have same suspiciousness scores because the statements in the same branch are covered by the same test cases). This is one reason why those SBFL techniques assign the same suspicious ranks to the buggy statements in the experiments. In contrast, MUSEUM mutates each statement in multiple different ways and can assign different suspiciousness scores to the statements in the same branch.

V. Case Study 1: Locating The Cause of Invalid Use of JNI References (Bug4)

This case study illustrates how MUSEUM localizes the cause of dangling JNI references (Bug4) accurately by using the new mutation operators (Table I).

A. Bug Overview

Dynamic error detectors [26] detect Bug4 and report the calling context at the fault location of using the dangling JNI reference as an argument to a JNI function. However, they cannot report the cause location where the JNI reference was stored into a callback object in C heap, which occurs at Line 524 of `binding_java_signal.c` as indicated as the buggy statement in the bug report:

```
387: GClosure* bindings(JNIEnv *env,
     jobject handler,  jclass receiver, ... ) {
...
524:    bjc->rec = receiver;
... }
```

When `bindings` at Line 387 is invoked, the `receiver` parameter is assigned with a local JNI reference. Line 524 stores the local reference in a data structure in the C heap pointed by `bjc`. However, once `bindings` returns back to Java, the local reference stored in `bjc->rec` is not valid anymore (i.e., becoming a dangling reference, see Resource constraints in Section II-A). Later, when the application calls a JNI function with an argument containing the dangling reference, the application crashes with a JNI invalid argument error.

TABLE V: The nine mutants generated by mutating the buggy statement of Bug4

No.	Mutant generated by mutating Line 524 of `bindings_java_signal.c`	$\lvert f(s)$ $\cap p_m \rvert$	$\lvert p(s)$ $\cap f_m \rvert$

$m1$	`bjc->rec=receiver;`
	`bjc->rec=(*env)->NewWeakGlobalRef(env,bjc->rec);`
		1	0

$m2$	`bjc->rec=receiver;`
	`bjc->rec=(*env)->NewLocalRef(env,bjc->rec);`
		0	0

$m3$	`return; // return back to the caller.`	0	0

$m4$	`bjc->rec=receiver;`
	`bjc->rec=(*env)->NewGlobalRef(env,bjc->rec);`
		1	0

$m5$	`; // remove a statement at Line 524`	0	2

$m6$	`bjc->rec=receiver;`
	`(*env)->DeleteGlobalRef(env, bjc->rec);`
		0	2

$m7$	`kill(getpid(), 9); //terminate the process`	0	2

$m8$	`bjc->rec=receiver;`
	`(*env)->DeleteLocalRef(env, bjc->rec);`
		0	2

$m9$	`bjc->rec=receiver;`
	`(*env)->DeleteWeakGlobalRef(env,bjc->rec);`
		0	2

B. Detailed Experiment Result

MUSEUM localizes the fault *exactly* by ranking Line 524 as the most suspicious statement (i.e., the first rank without a tie). Table V describes nine mutants ($m1$ to $m9$) that are generated by mutating Line 524. The second column shows the changed statement of each mutant. The third and the forth columns report the number of tests that failed on the original program but pass on the mutant (i.e., $|f(s) \cap p_m|$), and the number of tests that passed on the original program but fail on the mutant (i.e., $|p(s) \cap f_m|$), respectively (Section III-B). $m1$, $m2$, $m4$, $m6$, $m8$, and $m9$ are generated by applying our new multilingual mutation operators in Table I. These mutants are generated by inserting the statements of changing the life time of JNI references right after the target statement. $m3$, $m5$ and $m7$ from Proteum terminate the control flow at the level of procedure, statement, and whole program.

In the testing runs, our new mutation operators prevent mutated programs $m1$ and $m4$ from crashing in the failing test case (i.e., `Make-weak-global-reference` and `Make-global-reference` in Table I, respectively). $m1$ and $m4$ turn the failing test case into a passing one (the third column) because they keep `bjc->rec` to store a weak global reference and a global reference respectively and eliminate the dead reference problem caused by the short-lived local reference. On the other hand, the conventional mutation operators (i.e., $m3$, $m5$, and $m7$) do not affect the test results. $m1$ and $m4$ make the first term of the MUSEUM suspiciousness metric large and increase the suspiciousness score of Line 524 significantly because the denominator of the first term is small (i.e.,$f2p$=2) (Section III-B). In contrast, each of the mutants $m5$ to $m9$ make two passing test cases fail (the fourth column), which increases the second term but in only limited degree due to the large denominator (i.e., $p2f$=6053).

