MUSEUM: Debugging Real-World Multilingual Programs Using Mutation Analysis

Shin Hong[a], Taehoon Kwak[b], Byeongcheol Lee[c,*], Yiru Jeon[b], Bongseok Ko[c], Yunho Kim[b], Moonzoo Kim[b]

[a]*Handong Global University, 558 Handong-ro, Buk-gu, Pohang, Gyeongbuk, South Korea 37554*
[b]*Korea Institute of Science and Technology, 291 Daehak-ro, Yuseong-gu, Daejeon, South Korea 34141*
[c]*Gwangju Institute of Science and Technology, 123 Cheomdangwagi-ro, Buk-gu, Gwangju, South Korea 61005*

Abstract

Context: The programming language ecosystem has diversified over the last few decades. Non-trivial programs are likely to be written in more than a single language to take advantage of various control/data abstractions and legacy libraries.
Objective: Debugging multilingual bugs is challenging because language interfaces are difficult to use correctly and the scope of fault localization goes beyond language boundaries. To locate the causes of real-world multilingual bugs, this article proposes a mutation-based fault localization technique (MUSEUM).
Method: MUSEUM modifies a buggy program systematically with our new mutation operators as well as conventional mutation operators, observes the dynamic behavioral changes in a test suite, and reports suspicious statements. To reduce the analysis cost, MUSEUM selects a subset of mutated programs and test cases.
Results: Our empirical evaluation shows that MUSEUM is (i) effective: it identifies the buggy statements as the most suspicious statements for both resolved and unresolved non-trivial bugs in real-world multilingual programming projects; and (ii) efficient: it locates the buggy statements in modest amount of time using multiple machines in parallel. Also, by applying selective mutation analysis (i.e., selecting subsets of mutants and test cases to use), MUSEUM achieves significant speedup with marginal accuracy loss compared to the full mutation analysis.
Conclusion: It is concluded that MUSEUM locates real-world multilingual bugs accurately. This result shows that mutation analysis can provide an effective, efficient, and language semantics agnostic analysis on multilingual code. Our light-weight analysis approach would play important roles as programmers write and debug large and complex programs in diverse programming languages.

Keywords: debugging, mutation analysis, language interoperability, foreign function interface

1. Introduction

Modern software systems are written in multiple programming languages to reuse legacy code and leverage the languages best suited to the developers' needs such as performance and productivity. A few languages cover the most use in part due to open source libraries and legacy code while many languages exist for niche uses [30]. This ecosystem encourages developers to write a *multilingual program* which is a non-trivial program written in multiple languages. Correct multilingual programs are difficult to write due to the complex language interfaces such as Java Native Interface (JNI) and Python/C that require the programs to respect a set of thousands of interface safety rules over hundreds of application interface functions [22, 26]. Moreover, if a bug exists at interactions of code written in different languages, programmers have to understand the cause-effect chains across language boundaries [21].

Despite the advance of automated testing techniques for complex real-world programs, debugging multilingual bugs (e.g., a bug whose cause-effect execution chain crosses language boundaries) in real-world programs is still challenging and consumes 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 [21]. Locating and fixing this bug took a heroic debugging effort for 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 bug detectors targeting multilingual bugs [20, 22, 24, 25, 40, 41, 44] are not effective in debugging this case, because they can only report violations of predefined interface safety rules, but cannot indicate the location of the bug, especially when the bug does not involve any known safety rule violations explicitly. Moreover, these bug detectors do not scale well to a large number of languages and

[*] Corresponding author. Tel: +082 627152245.
Email addresses: `hongshin@handong.edu` (Shin Hong), `thkwak@kaist.ac.kr` (Taehoon Kwak), `byeong@gist.ac.kr` (Byeongcheol Lee), `podray@kaist.ac.kr` (Yiru Jeon), `bsk@gist.ac.kr` (Bongseok Ko), `kimyunho@kaist.ac.kr` (Yunho Kim), `moonzoo@cs.kaist.ac.kr` (Moonzoo Kim)

various kinds of program bugs since they have to deeply analyze the semantics of each language for each kind of bug.

This article proposes MUSEUM, a mutation-based fault localization technique which locates the cause of a multilingual bug by observing how mutating a multilingual code feature changes the target program behaviors. Mutation-based fault localization (MBFL) is an approach recently proposed for locating code lines that cause a test failure accurately. An MBFL technique takes target source code and a test suite including failing test cases as input, and assesses suspiciousness of each statement in terms of its relevance to the error through mutation analysis of target code. In more detail, an MBFL technique calculates suspiciousness scores of statements by observing how testing results (i.e., pass/fail) change if the statement is modified/mutated. MUSEUM extends an MBFL technique MUSE [31] which is limited for targeting monolingual bugs (i.e., bugs in C). MUSEUM applies new mutation operators that systematically modify the multilingual features/behaviors of a target program (see Section 3.3).

Our empirical evaluation on the eight real-world Java/C bugs (Sections 4– 7) demonstrates that MUSEUM locates the bugs in non-trivial real-world multilingual programs far more accurately than the state-of-the-art spectrum based fault localization techniques. MUSEUM identifies the buggy statements as the most suspicious statements for all eight bugs (Section 4). For example, for Bug 322222 in the Eclipse bug repository, MUSEUM indicates the statement at which the developer made a fix as the most suspicious statement among total 3,494 candidates (Table 2). Furthermore, one case study on an unresolved Eclipse bug (i.e., an open bug whose fix is not yet made) clearly demonstrates that MUSEUM generates effective information for developers to identify and fix the bug (Section 7).

In summary, this article's contributions are:

1. An automated fault localization technique (i.e., MUSEUM) which is effective to detect multilingual bugs which are known as notoriously difficult to debug.
2. New mutation operators on multilingual behavior which are highly effective to locate multilingual bugs (Section 3.3)
3. Detailed report of the three case studies to figure out why and how the proposed technique can localize real-world multilingual bugs accurately (Sections 5–7).

This article extends our prior conference publication [15] in three ways: (i) Section 3.3 elaborates the program mutation with the four additional mutation operators to increase the accuracy of localizing multilingual bugs (ii) Sections 5–6 describe the case studies on two additional resolved bugs (Bug5 and Bug7).[1] Also, Section 7 illustrates a case study on one unresolved open bug (Bug8) to demonstrate how MUSEUM can guide developers to debug a complex multilingual bug (iii) Section 8 shows that MUSEUM can significantly speedup the fault localization with marginal accuracy loss by selecting subsets of mutants and test cases to use.

2. Background and Related Work

2.1. 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 [26] and Python/C). These multilingual programs introduce new classes of programming bugs which obsolete the existing monolingual debugging tools and require much more debugging efforts of programmers than monolingual programs [21, 43]. We classify multilingual bugs into *language interface bugs* and *cross-language bugs*.

2.1.1. Language Interface Bugs

Language interfaces require multilingual programs to follow safety rules across language boundaries. Lee *et al.* [22] classify safety rules in Java/C programs into the following three classes:

- *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 that the program is not propagating 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. For example, a local reference l to a 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 [26].

For instance, the manuals for JNI [26] and Python/C describe thousands of safety rules over hundreds of API functions. When a program breaks an interface safety rule, the program crashes or generates undefined behaviors [22].

2.1.2. Cross-Language Bugs

Cross-language bugs have a cause-effect chain that goes through language interfaces while respecting all interface safety rules. For instance, a program would leak a C object referenced by a Java object that is garbage collected at some point without violating any safety rules of language interfaces. In this case, the cause of the memory leak is in Java at the last reference to this Java object while the effect is in C (see Section 3.1).

[1] The full description of all eight case studies is available at `http://swtv.kaist.ac.kr/publications/museum-techreport.pdf`.

2.2. Mutation-Based Fault Localization (MBFL)

Fault localization techniques [45] aim to locate the buggy statement that causes an error in the target program by observing test runs. Fault localization has been extensively studied for monolingual programs both empirically [18, 31, 39] and theoretically [46].

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 semantics agnostic* because they calculate the suspiciousness scores of target code entities by using 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 are often too low for large real-world programs.

To improve the accuracy of fault localization, MBFL is proposed recently, which analyze diverse program behaviors by using mutants (i.e., target program versions that are generated by applying simple syntactic code change such as replacing `if(x>10)` with `if(x<10)`). MBFL techniques are also language semantics agnostic since they utilize only the testing results (i.e., fail/pass) of test cases on the original target program and its mutants. Moon *et al.* [31] demonstrate that their MBFL technique (calling it MUSE) is 6.5 times more precise than the 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 executions with some test cases result 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 on P is more likely to pass on m_f than on m_c.* Mutation 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 the 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 on 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 m_f.

Our intuition behind Observation 1 is that we can view a bug at line l as a result of mutation operator $\mathcal{M}$ to l and there can be another mutation operator $\mathcal{M}'$ among many ones which works as $\mathcal{M}^{-1}$ (an inverse function of $\mathcal{M}$) with some context of l and some test cases. Also, an intuition of Observation 2 is that a correct statement is more fragile than a faulty statement in terms of pass/fail results. Note that the aforementioned observations are on multiple statements to compare relative suspiciousness scores identify more suspicious statements than the others (e.g., a statement s_1 is more suspicious than s_2 and s_3). Moon *et al.* [31] showed that these observations are valid through the experiments on the 15 versions of SIR subjects (e.g., the number of the failing test cases on P that pass on m_f is 1,435.9 times larger than the number on m_c on average). Also note that Observation 2 is important because Observation 2 can serve as a tiebreaker by differentiating statements that are equally suspicious in terms of Observation 1 (for instance, see the case study results on Bug 7 (Section 5.2)).