Among the 186 target statements, only Line 524 has mutants that fix Bug4 with regard to the given test cases and the given test oracle (i.e., making the failing test case pass). Consequently, Line 524 has the highest suspiciousness score due to the new mutation operators which generate partial fixes. Thus, through the case study for Bug4, we confirm that the new mutation operators such as `Make-global-reference` can increase the accuracy of MUSEUM (Section VII-B).

In contrast, the SBFL techniques rank the buggy statement as the 83rd suspicious one among the 186 target statements. Such poor result is due to the two coincidentally correct test cases (CCTs) that execute Line 524 but pass because the target program does not use `bjc->rec` as an argument to a JNI function call later with these test cases. Thus, the SBFL techniques considers Line 524 has low correlation with the failure and assign low suspiciousness score to Line 524.

Note that these CCTs do not make adverse effect to MUSEUM. This is because the mutants (i.e., $m1$ to $m9$) obtained by mutating the buggy statement (i.e., Line 524) do not make these two CCTs fail as the target program and the mutants do not use `bjc->rec` as an argument to a JNI function call later with these CCTs (i.e., the mutation on the buggy statement is inactive with CCTs because the buggy statement is dormant with CCTs). Thus, these CCTs do not increase the second term of the MUSEUM suspiciousness metric (Section III-B) and do not lower the suspiciousness score of the buggy statement.

VI. Case Study 2: Locating the Cause of a Segmentation Fault in Eclipse SWT (Bug6)

This case study in this section demonstrates how MUSEUM accurately localizes a complex multilingual bug whose cause-effect chain is long and complicated, which is often the case for multilingual bugs and makes debugging multilingual bugs very difficult.

A. Bug Overview

Bug 322222 (Bug6) in the Eclipse bug repository for Standard Widget Toolkit (SWT), a standard open-source GUI development library for Java programs crashes JVMs with a fatal segmentation fault by dereferencing `NULL` at Line 271 of `pango-layout.c`:

```
262: PangoLayout *
263: pango_layout_new (PangoContext *context)
264: {
...
271:    layout->context = context;
...
275: }
```

The origin of `NULL` is the native C function (`callback`) that acts as a gateway from C to Java in the SWT library. `callback` returns `NULL` when a Java exception is pending in the current thread. While the detection of this bug is trivial, locating the root cause took a heroic debugging effort for more than a year with hundreds of comments from dozens of programmers. This bug was difficult for experts to debug since the cause-effect chain goes through Java exception propagation and language transitions. Although the multilingual debuggers [24] aid programmers to locate the origin of `NULL`, they do not locate the root cause of the bug.

The root cause is turned out to be an immature implementation of a callback handler at Line 2602 of `Display.java`. [11]

```
// Simplified patch for Bug6
2595  :if(OS.GTK_VERSION>= OS.VERSION(2,4,0)) {
...
2601--:     OS.G_OBJ_CONSTRUCTOR(PLClass);
2602--:     OS.G_OBJ_SET_CONSTRUCTOR(PLClass,
                 newProc);

2601++:     p = OS.G_OBJ_CONSTRUCTOR(PLClass);
2602++:     OS.G_OBJ_SET_CONSTRUCTOR(PLClass,
                 new NewProcCB(p));
```

This patch replaces the `newProc` that calls `callback` at Line 2602 with a new `NewProcCB(p)` object that calls another callback function that never returns `NULL` at the presence of a pending exception. Although the location of the failure in C at the segmentation fault is fairly far away from the callback handler in Java, MUSEUM locates the root cause of the failure as most suspicious (i.e., the suspiciousness rank of Line 2602 is 3 as Line 2602 is tied with other two statements).

B. Detailed Experiment Result

We utilize a test suite consisting of one failing test case and 49 passing test cases. We selected these 49 passing test cases that cover the display module of SWT because all error traces in the bug report contain a method in the display module.

Table VI presents the top four suspicious statements and their mutants, which increase the suspiciousness scores. [12] MUSEUM ranks the first three statements in a tie as rank 3 that include the location of the root cause of the failure: `Display.java:2602`. The mutants for the top three statements change the failing test case into passing one without affecting passing test cases (see the third and the fourth columns).

These mutants disable the immature callback handler that transitively calls `callback`. The first mutant eliminates Line 2602 of `Display.java` that registers the immature callback handler. The second and the third mutants change the return value with zero, which in turn reverses the control flow decision at Line 2595 of `Display.java`, deactivates transitively Line 2602 of registering the immature callback handler, and avoids the segmentation fault. The fourth mutant disables the immature callback handler at the cost of turning one passing test case into a failing one, which decreases the suspiciousness of Line 2392 and lowers its rank to 4.

VII. Discussions

A. Advantages of Mutation-based Fault Localization for Real-world Multilingual Programs

One of the issues that make debugging real-world programs difficult is the poor quality of a test suite because fault localization can be more accurate if a test suite covers more diverse execution paths. For large real-world programs, however, it is challenging to build test cases that exercise diverse execution paths because it is non-trivial to understand and

[11] The bug report on Bug6 does not describe the root cause of the crash but only its symptom, which is often the case for real-world applications. Thus, we had to identify the buggy statement by analyzing the bug patch.

[12] All of the top four statements have only one mutant due to the limitation of PIT which supports only small number of mutation operators for Java code compared to Proteum for C code.

TABLE VI: The top four statements of the SWT target code whose suspiciousness scores are high

Rank	Susp. score	$\lvert f(s) \cap p_m \rvert$	$\lvert p(s) \cap f_m \rvert$	Statement	Mutant

3	0.0313	1	0	`/*Display.java:2602*/`
				`OS.G_OBJ_SET_CONSTRUCTOR`
				`(PLClass,NewProc);`
					`; /* the function`
					`call is removed */`

3	0.0313	1	0	`/*OS.java:8115*/`
				`return _major_version;`
					`return 0;`

3	0.0313	1	0	`/*OS.java:8125*/`
				`return _minor_version;`
					`return 0;`

4	0.0306	1	1	`/*Display.java:2392*/`
				`initializeSubclasses();`
					`; /* the function`
					`call is removed */`

TABLE VII: Statistics on the mutation operators that generate mutants in the experiments

	Bug1	Bug2	Bug3	Bug4	Bug5	Bug6
# of tests where the failing TC passes on the mutants	1	3	7	2	51	32
# of mutation operators that generate a mutant on which a failing TC passes	1	3	6	2	12	14
# of tests where the failing TC passes on the mutants generated by the new mutation operators	1	0	0	2	2	0
# of the new mutation operators that generate a mutant on which a failing TC passes	1	0	0	2	1	0

control a target program. In addition, generating diverse test cases for multilingual programs has additional burden to learn and satisfy constraints for foreign function interface such as JNI constraints. Therefore, it is often the case that multilingual programs are developed with a set of similar test cases. As a result, as shown in Table IV, the SBFL techniques fail to precisely localize the six real-world multilingual programs.

For example, the statement coverages of the test suites used for Bug2 and Bug3 are around 85% and 86% and the SBFL techniques localize these bugs somehow precisely (i.e., the suspiciousness rank of Bug2 and Bug3 are 4 and 5, respectively). However, the statement coverages of the test suites used for Bug1, Bug4, Bug5, and Bug6 are around 1%, 22%, 24%, and 19% and the accuracy of the SBFL techniques for these bugs are very low (Table IV). In contrast, MUSEUM can overcome this limitation by achieving the effect of diverse test cases through the diverse mutants with limited test cases. Thus, MUSEUM can be a promising technique for debugging complex real-world multilingual programs.

B. Effectiveness of New Mutation Operators for Localizing Multilingual Bugs

Table VII presents the information on the mutation operators that generate mutants on which the failing test case passes (i.e., partially-fixing mutants). The second row shows the number of tests where the failing test case on the target program passes on a mutant. The third row represents the number of mutation operators that generate a mutant on which the failing test case changes to pass. The fourth and the fifth rows show the similar information to the second and the third rows but on the mutants generated by the new mutation operators for multilingual programs (Section III-C).

Table VII shows that only the new mutation operators generate partially fixing mutants for Bugs 1 and 4 (i.e., since the numbers in the third row and the fifth row are the same). For Bug1, only the `Pin-Java-Object` mutation operator generates a mutant on which the failing test case passes, which indicates that the target statement of the mutant is closely related to the bug (i.e., memory leak in this case). Similarly, for Bug4, only `Make-global-reference` and `Make-weak-global-reference` generate the mutants that make the failing test case pass.

The table shows that the new mutation operators are effective to mutate multilingual program behaviors and discover critical code points related to the JNI constraints. To assess the impact of the new mutation operators on fault localization, we ran MUSEUM for Bugs 1, 4 and 5 without the new mutation operators. For Bugs 1 and 4, the suspiciousness ranks of the faulty lines become 1737 for Bug1 (89.6%) and 117 (62.9%) for Bug4. For Bug5, the rank of the faulty line changes from 8 to 9 (the faulty line no longer has the highest suspiciousness score). This result implies that language-interface specific mutation operators can effectively supplement the existing mutation operators for finding multilingual bugs.