There exist a few other MBFL approaches which focus on different characteristics of various executions caused by mutants. Papadakis and Traon developed Metallaxis-FL [37] which evaluates the suspiciousness of code elements by using the similarity of the behaviors of the mutants and the faulty program. The intuition of Metallaxis-FL is that a mutant m_1 has higher suspiciousness than another mutant m_2 if more *failing* tests kill m_1 than m_2 because m_1 is more sensitive to the characteristics of faulty executions than m_2. Metallaxis-FL considers a code element l whose mutants (i.e., mutants generated by mutating l) have high suspiciousness as a faulty statement. Zhang et el. [49] use mutation analysis to find a fault-inducing change between an old correct program P and a new faulty program P' in regression testing. This approach takes a regression test suite T, an old and correct program P with respect to T, and a new and faulty program P'. The intuition is that if a change c made by mutation to P makes test results similar to those of P', the code location changed by c is highly suspicious because c is similar to the fault in P'. Consequently, this technique reports a change between P and P' which is similar to c as a fault-inducing change.

3. 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).

3.1. Motivating Example

This section illustrates how MUSEUM locates the cause of a non-trivial bug in a target multilingual program with passing and failing test cases.

3.1.1. Target Program

Figure 1 presents a target Java/C program with a memory leak bug failing the assertion at Line 71. 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 [26, 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.

```
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: } }
```

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

`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 Lines 60 and 70 detect failures at arbitrary locations. For instance, runtime monitors such as QVM [28] would take a user-specified typestate specification of disposing native resources of a Java object before it becomes unreachable, detect a failure to dispose these native resources during garbage collection, and throw an asynchronous `OutOfMemoryError` exception at a GC safe point.

3.1.2. 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`.

3.1.3. 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 an unreachable memory leak (e.g., QVM [28], Jinn [22]). 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.

3.1.4. 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*
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 `pinnedObjects` (see `Pin-Java-Object` mutation operator in Section 3.3).

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 on m_2 and the passing test case fails on m_1 and m_3 (see Step 4 of Section 3.2).

Locating the root cause of this memory leak poses challenges in runtime monitoring and fault localization techniques. Memory leak detectors [19, 48] locate memory leaks and their allocation sites, not the cause of the leaks in general. While some leak chasers [28, 17, 47] 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 cannot localize the bug because both `passingTest` and `failingTest` cover the same branches/statements in their executions (i.e., SBFL cannot indicate any code element that is more correlated with the failure than the others).

3.2. Fault Localization Process of MUSEUM

MUSEUM takes the target source code and the test cases as input, and returns the suspiciousness scores of the target code lines as output. MUSEUM has the following basic assumptions on a target program P and a test suite

1. Existence of test oracles
   A target program has test oracle mechanism (i.e., user-specified `assert`, runtime failure such as null-pointer dereference, and/or runtime monitor such as Jinn [22]) 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 [22], and QVM [28].
- **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 [11]. 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 [31].

  To reduce the runtime cost, MUSEUM generates only one mutant for every applicable operator at each mutation point. 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) [11]. 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.

- **Step 3:** MUSEUM tests all generated mutants with T_S and records the testing results. MUSEUM runs a mutant with a passing test case only if the test case covers the mutated statement. We consider a test fails if the testing time exceeds a given time limit since a mutation may induce an infinite loop. Note that this step can be parallelized on multiple machines for fast fault localization.

- **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 $mut(s)$. 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.

3.3. New Mutation Operators for Multilingual Behavior

In addition to the conventional mutation operators which targets monolingual features of C [11] or Java [1], MUSEUM utilizes new mutation operators to directly mutate interactions at language interfaces and effectively localize multilingual bugs. We introduce 15 new mutation operators which change the semantics of a target program regarding the JNI constraints based on the language interface specifications [10, 26] and the previous bug studies [5, 29, 13, 22, 42].

3.3.1. New Mutation Operators for State Constraints

1–3. The `Clear-pending-exceptions` mutation operator clears a pending exception in a native method to ensure the JVM state constraints. Similarly, `Propagate-pending-exception` and `Throw-new-exceptions` propagate or generate a pending exception in a native method. Targets of these three mutation operators are all JNI function calls (i.e., `(*env)->`< *JNIFunction* >`(...);`). For example, `Clear-pending-exceptions` clears a pending exception in a current thread by inserting

```
(*env)->ExceptionClear(env);
```

immediately before a JNI function call and immediately after a JNI function call that may throw a Java exception.[2] `Propagate-pending-exceptions` propagates a pending exception to the caller by inserting

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

immediately before a JNI function call and immediately after a JNI function call that may throw a Java exception. `Throw-new-exceptions` creates a new Java exception by inserting

```
Throw_New_Java_Exception(env,
"java/lang/Exception");
```

immediately before a JNI function call and immediately after a JNI function call that may throw a Java exception. As exception handling is a regular feature of the Java control-flow, a native function is obligated to create, modify, or eliminate Java exceptions depending on execution paths. The suggested mutation operators intend to alter an exception-related JNI function call to check if the JNI function call is related to the multilingual bug. The first and the second mutation operators are defined based on a best practice in JNI programming [29] and general solutions for JNI exception bugs [23]. The third mutation operator is motivated by a case of a real-world multilingual bug regarding exception handling across language boundaries [8].

3.3.2. New Mutation Operators for Type Constraints

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

[2] 154 among total 229 JNI functions may throw an exception [26].

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

`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). This mutation operation is motivated by the common pitfall of JNI programming [26, pp.132–133].

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

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

This mutation operator would generate interesting mutants for fault localization if the target bug is related with an incorrect casting. This mutation operator is motivated by a report of a real-world bug found in Eclipse 3.4 [22].

6. `Replace-array-elements-with-constants` replaces a Java array reference with another constant Java array. This mutation operator changes a Java array reference used at a JNI function call to the reference to the predefined constant array. For example, this mutation operator change `(*env)->GetIntArrayElements(env, arr, null);` into

```
(*env)->GetIntArrayElements(env,
IntConstArr, null);
```

This mutation operator intends to mutate the values in an array copied from Java to C. This mutation is inspired by a real-world bug with an incorrect array data transfer from Java to C [2].

7. `Replace-target-Java-member` replaces a target field in a class member access with the field of a different class member with the same type, by mutating `(*env)->GetFieldID(env, class, NAME1, SIG);` with

```
(*env)->GetFieldID(env, class,
NAME2, SIG);
```

where `NAME1`, `NAME2`, and `SIG` are the strings of the original and the changed field names and their type signature, respectively. This mutation operator is motivated by a common pitfall in JNI programming [26, pp.131–132].

3.3.3. New Mutation Operators for Resource Constraints

8–13. There are six mutation operators, `Make-global-reference`, `Remove-global-reference`, `Make-weak-global-reference`, `Remove-weak-global-reference`, `Make-local-reference` and `Remove-local-reference`, each of which increases or decreases the life time of a reference to a Java object (and probably the life time of the referenced Java object 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 [5].

14. `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 a 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 management. This mutation operator is inspired by a safe memory management scheme of SafeJNI [42].

15. `Switch-array-release-mode` alternates the release mode of a Java array access. The release mode decides whether an updated native array will be copied back to the Java array or discarded. For every `(*env)->Release<Type>ArrayElements(env, arr, elems, mode)`, this mutation operator changes the `mode` value from `0` to `JNI_ABORT`, or vice versa. This mutation operator is motivated by a best practice in JNI programming [29].

3.4. 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 to obtain the coverage information on C code and Java code of a target program, respectively.

MUSEUM uses the mutation tools Proteum/IM 2.0 [27] for C and PIT version 0.33 for Java bytecode [1] together with the 15 new mutation operators for multilingual behaviors (Section 3.3). Proteum/IM implements 107 mutation operators defined in Agrawal *et al.* [11]. Among the 107 mutation operators, MUSEUM uses 75 mutation operators that change only one statement. PIT implements 14 mutation operators all of which are used by MUSEUM. Among the 15 new mutation operators, 14 new mutation operators for C code are implemented with Clang version 3.4, and the one new mutation operator for Java (i.e., `Pin-Java-object`) is built with the ASM bytecode engineering tool version 3.3.1.

4. Experiment Setup and Result

We have evaluated the effectiveness of MUSEUM on the eight bugs in four real-world multilingual software projects. The full experiment data and the target program code are available at `http://swtv.kaist.ac.kr/data/museum.zip`.