For the other four bugs, the existing mutation operators generate more partially fixing mutants than the new ones. Among the 89 (=75 mutation operators for C + 14 mutation operators for Java) existing mutation operators, the top-3 operators that generate a large number of partially fixing mutants are ‘remove a function call’, ‘remove a statement’, and ‘change the return value at a return statement’. These three mutation operators generate mutants on which 46 failing tests pass out of total 96 failing tests that pass on mutants (see the second row of the table). We found that these three operators generate mutants that change function call/return behaviors. We conjecture that mutating function call/return effectively changes multilingual behaviors of a target program as the interaction between different languages are made through function calls.

VIII. Conclusion and Future Work

We have presented MUSEUM which localizes bugs in complex real-world multilingual programs in a language agnostic manner through mutation analyses. The experiments on the six real-world multilingual programs show that MUSEUM precisely locates the faulty statement for all non-trivial Java/C bugs. In addition, we show that the accuracy of fault localization for multilingual programs can be increased by adding new mutation operators relevant with FFI constraints.

As future work, we will add more mutation operators targeting features in multilingual programs and modify existing mutation operators to reduce equivalent mutants. Also, we will apply MUSEUM to an interactive debugger such as Blink [24] and/or advanced automated testing techniques [15], [22] to maximize the debugging effectiveness. Finally, we will investigate effective methods to utilize MUSEUM to improve automated program repair and/or search-based program analysis for multilingual programs.

Acknowledgement

This work is supported by the NRF Mid-career Research Program by the MSIP (NRF-2012R1A2A2A01046172); the ITRC support program (IITP-2015-H8501-15-1012) and the ICT R&D program (13-912-06-003) by the MSIP and the IITP; the IITP grant programs funded by the MSIP (Research and Development of Dual Operating System Architecture with High-Reliable RTOS and High-Performance OS [No. R0101-15-0081]; High Performance Big Data Analytics Platform Performance Acceleration Technologies Development [No. R0190-15-2012]).

References

[1] PIT - mutation testing tool for Java. http://pitest.org.

[2] Azureus-commitlog: Listview.java. http://sourceforge.net/p/azureus/mailman/message/18318135/, 2008.

[3] GNOME Bug 576111 - Unit test failed on SUN Hotspot and IBM J9 with -Xcheck:jni. https://bugzilla.gnome.org/show_bug.cgi?id=576111, 2009.

[4] Java-GNOME Avoid segfault lurking in GtkSpell library. https://openhub.net/p/java-gnome-gstreamer/commits/167384488, 2009.

[5] Eclipse SWT Segfault in pango_layout_new when closing a dialog. http://bugs.eclipse.org/bugs/show_bug.cgi?id=322222, 2010.

[6] JNI Local Reference Changes in ICS. Android Developers Blog. http://android-developers.blogspot.com/2011/11/jni-local-reference-changes-in-ics.html, 2011.

[7] Xerial SQLite-JDBC, Issue 16: DDL statements return result other than 0. https://bitbucket.org/xerial/sqlite-jdbc/issue/16, 2012.

[8] Xerial SQLite-JDBC, Issue 36: Calling PreparedStatement.clearParameters() after a ResultSet is opened, causes subsequent calls to the ResultSet to return null. https://bitbucket.org/xerial/sqlite-jdbc/issue/36, 2013.

[9] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford. Design of mutant operators for the C programming language. Technical Report SERC-TR-120-P, Purdue University, 1989.

[10] M. Arnold, M. Vechev, and E. Yahav. QVM: An efficient runtime for detecting defects in deployed systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(1):2:1–2:35, 2011.

[11] J. Clause and A. Orso. LEAKPOINT: Pinpointing the causes of memory leaks. In *International Conference on Software Engineering (ICSE)*, 2010.

[12] M. Dawson, G. Johnson, and A. Low. Best practices for using the Java Native Interface. IBM developerWorks, 2009.

[13] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[14] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *ACM Conference on Program Language Design and Implementation (PLDI)*, 2005.

[15] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold. Testing concurrent programs to achieve high synchronization coverage. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2012.

[16] P. Jaccard. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bull. Soc. vaud. Sci. nat*, 37:547–579, 1901.

[17] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *IEEE/ACM International Conference on. Automated Software Engineering (ASE)*, 2005.

[18] C. Jung, S. Lee, E. Raman, and S. Pande. Automated memory leak detection for production use. In *International Conference on Software Engineering (ICSE)*, 2014.

[19] M. Kim, Y. Kim, and Y. Choi. Concolic testing of the multi-sector read operation for flash storage platform software. *Formal Aspects of Computing (FAC)*, 24(2), 2012.