4.1. Experiment Setup

4.1.1. Real-world Multilingual Program Bugs

Table 1 presents the eight multilingual bugs in four real-world software projects with 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. Azureus is a popular P2P file-sharing application. Sqlite-jdbc is a Java Database Connectivity (JDBC) library to access the SQLite relational database management system written in C. Java-gnome is a set of language bindings for the Java programming language for use in the GNOME desktop environment. SWT (Standard Widget Toolkit) is an Eclipse widget toolkit for Java to provide user-interface facilities. We selected these projects as target projects because these projects have multilingual bugs that had been analyzed by other practitioners and researchers.

The bug reports/commit logs in the last column describe the symptoms of the target bugs. A corresponding bug report indicates both buggy version and its fixed version. All target programs are written in Java and C (except `Azureus` which is a pure Java program but triggers a memory leak in C when it misuses the application program interface of the Eclipse SWT library written in Java and C).

4.1.2. Test Cases

We 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. If the fixed version has no test case that fails on the buggy version, we create a failing test case based on the bug report. For Bug1, since Azureus code repository has no test case, we created one failing test case and seven passing test cases to cover reasonable fraction of the source files. In addition, for those test cases which require manual operations, we carefully encoded the operations described in the bug reports. [3]

To localize a fault accurately, 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.

4.1.3. System Platform

The experiments were performed on the 30 machines equipped with Intel i5 3.4 GHz with 8 GB 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. We set the time limit (10 seconds) for 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 eight subjects on average.

4.2. Experiment Results

Table 2 reports the experiment data on the eight bugs. The second row shows the number of the target source lines executed by the failing test case (see Step 1 of Section 3.2). The third row shows the total number of the mutants generated by MUSEUM, and the fourth 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 Bug4, 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 of Table 1). MUSEUM generated 718 mutants (at least one mutant for 71% of the target lines (=132/186)). Among the 718 mutants, there are two mutants on which the failing test case passes (see the sixth row of Table 2). [4] 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 0.28% of the mutants are partial fixes (=2/718). Note that partial fix mutants at s can largely increase the suspiciousness

[3] We tried to make only unavoidable changes at the original test cases. All edit records are found at the experiment data on the web.

[4] 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 1: Target multilingual Java/C bugs, their symptoms, 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	Rev. 1.64 of ListView.java [3]
Bug2	`sqlite-jdbc` 3.7.8	Assertion violation in Java	20	4.6K	3	1.8K	150	Issue 16 [6]
Bug3	`sqlite-jdbc` 3.7.15	Assertion violation in Java	19	4.2K	2	1.7K	159	Issue 36 [7]
Bug4	`java-gnome` 4.0.10	Invalid JNI reference in C	1,097	64.2K	496	65.6K	170	Bug 576111
Bug5	`java-gnome` r-658	Segmentation fault 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	43.3K	50	Bug 322222 [21]
Bug7	`sqlite-jdbc` 3.6.0	Exception state violation in C	25	4.9K	2	0.6K	112	`UDFTest` bug [21]
Bug8	`SWT` 4.3.0	Segmentation fault in C	591	126.6K	29	48.5K	204	Bug 419729 [9]

Table 2: Overview of the experiment data

	Bug1	Bug2	Bug3	Bug4	Bug5	Bug6	Bug7	Bug8
# of the target lines	1,939	299	443	186	186	3,494	294	4,998
# of mutants	2,861	691	965	718	369	9,479	844	14,490
# of lines which have a mutant	1,575	219	327	132	103	2,524	226	3,855
# of mutants that make a passing test case fails	305	462	681	364	311	3,044	542	8,766
# of mutants that make a a failing test case passes	1	3	7	2	51	32	3	1
Time cost (in minutes)	12	60	45	25	23	175	50	511

score of s since partial fix mutants increase the numerator of the first term of the suspiciousness formula whose denominator $f2p$ is usually small (e.g., 2 for Bug4) (see the formula in the Step 4 of Section 3.2). MUSEUM takes 25 minutes to localize Bug4 using 30 machines.

Table 3 compares the fault localization results of MUSEUM and the cutting-edge SBFL techniques including Jaccard [16], Ochiai [33], and Op2 [32]. Each entry reports the suspiciousness ranking which is the maximum number of the statements to examine until finding the faulty statement described in the bug report. The percentage number in the parentheses indicates the normalized ranking of the faulty statement out of the total target statements (i.e., $\frac{\text{ranking}}{\text{\# of the target statements}}$). The second row of the table clearly shows that MUSEUM accurately identifies the buggy statement. MUSEUM ranks the buggy statements in Bug1, Bug3, Bug4, Bug7, and Bug8 as the most suspicious statements (i.e., the first ranking). Even for Bug2, Bug5, and Bug6, MUSEUM identifies the buggy statement as the most suspicious statement with the other one, seven, and two statements together (e.g., for Bug5, the suspiciousness scores of the eight statements including the buggy statement are equal). Thus, we conclude that MUSEUM localizes a multilingual bug accurately.

In contrast, SBFL techniques fail to localize multilingual bugs accurately. For example, Op2 ranks the buggy statement of Bug6 as the 3,494nd among the 3,494 target statements (see the fifth row of Table 2), which means that a developer has to examine all target statements to identify the faulty statement.

4.3. Threats to Validity

A major external threat to validity is that the experiment uses a limited number of target programs. To limit this threat, we chose the target subjects that include both language interface bugs and cross-language bugs, and have different symptoms and various related language features. Also, we collected these target programs from various real-world projects used by the related work.

Another threat is that the test cases used in the experiments are limited. To limit this threat, we utilized all available test cases in the real-world target subjects (except Azureus that has no test cases for Bug1).

A construct threat is that there may be statements that can be recognized as buggy statements other than the ones indicated by the bug reports/fixes used in the studies. Although there might be other buggy statements, we believe that the conclusions still hold because MUSEUM localized the buggy statements reported by the bug reports/fixes as most suspicious ones.

Possible internal threats are that the target programs may have unidentified nondeterminism and/or the MUSEUM tool may have faults. To limit these threats, we carefully reviewed the target programs, the MUSEUM

Table 3: The ranking of the buggy line identified by MUSEUM and the SBFL techniques

	Bug1	Bug2	Bug3	Bug4	Bug5	Bug6	Bug7	Bug8
MUSEUM	1 (0.1%)	2 (0.7%)	1 (0.2%)	1 (0.1%)	8 (4.3%)	3 (0.2%)	1 (0.2%)	1 (0.02%)
Jaccard	80 (4.1%)	4 (1.3%)	5 (1.1%)	83 (44.6%)	61 (32.8%)	3,494 (100.0%)	84 (17.5%)	574 (10.2%)
Ochiai	80 (4.1%)	4 (1.3%)	5 (1.1%)	83 (44.6%)	61 (32.8%)	3,494 (100.0%)	84 (17.5%)	574 (10.2%)
Op2	80 (4.1%)	4 (1.3%)	5 (1.1%)	83 (44.6%)	61 (32.8%)	3,494 (100.0%)	84 (17.5%)	574 (10.2%)

tools, and the experiment results. For further analysis, full experiment data and the target program code are available at `http://swtv.kaist.ac.kr/data/museum.zip`.

5. Case Study with Language Interface Bug (Bug7)

Language interface bugs violate one of the three classes of the safety rules on language interfaces [22]: state constraints, type constraints, and resource constraints. This section presents the case study of Bug7 to illustrate how MUSEUM locates the causes of the bugs of violating state constraints.

5.1. Bug Overview

Bug7 violates a safety rule on a language interface that the native code must not invoke a JNI function while the current thread is propagating a pending Java exception. For instance, consider Lines 183 and 184 of `NativeDB.c` in the `sqlite-jdbc 3.6.0` source release:

```
/* sqlite/src/main/java/org/sqlite/NativeDB.c */
154: static xCall(...) {
..
183:  (*env)->CallVoidMethod(env, func, method) ;
184:  (*env)->SetLongField(env, func, ...) ;
```

In an erroneous run, the native code at Line 183 invokes a Java method identified by the `method` argument, which throws a Java exception and abruptly returns to the native code. Then, the current thread is propagating the pending Java exception, and the call statement at Line 184 executes the `SetLongField` JNI function. These event series of throwing Java exception and calling a JNI function violate the exception state rule. The semantics of the `SetLongField` JNI function is left undefined, and JVMs may crash [21].

The bug fix checks and clears explicitly the pending Java exception before calling the `SetLongField` JNI function with the following updates:

```
/* sqlite/src/main/java/org/sqlite/NativeDB.c */
154: static xCall(...) {
..
183:  (*env)->CallVoidMethod(env, func, method) ;
+++   if((*env)->ExceptionCheck(env))
+++     xFunc_error(context,env);
184:  (*env)->SetLongField(env, func, ...) ;
```

The conditional part of the inserted statement examines if a Java exception is pending. When a Java exception is pending, `xFunc_error` clears the pending Java exception and records this error state. Then, the native code executes the `SetLongField` JNI function without violating the JNI exception state rule.

5.2. Detailed Experiment Result

We use the 112 tests cases in the Xerial SQLite JDBC regression test suite. MUSEUM successfully finds the location where the developer inserts the new code to fix the bug as the most suspicious statement. Table 4 shows the mutants generated from the top four most suspicious statements. Line 184 of `NativeDB.c` has the highest suspiciousness score because it has two fail-to-pass test runs. Line 183 of `NativeDB.c` is ranked as the second most suspicious statement because it has one fail-to-pass test run. The other statements have no fail-to-pass test run (see the sixth row of Table 2). Note that the three partial fix mutants on Lines 183–184 do not call an JNI function with a pending exception on the failing test case. The statements of the fourth and the fifth rows of Table 4 are ranked as the 114th together with other 110 statements on which their test results do not change at all.

6. Case Study with Cross-Language Bug (Bug5)

Cross-language bugs have their cause-effect chains across a language boundary while respecting all safety rules on language interfaces. To demonstrate how MUSEUM locates the causes of these cross-language bugs, this section presents the case study of Bug5. Bug5 has its effect of a segmentation fault in C while the cause is an attempt to access the freed native peer resource from Java.

6.1. Bug Overview

Bug5 crashes JVMs due to a segmentation fault at Line 738 of `gtkspell.c` in Revision 658 of Java-gnome because the `spell` pointer parameter is dangling:

Table 4: Four most suspicious statements of the Xerial SQLite JDBC target code (Bug7)

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

1	0.111	/* NativeDB.c:184 */
		(*env)->SetLongField(env,...);
			if ((*env)->ExceptionOccured(env))
			  return;
			(*env)->SetLongField(env,...);
				1	16

			return ;	1	16
2	0.055	/* NativeDB.c:183 */ (*env)->CallVoidMethod(env,...);	return ;	1	64

114	0.0	/* Conn.java:81 */
		this.url = url;
			Test.pinnedObjects.add(url);
			this.url = url;
				0	0

114	0.0	/* Conn.java:188 */ checkCursor(rst, rsc, rsh);	; // remove a statement at Line 188	0	0

```
/* gtkspell/gtkspell.c */
727: gtkspell_detach(GtkSpell *spell) {
...
738:  g_object_set_data(G_OBJECT(spell->view),
        GTKSPELL_OBJECT_KEY,NULL);
739:  gtkspell_free(spell);
740: }
```

Detailed description of Bug5 is as follows. The TextView class of Java-gnome creates a text editor by creating a native peer GtkTextView object. The TextView class may contain a Spell object that provides a spell-checking feature by creating a native peer GtkSpell object. In such case, Java-gnome deallocates the GtkSpell object by calling gtkspell_detach when the corresponding GtkTextView object is deallocated. Also, when a Spell object is reclaimed, the Spell.finalize method calls Spell.release method which eventually calls gtkspell_detach to deallocate the GtkSpell object of the Spell object. Thus, a segmentation fault occurs when JVM garbage collector reclaims a TextView object (and consequently deallocating GtkTextView and GtkSpell objects), and then the Spell object contained in the TextView object.

The bug fix removes Line 57 in the release method to avoid the failure:

```
/* Spell.java */
31: public class Spell {
...
56:  protected void release() {
57:   GtkSpell.detach(this) ;
```

Although the fix looks simple, analyzing the buggy statement is challenging because the execution path involves complicated features such as garbage collection, finalization, and reference counting memory management in the external library execution (e.g., glib signal mechanism).

6.2. Detailed Experiment Result

We make one failing test case that reveals Bug5 based on the bug report, and used 183 passing test cases in the Java-gnome regression test suite (revision 659). Our test environment triggers garbage collection at the end of test runs to trigger finalization activities for reclaimed Java objects. To handle the non-deterministic behaviors of garbage collection, we repeatedly execute the failing test case 3 times per mutant, and our test oracle reports that a test run fails if at least one execution with the test case fails.

Table 5 presents the four most suspicious statements. Line 57 of Spell.java gets the highest suspiciousness score. The mutant at Line 57 is identical to the bug fix. The other seven statements have the same suspiciousness score because the mutants of these statements also deactivate gtkspell_detach in the Java finalization context. For example, Line 68 of Pointer.java and Line 42 of Proxy.java (the third and the fourth rows of Table 5) belong to the call sequence from Spell.finalize to gtkspell_detach; the mutation at Line 48 of GtkSpell.java changes the GtkSpell.detach method not to call gtkspell_detach.

7. Case Study of Debugging Open Bug in Eclipse SWT (Bug8)

This section demonstrates how MUSEUM supports debugging open bugs in real-world software projects. Specifically, our qualitative evaluation demonstrates how to utilize partial fix mutants and the suspicious rankings in diagnosing the cause of bug and suggesting a bug patch. Note that an Eclipse maintainer acknowledged our debugging analysis and patch posted at the Eclipse Bugzilla [9].

7.1. Methodology

Bug Description. Bug 419729 (Bug8) in the Eclipse bug repository for Standard Widget Toolkit (SWT) was reported first on October 17, 2013, and it was open and unresolved since this case study. This bug is chosen for the case study because it appears to be critical for developers and nontrivial to diagnose. First, this bug crashes a JVM and the “Importance” field of the report is marked as “P3 critical” based on the votes by more than dozens of

Table 5: Four most suspicious statements of Java-gnome r-695 (Bug5)

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

8	0.020	`/*Spell.java:57*/`
		`GtkSpell.detach(this);`
			`; //the statement is removed.`	1	0

8	0.020	`/*Pointer.java:68*/`
		`release();`
			`; //the statement is removed.`	1	0

8	0.020	`/*Proxy.java:42*/`
		`super.finalize();`
			`; //the statement is removed.`	1	0

8	0.020	`/*GtkSpell.java:48*/`
		`if (self == null){`
			`if(self != null){`	1	0

developers. Second, this bug is difficult to debug because this bug had not been resolved for more than 22 months (at the time when this case study begins). Bug 419729 is related to the Eclipse SWT module, especially to a subcomponent that binds the SWT interface with the Ubuntu Unity graphics library.

Participants. Two graduate students with little background on the target project (i.e., Eclipse SWT and Ubuntu Unity) used debugging tools, diagnosed the causes of bugs, produced bug fixing patches, and reported their analysis to the bug report database.

Debugging Process. First, MUSEUM was run to identify a suspected bug location and obtain a partial fix mutant that makes the failing test case pass. Based on these results, the participants refined the partial fix mutant into a complete patch for the failure.

Debugging Tools. MUSEUM (version 1.3.21) and Blink [21] (version 2.4.0) are used to locate buggy statements, examined the partial fix mutant, and compared the program states after applying these mutants.

7.2. Debugging the Open Bug Using MUSEUM

Our debugging process consists of fault localization, refining a partial fix mutant, validating the refined mutant, and suggesting a bug patch from the refined mutant.

7.2.1. Fault Localization

Bug 419729 triggers a segmentation fault by dereferencing the `NULL` value in the `state_name` variable at Line 921 of `unity-gtk-action-group.c`. This NULL value is assigned to `state_name` by `unity_gtk_action_group_get_state_name` at Line 920.

```
/* unity-gtk-action-group.c */
858: void unity_gtk_action_group_connect_item(
      UnityGtkActionGroup *group,
859:  UnityGtkMenuItem *item) {
      ...
920:  state_name =
            unity_gtk_action_group_get_state_name(
                                         group,item);
921:  g_hash_table_insert(action->items_by_name,
                       state_name, g_object_ref(item));
```

To reproduce this bug and localize the buggy statements, one failing test case is created based on the bug report. Since the original code snippet in the bug report is not a fully self-contained automated test case, the following two features are added to the original code snippet. First, the user scenario (e.g., mouse-click) in the bug report is encoded as automatic GUI events to eliminate human interaction at the test case executions. Second, the test case is maded to fail when any GUI event in the user scenario is not activated at the test case execution. Also, 203 passing test cases related to the Eclipse SWT are selected from whole Eclipse regression test suite.

MUSEUM generated 14,490 mutants on the 3,855 out of the 4,998 target source lines covered by the failing test case. Only one mutant makes the failing test case pass (i.e., a partial fix mutant) and the 8,766 mutants make some of the 203 passing test cases fail. MUSEUM generated the partial fix mutant by mutating Line 39339 of `os.c` and reported that line the as most suspicious one:

```
/* os.c */
38334: jlong Java_gtk_radio_menu_item_with_label(...,
                                     jbyteArray arg1) {
        ...
39339:   if ((lparg1=(*env)->GetByteArrayElements(env,
            arg1,NULL))==0)
39340:     goto fail;
39341:   rc = gtk_radio_menu_item_with_label(...,
              lparg1) ;
```

Line 39339 calls the JNI function `GetByteArrayElements` to copy a Java array indicated by `arg1` into a new native array, and the address of the new array is stored in `lparg1`. If the copy operation successes, the address value in `lparg1` flows into `gtk_radio_menu_with_label` as an argument (Line 39341).

MUSEUM generated the following partial fix mutant at Line 39339 using `Replace-array-elements-with-constants` mutation operator that replaces the `arg1` with a predefined constant byte array `ByteConst`.

```
39339--: if ((lparg1=(*env)->GetByteArrayElements(env,
           arg1, NULL))==0)
39339++: if ((lparg1=(*env)->GetByteArrayElements(env,
           ByteConst, NULL))==0)
```

This mutation changes the flow of values such that the `NULL` value at the failure site (i.e., Line 921 of `unity-gtk-action-group.c`) with the failing test case is replaced with a pointer to a C string derived from `ByteConst`. This mutation does not change the results of the passing test cases.