[20] M. Kim, Y. Kim, and Y. Kim. Industrial application of concolic testing approach: A case study on libexif by using CREST-BV and KLEE. In *International Conference on Software Engineering (ICSE) Software Engineering In Practice Track*, 2012.

[21] Y. Kim, Y. Kim, T. Kim, G. Lee, Y. Jang, and M. Kim. Automated unit testing of large industrial embedded software using concolic testing. In *IEEE/ACM Automated Software Engineering (ASE) Experience track*, 2013.

[22] Y. Kim, Z. Xu, M. Kim, M. Cohen, and G. Rothermel. Hybrid directed test suite augmentation: An interleaving framework. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2014.

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

[24] B. Lee, M. Hirzel, R. Grimm, and K. S. McKindley. Debugging mixed-environment programs with blink. *Software: Practice and Experience*, 2014. DOI 10.1002/spe.2276.

[25] B. Lee, M. Hirzel, R. Grimm, and K. S. McKinley. Debug all your code: Portable mixed-environment debugging. In *ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2009.

[26] B. Lee, B. Wiedermann, M. Hirzel, R. Grimm, and K. McKinley. Jinn: Synthesizing dynamic bug detectors for foreign language interfaces. In *ACM Conference on Program Language Design and Implementation (PLDI)*, 2000.

[27] S. Li and G. Tan. Finding bugs in exceptional situations of JNI programs. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[28] S. Li and G. Tan. JET: exception checking in the java native interface. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, 2011.

[29] S. Li and G. Tan. Finding reference-counting errors in Python/C programs with affine analysis. In *European Conference on Object-Oriented Programming (ECOOP)*, 2014.

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

[31] M.A.Alipour. Automated fault localization techniques: a survey. Technical report, Oregon State University, 2012.

[32] J. C. Maldonado, M. E. Delamaro, S. C. Fabbri, A. da Silva Simão, T. Sugeta, A. M. R. Vincenzi, and P. C. Masiero. Proteum: A family of tools to support specification and program testing based on mutation. In *Mutation Testing for the New Century*. 2001.

[33] L. A. Meyerovich and A. S. Rabkin. Empirical analysis of programming language adoption. In *ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2013.

[34] S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2014.

[35] L. Naish, H. J. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(3):11:1–11:32, August 2011.

[36] A. Ochiai. Zoogeographic studies on the soleoid fishes found in Japan and its neighbouring regions. *Bull. Jpn. Soc. Sci. Fish.*, 22(9):526–530, 1957.

[37] M. Papadakis and Y. Le-Traon. Using mutants to locate "unknown" faults. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Mutation Workshop, 2012.

[38] M. Papadakis and Y. Le-Traon. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability (STVR)*, To be published.

[39] Y. Park, S. Hong, M. Kim, D. Lee, J. Cho, M. Kim, Y. Kim, and Y. Kim. Systematic testing of reactive software with non-deterministic events: A case study on LG electric oven. In *International Conference on Software Engineering (ICSE) Software Engineering In Practice Track*, 2015.

[40] R.Abreu, P.Zoeteweij, and A. Gemund. An evaluation of similarity coefficients for software fault localization. In *Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2006.

[41] T. Ravitch, S. Jackson, E. Aderhold, and B. Liblit. Automatic generation of library bindings using static analysis. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[42] T. Ravitch and B. Liblit. Analyzing memory ownership patterns in C libraries. In *International Symposium on Memory Management (ISMM)*, pages 97–108, 2013.

[43] J. Siefers, G. Tan, and G. Morrisett. Robusta: taming the native beast of the JVM. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[44] G. Tan, A. Appel, S. Chakradhar, A. Raghunathan, S. Ravi, and D. Wang. Safe Java Native Interface. In *IEEE International Symposium on Secure Software Engineering (ISSSE)*, 2006.

[45] G. Tan and G. Morrisett. Ilea: inter-language analysis across Java and C. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

[46] E. Wong and V. Debroy. A survey of software fault localization. Technical Report UTDCS-45-09, University of Texas at Dallas, 2009.

[47] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):31, 2013.

[48] G. Xu, M. D. Bond, F. Qin, and A. Rountev. LeakChaser: Helping programmers narrow down causes of memory leaks. In *ACM Conference on Program Language Design and Implementation (PLDI)*, 2011.

[49] G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. In *International Conference on Software Engineering (ICSE)*, 2008.

[50] S. Yoo, X. Xie, F.-C. Kuo, T. Y. Chen, and M. Harman. No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist. RN/14/14, Department of Computer Science, University College London, 2014.

[51] L. Zhang, L. Zhang, and S. Khurshid. Injecting mechanical faults to localize developer faults for evolving software. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013.