7.2.2. Refining the Partial Fix Mutant with Failure-inducing Condition

The participants manually refine the partial fix mutant by figuring out a *failure-inducing condition* and applying the partial fix only when the condition is true (i.e., the partial fix mutant is refined to execute the mutated source line (i.e., 39339++) if the condition holds; the original source line (i.e., 39339–) otherwise).

To identify the failure-inducing condition, the participants monitored and compared the program states at Line 39339 when running both failing and passing test cases. In the failing executing, the byte array pointed by `lparg1` has its first element as `’\0’` while the first element in the passing executions is not `’\0’`. Thus, the participants guess that the failure-inducing condition is `lparg1[0]==’\0’`. Using this condition, the participants refine the partial fix mutant into the following one:

```
if (lparg1[0] == ’\0’)
  lparg1=(*env)->GetByteArrayElements(env,ByteConst,
                                     NULL);
else
  lparg1=(*env)->GetByteArrayElements(env,arg1,NULL);
```

7.2.3. Validating the Refined Partial Fix Mutant

The participants validate the refined partial fix mutant by checking if the obtained failure-inducing condition (i.e.,`lparg1[0]==’\0’`) is general to trigger the failure. For that purpose, the participants compared the execution paths of the original program (i.e., failing execution path) and the refined mutant (i.e., passing execution path) with the same failing test case because the participants guess that the diversing point between the two execution paths indicates the general condition to trigger the failure. The participants found that the these executions diverse at Line 766 of `unity-gtk-action-group.c` in the following code snippet:

```
/* unity-gtk-action-group.c */
753: static gchar *
754: unity_gtk_action_group_get_state_name(
       UnityGtkActionGroup *group,
755:   UnityGtkMenuItem    *item) {

756:   gchar  *name = NULL ;
    ...
765:   gchar *label =
         unity_gtk_menu_item_get_label(item) ;
766:   if (label != NULL && label[0] != ’\0’) {
    ...
800:   else {
    ...}
854:  return name ; }
```

After code review, the participants found that the then-branch of Line 766 never makes `name` as NULL, which makes `unity_gtk_action_group_get_state_name` return non-NULL-value and avoids the segmentation fault at Line 921. But the else-branch can assign NULL to `name`. With the failing test case, the original program execution takes the else-branch while the refined mutant execution takes the then-branch.

As the branch decision at Line 766 depends on `label`, the `label` values are monitored in the aforementioned two executions (i.e., the executions on the original program and the refined fixing mutant with the failing test case) and the executions with all passing test cases that cover Line 766. The monitoring result shows that, in every test case execution, the array pointed by `label` at Line 766 has the same value as the array pointed by `lparg1` at Line 39339 of `os.c`. For the failing execution, `lparg1[0]` at Line 39339 has `’\0’` value. Meanwhile, in the passing executions, the array pointed by `lparg1` has a non-NULL-value and avoids the crash. Thus, the participants conclude that the failure-inducing condition `lparg1[0]==’\0’` is general to trigger the failure and the refined partial fix mutant can fix Bug8.

7.2.4. Suggesting a Bug Fixing Patch

Finally, the participants revised the refined mutant and designed a bug fixing patch. For readability, instead of modifying the second argument of `GetByteArrayElement`, the participants replaced the byte array `lparg1` given to `gtk_radio_menu_item_with_label` with `" "` (a string literal containing one space character) if `lparg1[0] == ’\0’`. We posted our analysis on the fault and the following patch to the Eclipse Bugzilla and an Eclipse maintainer acknowledged our analysis and patch [9]:

```
39339: if (lparg1=(*env)->GetByteArrayElements(env,
         arg1,NULL)==0)
39340:   goto fail;
+++    if (lparg1[0] == ’\0’)
+++      rc=gtk_radio_menu_item_with_label(..., " ");
+++    else
39341:   rc=gtk_radio_menu_item_with_label(...,lparg1);
```

8. Selective Mutation Analyses for Runtime Cost Reduction

8.1. Overview

Although MUSEUM consumes modest amount of time to localize a fault accurately (i.e., 112.6 minutes using 30 machines on average over the eight bugs (Table 2)), we can reduce the runtime cost further with marginal accuracy loss by carefully selecting mutants and test cases to utilize. Also, by selecting mutants and test cases in various ways, we can control the time cost of fault localization, which is desirable for real-world projects where testing/debugging time budget is tightly given.

We present selective use of mutants and test cases and report the effects of various selection strategies on the accuracy and the cost of fault localization. We have designed total 184 selection strategies based on how to select mutants (23) and how to select test cases (8) and their combinations. If a selection strategy involves randomness, we repeated the selection 30 times to obtain statistical confidences of the result. We found that, with selected mutants and test cases, MUSEUM can reduce 96% of the time cost for the eight target programs on average (see Table 10) while still locating the buggy statements as the most suspicious statements.

There exist related works that selectively use mutation operators to reduce computational cost of mutation-based fault localization. Papadakis *et al.* [36] present a mutation-based fault localization tool that uses a small number of mutation operators to avoid heavy cost of mutant executions. Subsequently, Papadakis and Le Traon [38] suggest four sets of selected mutation operators, based on their empirical study of different mutation operator uses and the fault localization results. While the earlier work concentrated on selecting mutation operators, our study explores different chances of selective mutation analyses. For example, our study uses different test case selection criteria and their combinations with new mutant selection criteria.

8.2. Selection Strategies

We have examined 184 (=23×8) selection strategies based on the 23 mutant selection strategies (Section 8.2.1) and the eight test case selection strategies (Section 8.2.2).

8.2.1. Mutant Selection Strategies

We have developed total 23 mutant selection strategies based on the following four criteria where MR(x) and MP(p) are from the existing mutation testing research [35, 12, 34] while MS(n) and MPS(p, n) are developed by the authors:

- **MR(x)**: this strategy randomly selects x% of all generated mutants [35] where $x \in \{10, 20, 30\}$.
- **MS(n)**: it randomly selects n mutants per target line where $n \in \{1, 2, 3\}$. If a target line has only m mutants ($m < n$), MS(n) selects m mutants.
- **MP(p)**: it selects the mutants generated by a mutation operator p in the three sets of mutation operators (i.e., SD, CR, and SM) and the set that includes all mutation operators of the three sets:
  - **MP(SD)**: it uses the statement deletion mutation operator [12] together with the 15 new mutation operators for multilingual behavior.
  - **MP(CR)**: it uses the constant replacement mutation operators [35] together with the 15 new mutation operators for multilingual behavior.
  - **MP(SM)**: it uses the five mutation operators [34] (i.e., 'replace a constant value with its absolute value', 'replace an arithmetic operator with another arithmetic operator', 'change a logical connector', 'change a relational operator', and 'insert an unary operator') with the 15 new mutation operators. Offutt *et al.* [34] claim that mutants generated by these five mutation operators are consistent with the mutants generated by more mutation operators.
  - **MP(All)**: it uses all mutants selected by MP(SD), MP(CR), and MP(SM).
- **MPS(p,n)**: this strategy is a combined strategy of MP(p) and MS(n). Among the mutants selected by MP(p), MPS(p,n) randomly selects n mutants per a target line. If a target line has only m mutants selected by MP(p) ($m < n$), MPS(p,n) randomly selects more mutants generated by other mutation operators to make the target line has n mutants. In this study, we used 12 strategies by combining p = {SD, CR, SM, All} and $n = \{1, 2, 3\}$.
- **MA**: it selects all generated mutants.

8.2.2. Test Case Selection Strategies

We have developed eight test case selection strategies based on the random selection and coverage based selection as motivated by the test case selection work [14]. All test case selection strategies select the failing test case in the test suite.

- **TR(x)**: it randomly selects x% of the passing test cases where $x \in \{10, 20, 30\}$.
- **TC(x)**: it selects x% of the passing test cases that achieve high coverage of the target lines (i.e., the source code lines covered by the failing test case) where $x \in \{10, 20, 30\}$. TC(x) uses a greedy algorithm which repeats to select a passing test case that covers a largest number of uncovered target lines. If there are multiple such passing test cases, the algorithm selects one among the choices.
- **TM**: it selects a small number of passing test cases that cover all target lines. TM uses a greedy algorithm which repeats to select a test case that covers a largest number of uncovered target lines until the selected test cases cover all target lines (the algorithm stops selection when no passing test case can increase the coverage).
- **TA**: TA uses all given passing test cases.

8.2.3. Reduction in Mutants and Test Cases

Table 6 shows that our mutant selection strategies except MA reduce the generated mutants. Each entry reports the ratio of the number of the selected mutants to the

Table 6: Ratio of the number of the selected mutants to the number of all mutants (%)

Strategy	Bug1	Bug2	Bug3	Bug4	Bug5	Bug6	Bug7	Bug8	Average
MR(10)	10.0	10.0	10.0	9.8	10.0	9.9	10.0	10.0	10.0
MR(20)	20.0	20.0	19.9	19.8	19.4	20.0	20.0	20.0	19.9
MR(30)	30.0	30.0	29.9	30.1	30.1	30.0	30.1	30.0	30.0
MS(1)	55.5	31.8	32.7	27.1	29.4	25.0	29.7	25.6	32.1
MS(2)	76.9	55.1	56.3	46.8	50.0	44.7	53.6	44.9	53.5
MS(3)	89.2	68.8	69.7	60.1	63.0	60.5	69.1	60.3	67.6
MP(SD)	10.4	22.9	21.2	26.1	28.6	20.0	24.9	22.3	22.1
MP(CR)	56.0	38.5	35.3	19.3	20.9	30.0	35.7	33.6	33.7
MP(SM)	18.5	21.3	20.0	13.7	15.9	11.9	21.5	16.1	17.4
MP(All)	74.7	56.3	53.6	43.1	45.7	51.0	53.5	55.8	54.2
MPS(SD,1)	57.6	41.2	40.8	35.1	39.2	29.5	40.8	33.2	39.7
MPS(SD,2)	77.5	62.3	62.5	52.8	57.4	46.9	61.9	49.4	58.8
MPS(SD,3)	89.3	74.2	74.3	64.2	68.1	61.7	74.8	63.2	71.2
MPS(CR,1)	66.2	49.9	47.6	36.8	40.2	39.1	46.2	41.8	46.0
MPS(CR,2)	79.5	65.7	65.1	52.3	56.7	52.0	63.1	53.4	61.0
MPS(CR,3)	90.3	75.3	74.9	62.9	66.6	63.4	74.5	64.3	71.5
MPS(SM,1)	60.9	40.4	40.3	33.1	36.9	29.7	38.5	33.2	39.1
MPS(SM,2)	78.4	60.7	61.2	50.8	55.1	46.7	59.3	48.7	57.6
MPS(SM,3)	89.8	72.5	72.8	62.9	66.6	61.2	72.5	62.0	70.0
MPS(All,1)	77.6	60.0	57.9	48.1	51.5	51.7	56.6	56.7	57.5
MPS(All,2)	84.9	70.7	69.5	58.1	62.5	58.2	67.5	61.8	66.7
MPS(All,3)	92.7	79.0	77.9	66.4	70.5	66.3	77.9	68.7	74.9

number of all generated mutants. For example, MP(All) selects 2,137 mutants (= 2,861 mutants × 74.7%) for the target code of Bug1 (see the second column of the 11th row of the table).

Table 7 shows that our test case selection strategies except TA reduce test cases significantly while reducing target line coverage modestly. Table 7(a) presents the ratio of the reduced test set size to the original test set size. For example, TM selects 2.3 test cases (=150×1.5%) for Bug2 on average (see the third column of the eighth row of Table 7(a)). TM selects less test cases than TR(x) and TC(x) for all bugs except Bug1 with x = 10 or 20 and Bug6 with x = 10. Table 7(b) presents the target line coverage achieved by the passing test cases selected by the test case selection strategies. For example, TR(20) covers the 96% of the target lines for Bug1 on average (see the second column of the third row of Table 7(b)). TC(x) achieves the highest target line coverage in all cases except TC(10) on Bug1. TM also achieves the highest coverage with the smallest number of selected test cases among the all strategies that achieve the highest coverage for all target programs (see Table 6(a)). The test case selection strategies do not achieve the 100% coverage if a target line is not covered by any passing test case.

8.3. *Effects of the Selection Strategies on Fault Localization*

8.3.1. *Effect on the Fault Localization Accuracy*

Table 8 shows how much the ranking of the faulty line improves with the selection strategies. Table 8(a) presents the improved ranking of the faulty line with the mutant selection strategies (except MA) with all test cases. Table 8(b) presents the improved ranking of the faulty line with the test case selection strategies (except TA) with all mutants. Table 8(c) presents the improved ranking with 12 combined strategies of the four mutant selection strategies (i.e., MPS(p,1) with $p \in \{\mathrm{SD}, \mathrm{CR}, \mathrm{SM}, \mathrm{All}\}$) and the three test case selection strategies (i.e., TR(10), TC(10), and TM). Note that 0 in the table indicates that the ranking of the faulty statement does not change with a given selection strategy (i.e., keeping the same fault localization accuracy).

Table 8(a) shows that all 12 MPS strategies do not improve the ranking of the faulty statement in all target bugs except Bug5, Bug6 and Bug7. Note that even for Bug5, Bug6, and Bug7, MUSEUM still reports the faulty line as the most suspicious one (i.e., MPS increases the number of the most suspicious lines whose suspiciousness scores are all equal to that of the faulty statement). For example, MUSEUM with MPS(SD,1) reports the suspiciousness ranking of the faulty statement in Bug6 as 5.0 (=3+2.0) on average, but still reports the faulty statement as the most suspicious one with other 4.0 statements. However, the other selection strategies in Table 8(a) improves the ranking significantly. For example, MR and MS improve the ranking at least by 853.6 and 148.6 on average.

Table 8(b) shows that the test case selection with all mutants do not improve the ranking of the faulty statement in all target bugs except Bug2 and Bug5 (these strategies still report the faulty statement of Bug2 and Bug5 as the most suspicious statement with other statements in a tie).

Table 8(c) presents the improved ranking of the faulty statements with the 12 balanced combinations of the four

Table 7: Results of the test cases selection strategies

(a) Ratio of the number of the selected test cases to the number of all test cases (%)

Strategy	Bug1	Bug2	Bug3	Bug4	Bug5	Bug6	Bug7	Bug8	Average
TR(10)	12.9	10.0	11.0	11.9	11.0	11.0	9.8	10.9	10.1
TR(20)	25.2	19.7	18.9	20.3	22.0	24.7	20.9	20.0	20.1
TR(30)	38.8	29.1	31.3	32.3	32.0	31.6	30.9	30.7	30.0
TC(10)	12.9	10.2	10.6	13.9	12.4	13.3	10.6	11.4	10.1
TC(20)	25.9	20.4	20.6	23.6	21.5	23.0	20.8	21.2	20.1
TC(30)	38.3	30.2	30.5	32.3	31.1	32.5	30.8	30.8	30.0
TM	25.9	1.5	2.4	6.0	3.7	13.3	3.6	7.6	3.3

(b) Target line coverage achieved by the passing test cases selected by the selection strategies (%)

Strategy	Bug1	Bug2	Bug3	Bug4	Bug5	Bug6	Bug7	Bug8	Average
TR(10)	0	100	100	55	69	50	93	86	69.1
TR(20)	96	100	98	55	92	81	93	88	87.9
TR(30)	100	100	100	90	94	82	93	95	94.3
TC(10)	0	100	100	90	94	82	93	99	82.3
TC(20)	100	100	100	90	94	82	93	99	94.8
TC(30)	100	100	100	90	94	82	93	99	94.8
TM	100	100	100	90	94	82	93	99	94.8

MPS(p,1) strategies where $p \in \{\text{SD}, \text{CR}, \text{SM}, \text{All}\}$ and the three test case selection strategies TR(10), TC(10) and TM. These 12 selection strategies improve the ranking by 1.3 on average over all eight target programs. More importantly, these 12 strategies still report the faulty statement as the most suspicious statement with other statements in a tie. For Bug5 and Bug6, the improved ranking is larger than the other target programs because the number of mutants that change the test case execution results is reduced significantly for Bug5 and Bug6. For example, the MPS(SD,1) and TM selection strategy decreases the number of mutants that make the failing test case pass from 51 to 26 for Bug5 and from 32 to 7 for Bug6; consequently, more lines have the same numbers of the fail-to-pass mutant executions and pass-to-fail mutant executions after the mutant and test case selections. For the other six mutants, the number of mutants that make the failing test case pass is decreased by 0 to 2. We do not present the results of the other selection strategies because they are worse than these 12 presented strategies. For example, as shown in Table 8(a), MR, MP, and MS degrade the fault localization accuracy significantly. We do not present MPS(p,2) and MPS(p,3) because they are similar to MPS(p,1) in terms of the accuracy but they select much more mutants than MPS(p,1) (Table 6). For the similar reason, we present the results with TR(10), TC(10) and TM, not the other test case selection strategies.

8.3.2. Effect on the Fault Localization Time Cost

Table 9 presents the ratio of the number of the selective mutant executions (i.e., the number of all pairs (m_i, t_{ij}) where m_i is a selected mutant and t_{ij} is a selected test case that covers the mutated line of m_i) to that of the full mutant executions (i.e., mutant executions with all mutants and all test cases). The 12 strategies execute only 3.5%–6.8% of the full mutant executions for the eight target bugs on average. MPS(SD,1) with TM executes the smallest number of mutant executions on average (i.e., MPS(SD,1) with TM removes 99.4% (=100%-0.6%) of the full mutant executions for Bug2).

Figure 2 visualizes the accuracy-cost trade-offs in the 12 selection strategies. The x axis represents the average ratio of the cost of the selective mutant testing to the full mutant testing. The y axis represents the average ranking improvement of the faulty statement. Each data point represents the cost and accuracy of a selection strategy. For example, MPS(SD,1) with TM reduces the number of the mutant executions to 3.5% of the full mutant executions on average (the last column of the fourth row of Table 9) and improvements the ranking by 1.4 on average (the last column of the fourth row of Table 8(c)). In general, more mutant testing achieves higher accuracy. For example, MPS(SD,1) with TC(10) is represented by '×' located at x=4.7 and y=1.1, which indicates that MPS(SD,1) with TC(10) executes more mutant testing than MPS(SD,1) with TM (4.7% v.s. 3.5%) but it improves the ranking less than MPS(SD,1) with TM (1.1 v.s. 1.4). Note that these 12 strategies achieve both high accuracy (i.e., the average ranking improvement is less than 2.0) and high cost reduction (i.e., the number of the selective mutant executions is reduced to less than 7% of the full mutant executions).

Finally, Table 10 shows the overall time cost of the fault localization with all mutants and all test cases (the second

Table 8: Ranking improvements of the faulty statements with various selection strategies

(a) Strategies that reduce the mutants only

Strategy	Bug1	Bug2	Bug3	Bug4	Bug5	Bug6	Bug7	Bug8	Average
MR(10)	1,590.7	218.5	340.2	140.3	135.3	2,867.8	207.6	3,760.8	1,157.7
MR(20)	1,571.3	192.0	232.7	97.5	112.6	1,968.3	110.5	3,226.0	938.9
MR(30)	1,368.5	166.8	201.0	62.2	82.6	2,039.8	94.5	2,813.4	853.6
MS(1)	1,141.0	0.0	0.0	103.2	1.9	2.2	118.0	2,257.8	453.0
MS(2)	824.9	0.0	0.0	85.6	0.4	0.4	60.7	1,879.3	356.4
MS(3)	409.3	0.0	0.0	73.8	0.0	0.0	32.0	673.4	148.6
MP(SD)	0.0	238.0	367.0	0.0	-1.0	0.0	0.0	0.0	75.5
MP(CR)	0.0	-1.0	0.0	0.0	153.0	2,960.0	1.0	0.0	389.1
MP(SM)	0.0	244.0	358.0	0.0	162.0	3,213.0	1.0	0.0	497.3
MP(All)	0.0	-1.0	0.0	0.0	1.0	2.0	0.0	0.0	0.3
MPS(SD,1)	0.0	0.0	0.0	0.0	2.7	2.0	0.0	0.0	0.6
MPS(SD,2)	0.0	0.0	0.0	0.0	0.3	1.0	0.0	0.0	0.2
MPS(SD,3)	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.0
MPS(CR,1)	0.0	0.0	0.0	0.0	0.6	3.3	1.0	0.0	0.6
MPS(CR,2)	0.0	0.0	0.0	0.0	0.0	0.9	1.0	0.0	0.2
MPS(CR,3)	0.0	0.0	0.0	0.0	0.0	0.1	1.0	0.0	0.1
MPS(SM,1)	0.0	0.0	0.0	0.0	2.1	2.2	1.0	0.0	0.7
MPS(SM,2)	0.0	0.0	0.0	0.0	0.4	0.7	1.0	0.0	0.3
MPS(SM,3)	0.0	0.0	0.0	0.0	0.0	0.2	1.0	0.0	0.2
MPS(All,1)	0.0	0.0	0.0	0.0	1.0	2.0	0.0	0.0	0.4
MPS(All,2)	0.0	0.0	0.0	0.0	0.0	1.4	0.0	0.0	0.2
MPS(All,3)	0.0	0.0	0.0	0.0	0.0	0.4	0.0	0.0	0.1

(b) Strategies that reduce the test cases only

Strategy	Bug1	Bug2	Bug3	Bug4	Bug5	Bug6	Bug7	Bug8	Average
TR(10)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
TR(20)	0.0	0.0	0.0	0.0	3.0	0.0	0.0	0.0	0.4
TR(30)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
TC(10)	0.0	0.0	0.0	0.0	1.8	0.0	0.0	0.0	0.2
TC(20)	0.0	0.0	0.0	0.0	1.6	0.0	0.0	0.0	0.2
TC(30)	0.0	0.0	0.0	0.0	1.1	0.0	0.0	0.0	0.1
TM	0.0	0.4	0.0	0.0	3.0	0.0	0.0	0.0	0.4

(c) Strategies that reduce both mutants and test cases

Strategy		Bug1	Bug2	Bug3	Bug4	Bug5	Bug6	Bug7	Bug8	Average
Mutant	Test case									
MPS(SD,1)	TR(10)	0.0	0.0	0.2	0.0	8.7	2.2	0.3	0.0	1.4
MPS(SD,1)	TC(10)	0.0	0.0	0.0	0.0	6.6	2.1	0.0	0.0	1.1
MPS(SD,1)	TM	0.0	0.5	0.0	0.0	8.5	2.2	0.0	0.0	1.4
MPS(CR,1)	TR(10)	0.0	0.0	0.1	0.0	7.6	3.6	1.0	0.0	1.5
MPS(CR,1)	TC(10)	0.0	0.0	0.0	0.0	4.4	3.3	1.0	0.0	1.1
MPS(CR,1)	TM	0.0	0.4	0.0	0.0	5.0	3.4	1.0	0.0	1.2
MPS(SM,1)	TR(10)	0.0	0.1	0.0	0.0	9.2	1.9	1.0	0.0	1.5
MPS(SM,1)	TC(10)	0.0	0.0	0.0	0.0	6.6	1.9	1.0	0.0	1.2
MPS(SM,1)	TM	0.0	0.4	0.0	0.0	7.8	1.9	1.0	0.0	1.4
MPS(All,1)	TR(10)	0.0	0.0	0.2	0.0	7.6	2.0	0.0	0.0	1.2
MPS(All,1)	TC(10)	0.0	0.0	0.0	0.0	5.0	2.0	0.0	0.0	0.9
MPS(All,1)	TM	0.0	0.3	0.0	0.0	6.0	2.0	0.0	0.0	1.0

row) and that of the fault localization with MPS(SD,1) and TM (the third row) on one machine. The numbers in the second row are calculated by multiplying 30 to the time cost in Table 2. MUSEUM with the MPS(SD,1) and TM selection strategies consumes only 3.8% of the time cost with all mutants and all test cases for the eight target bugs on average (see the last column of the last row). Thus, this result confirms that the selection strategy can effectively reduce the time cost of MUSEUM as the number of the mutant executions is reduced. [5]

9. Discussions

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

For large real-world programs, it is challenging to build test cases that exercise diverse execution paths because

[5] The ratio in Table 10 can be different from the ratio in Table 6 because the time cost of MUSEUM involves mutant generations, data processing and other operational steps in addition to mutant executions (also execution time of a mutant can be different depending on the mutant and the test case used).

Table 9: Ratio of the number of the selective mutant executions to the full mutant executions (%)

Strategy		Bug1	Bug2	Bug3	Bug4	Bug5	Bug6	Bug7	Bug8	Average
Mutant	Test case									
MPS(SD,1)	TR(10)	7.4	4.2	4.2	3.9	4.4	3.6	4.3	3.6	4.5
MPS(SD,1)	TC(10)	7.4	4.1	4.3	4.4	4.7	4.2	4.4	3.8	4.7
MPS(SD,1)	TM	14.8	0.6	1.0	1.6	1.4	4.2	1.5	2.5	3.5
MPS(CR,1)	TR(10)	8.5	5.0	4.9	4.3	4.5	4.6	4.8	4.5	5.1
MPS(CR,1)	TC(10)	8.5	5.0	5.0	4.8	4.8	5.3	5.0	4.8	5.4
MPS(CR,1)	TM	17.0	0.7	1.1	1.8	1.4	5.3	1.6	3.2	4.0
MPS(SM,1)	TR(10)	7.8	4.1	4.1	3.8	4.2	3.6	4.0	3.6	4.4
MPS(SM,1)	TC(10)	7.8	4.2	4.1	4.1	4.5	4.2	4.2	3.8	4.6
MPS(SM,1)	TM	15.7	0.6	1.0	1.5	1.3	4.2	1.4	2.6	3.5
MPS(All,1)	TR(10)	10.0	6.0	5.8	5.6	5.7	6.2	5.9	6.1	6.4
MPS(All,1)	TC(10)	10.0	6.0	6.1	6.3	6.3	7.1	6.0	6.6	6.8
MPS(All,1)	TM	20.0	0.9	1.4	2.5	1.8	7.1	2.0	4.4	6.5

Table 10: Overall time cost of fault localization (in minutes)

	Bug1	Bug2	Bug3	Bug4	Bug5	Bug6	Bug7	Bug8	Average
MUSEUM with all mutants and all test cases	360	1,785	1,346	738	682	5,262	1,501	15,334	3,376.0
MUSEUM with MPS(SD,1) and TM	29	21	34	10	10	186	63	1,166	189.9
Ratio	8.1%	1.2%	2.5%	1.4%	1.5%	3.5%	4.2%	7.6%	3.8%

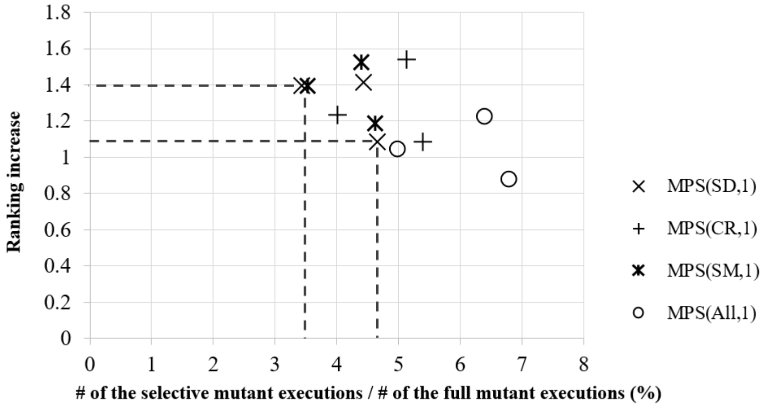

Figure 2: Ranking improvement of the faulty statements and the ratio of the number of selective mutant executions to the full mutant executions

it is non-trivial to understand and control a target program. Also, generating diverse test cases for multilingual programs has additional burden to learn and satisfy safety rules on language interfaces. Thus, multilingual programs are often developed with only simple test cases, which makes the SBFL techniques fail to accurately localize the eight real-world multilingual bugs (Table 3).

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, Bug6, and Bug8 are around 1%, 22%, 24%, 19%, and 11% and the accuracy of the SBFL techniques for these bugs are very low (Table 3). In contrast, MUSEUM can alleviate 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.

9.2. Effectiveness of the New Mutation Operators for Localizing Multilingual Bugs

The experiment results show that the new mutation operators are effective to generate informative mutants (i.e., partial fix mutants) to localize multilingual bugs. For Bug1, Bug4, and Bug8, only the new mutation operators generate partial fix mutants. For Bug5 and Bug7, the new mutation operators and the existing ones generate partial fix mutants. For Bug2, Bug3 and Bug6, only existing mutation operators generate partial fix mutants.

To assess the impact of the new mutation operators on fault localization, we ran MUSEUM for Bug1, Bug4, Bug8, Bug5 and Bug7 without the new mutation operators. For

Bug1, Bug4 and Bug8, the suspiciousness ranking of the faulty line becomes significantly low (1,737 (89.6%) for Bug1, 117 (62.9%) for Bug4, and 3,061 (61.2%) for Bug8). For Bug5, the ranking of the faulty line changes from the eighth to the ninth and the faulty line is not anymore the most suspicious statement. For Bug7, the ranking of the faulty line remains unchanged.

9.3. High Accuracy with Low Runtime Cost through Selective Mutation Analysis

The selective mutation analysis for MUSEUM can achieve high fault localization accuracy with significantly reduced runtime cost (e.g., MPS(SD,1) and TM can reduce the runtime cost up to 96% and identifies the faulty statements as the most suspicious ones) (Section 8.3.1). Also, more mutants and test cases can increase the fault localization accuracy with the selective mutation analysis (Figure 2).

Thus, MUSEUM should start with the mutants and test cases selected by a selection strategy (e.g., MPS(SD,1) and TM). Then, MUSEUM can add more mutants and test cases by relaxing the parameter of the selection strategy or changing the selection strategy. In this way, MUSEUM can achieve high fault localization accuracy with low runtime cost first and then increase the fault localization accuracy gradually within the given time budget.

10. Conclusion and Future Work

We have presented MUSEUM which localizes bugs in complex real-world multilingual programs in a language semantics agnostic manner through mutation analyses. The experiments and the case studies show that MUSEUM accurately locates the faulty statements for all non-trivial Java/C bugs. Also, we show that the accuracy of fault localization for multilingual programs can be increased by adding new mutation operators relevant with language interface constraints. Finally, our selection strategies over mutants and test cases significantly reduce the analysis time with marginal accuracy loss.

As future work, we will add more mutation operators targeting multilingual features and higher-order mutation operators to reduce equivalent mutants and generate useful mutants. Also, we will apply MUSEUM to an interactive debugger such as Blink [21] to maximize the debugging effectiveness. Finally, we will investigate how to utilize MUSEUM to improve program repair and search-based program analysis for multilingual programs.

Acknowledgements

This work is supported by the MSIP, Korea, under the ITRC support program (IITP-2016-H85011610120001002) supervised by the IITP, the NRF grants (NRF-2014K1A3A1A09063167, NRF-2015R1C1A1A01055259, NRF-2015R1C1A1A01052876, and NRF-2016R1A2B4008113) funded by the Korea government (MSIP), 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]), the IITP grant funded by the Korea government(MSIP) (No. R0190-16-2012), and the GIST Research Institute (GRI) grant in 2016.

References

[1] PIT v0.33 - mutation testing tool for Java. http://pitest.org.

[2] JDK-4804447: JNI get⟨type⟩arrayelements fail with zero length arrays. https://bugs.openjdk.java.net/browse/JDK-4804447, 2003.

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

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

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

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

[7] 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.

[8] Firefox Bug 958706 - Don't hide JNI exceptions. https://bugzilla.mozilla.org/show_bug.cgi?id=958706, 2014.

[9] Eclipse SWT bug419729: Native crash in org.eclipse.swt.internal.gtk.OS._gtk_widget_show. https://bugs.eclipse.org/bugs/show_bug.cgi?id=419729, 2015.

[10] JNI APIs and developer guides. http://docs.oracle.com/javase/8/docs/technotes/guides/jni, 2015.

[11] 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.

[12] L. Deng, J. Offutt, and N. Li. Empirical evaluation of the statement deletion mutation operator. In *ICST*, 2013.

[13] M. Furr and J. S. Foster. Checking type safety of foreign function calls. *ACM Trans. Prog. Lang. Syst.*, 30(4):1–63, 2008.

[14] T. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In *ICSE*, 1998.

[15] S. Hong, B. Lee, T. Kwak, Y. Jeon, B. Ko, Y. Kim, and M. Kim. Mutation-based fault localization for real-world multilingual programs. In *ASE*, 2015.

[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.Clause and A.Orso. LEAKPOINT: Pinpointing the causes of memory leaks. In *ICSE*, 2010.

[18] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *ASE*, 2005.

[19] C. Jung, S. Lee, E. Raman, and S. Pande. Automated memory leak detection for production use. In *ICSE*, 2014.

[20] G. Kondoh and T. Onodera. Finding bugs in Java Native Interface programs. In *ISSTA*, 2008.

[21] B. Lee, M. Hirzel, R. Grimm, and K. S. McKinley. Debugging mixed-environment programs with Blink. *Software: Practice and Experience*, 45(9):1277–1306, 2014.

[22] B. Lee, B. Wiedermann, M. Hirzel, R. Grimm, and K. McKinley. Jinn: Synthesizing dynamic bug detectors for foreign language interfaces. In *PLDI*, 2010.

[23] S. Li and G. Tan. Finding bugs in exceptional situations of JNI programs. In *CCS*, 2009.

[24] S. Li and G. Tan. JET: exception checking in the java native interface. In *OOPSLA*, 2011.

[25] S. Li and G. Tan. Finding reference-counting errors in Python/C programs with affine analysis. In *ECOOP*, 2014.

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

[27] 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.* Kluwer Academic Publishers, 2001.

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

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

[30] L. A. Meyerovich and A. S. Rabkin. Empirical analysis of programming language adoption. In *OOPSLA*, 2013.

[31] S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *ICST*, 2014.

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

[33] 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.

[34] J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *ICSE*, 1993.

[35] P.Ammann, M.E.Delamaro, and J.Offutt. Establishing theoretical minimal sets of mutants. In *ICST*, 2014.

[36] M. Papadakis, M. E. Delamaro, and Y. L. Traon. Proteum/FL: a tool for localizing faults using mutation analysis. In *Proceeding of IEEE International Working Conference on Source Code Analysis and Manipulation*, 2013.

[37] M. Papadakis and Y. Le-Traon. Metallaxis-FL: mutation-based fault localization. *STVR*, 25:605–628, 2015.

[38] M. Papadakis and Y. L. Traon. Effective fault localization via mutation analysis: Selective mutation approach. In *SAC*, 2014.

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

[40] T. Ravitch, S. Jackson, E. Aderhold, and B. Liblit. Automatic generation of library bindings using static analysis. In *PLDI*, 2009.

[41] J. Siefers, G. Tan, and G. Morrisett. Robusta: taming the native beast of the JVM. In *CCS*, 2010.

[42] G. Tan, A. Appel, S. Chakradhar, A. Raghunathan, S. Ravi, and D. Wang. Safe Java Native Interface. In *ISSSE*, 2006.

[43] G. Tan and J. Croft. An empirical security study of the native code in the JDK. In *USENIX SS*, 2008.

[44] G. Tan and G. Morrisett. Ilea: inter-language analysis across Java and C. In *OOPSLA*, 2007.

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

[46] 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 TOSEM*, 22(4):31:1–31:40, 2013.

[47] G. Xu, M. D. Bond, F. Qin, and A. Rountev. LeakChaser: Helping programmers narrow down causes of memory leaks. In *PLDI*, 2011.

[48] G. Xu and A. Rountev. Precise memory leak detection for java software using container profiling. In *ICSE*, 2008.

[49] L. Zhang, L. Zhang, and S. Khurshid. Injecting mechanical faults to localize developer faults for evolving software. In *OOPSLA*, 2013.