석사 학위논문

Master's Thesis

프로그램 변이를 이용한 실제 다중 언어 프로그램 디버깅

Mutation-based Debugging of Real-world Multilingual Programs

곽 태 훈 (郭 太 勳 Kwak, Taehoon)

전산학부

School of Computing

KAIST

2016

프로그램 변이를 이용한 실제 다중 언어 프로그램 디버깅

Mutation-based Debugging of Real-world Multilingual Programs

Mutation-based Debugging of Real-world Multilingual Programs

Advisor : Professor Kim, Moonzoo

by

Kwak, Taehoon
School of Computing
KAIST

A thesis submitted to the faculty of KAIST in partial fulfillment of the requirements for the degree of Master of Science in Engineering in the School of Computing . The study was conducted in accordance with Code of Research Ethics[1].

2015. 12. 14.

Approved by

Professor Kim, Moonzoo

[Advisor]

[1]Declaration of Ethical Conduct in Research: I, as a graduate student of KAIST, hereby declare that I have not committed any acts that may damage the credibility of my research. These include, but are not limited to: falsification, thesis written by someone else, distortion of research findings or plagiarism. I affirm that my thesis contains honest conclusions based on my own careful research under the guidance of my thesis advisor.

프로그램 변이를 이용한 실제 다중 언어 프로그램 디버깅

곽 태 훈

위 논문은 한국과학기술원 석사학위논문으로
학위논문심사위원회에서 심사 통과하였음.

2015년 12월 14일

MCS 20143037

곽 태 훈. Kwak, Taehoon. Mutation-based Debugging of Real-world Multilingual Programs. 프로그램 변이를 이용한 실제 다중 언어 프로그램 디버깅. School of Computing . 2016. 34p. Advisor Prof. Kim, Moonzoo. Text in English.

ABSTRACT

Programmers maintain and evolve their software in a variety of programming languages to take advantage of various control/data abstractions and legacy libraries. The programming language ecosystem has diversified over the last few decades, and non-trivial programs are likely to be written in more than a single language. Unfortunately, language interfaces such as Java Native Interface and Python/C are difficult to use correctly and the scope of fault localization goes beyond language boundaries, which makes debugging multilingual bugs challenging. Moreover, since existing fault localization techniques mostly focus on small-size monolingual programs, the techniques show low accuracy on localizing the fault of large-size multilingual programs.

To overcome the aforementioned limitations, I propose a mutation-based fault localization technique for real-world multilingual programs. The key intuition behind the mutation-based approach is that mutation testing results will be largely different depending on whether mutation is on correct statement or mutation is on faulty statement. To improve the accuracy for locating multilingual bugs, I have developed and applied new mutation operators as well as conventional mutation operators.

The evaluation of the proposed approach is performed on six non-trivial multilingual bugs which are obtained from bug repositories for real-world open-source projects, such as Eclipse SWT, SQLite-jdbc, Java-gnome, and Azureus. The size of target projects is ranging from 6KLOC to 341KLOC. The result is promising in that the proposed technique identifies the faulty statements as the most suspicious statements for all six bugs. Even in the worst case a programmer can locate the faulty statement within 8 statements recommended by the proposed technique. Especially, I describe a detailed experiment result to suggest that mutation-based approach often provides the mutants which can be served as a hint to fix the multilingual bug.

Contents

List of Tables

List of Figures

Chapter 1. Introduction

As software evolves, many software systems are written in multiple programming languages to reuse legacy code and leverage the languages best suited to the developers' needs. Over the last few decades language designers have made various choices in designing the syntax and semantics of their languages. The result is a robust ecosystem where a few languages cover the most use in part due to open source libraries and legacy code while many languages exist for niche uses [37]. This ecosystem is likely to make developers write a *multilingual program* which is a non-trivial program written in more than a single language. High-level languages such as Java, Python, and OCaml provide standard libraries, which typically call legacy code written in low-level languages (e.g., C) to interface with the operating system. A number of projects for the legacy libraries that have evolved for decades provide language bindings for each language. A large scale software project employs a number of libraries written in multiple languages.

Correct multilingual programs are difficult to write in general in part due to the complex language interfaces such as Java Native Interface (JNI) and Python/C, which require the programs to respect a set of thousands of interface safety rules over hundreds of application interface functions [29, 33]. Moreover, once a bug occurs at interactions of code written in different languages, programmers are required to understand the cause-effect chains across language boundaries. Despite the advances of automated testing techniques for complex real-world programs [19, 24–26, 44], debugging multilingual bugs in real-world programs is still challenging and requires significant human effort. For instance, Bug 322222 in the Eclipse bug repository crashes JVMs with a segmentation fault in C as an effect when the program throws an exception in Java as the cause (Chapter 7). Locating and fixing this bug took a heroic debugging effort of more than a year from 2009 to 2010 with hundreds of comments from dozens of programmers before the patch went into Eclipse 3.6.1 in September 2010.

The most time-consuming step in debugging activity is to find the *root cause* of a bug (i.e., fault localization). When a developer faces a program error (e.g., an assertion failure and a rule violation), the developer should spend a lot of time on fault localization before fixing the error. More particularly, as the complexity of a program increases like multilingual programs, the difficulty of fault localization on the program also increases. For example, the Eclipse bug 322222 took 99% of the debugging time to perform fault localization. However, many researchers did not conduct a study on fault localization targeting multilingual bugs.

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

[1]The proposed technique is an extension for multilingual bugs based on MUSE which targets to localize C bugs [38].

The empirical evaluation of six real-world Java/C bugs demonstrates that the proposed technique locates the bugs in non-trivial real-world multilingual programs far more precisely (i.e., the technique identifies the buggy statements as the most suspicious statements for all six bugs) than do the state-of-the-art spectrum based fault localization (SBFL) techniques (Chapter 4). For example, for Bug 322222 in the Eclipse bug repository, the technique indicates the statement at which the developer made a fix as the most suspicious statement among a total of 3482 candidates (Chapter 7).

1.1 Limitations of Previous Approaches

A number of bug detection techniques targeting multilingual program errors [27, 29–32, 45–47, 49] have been proposed. However, the techniques are not effective in debugging, because they can only report certain kinds of safety rule violations; they cannot indicate the *root cause* of the bug, especially when the bug does not explicitly involve any known safety rule violations. Moreover, these bug detectors do not scale well to a large number of languages and various kinds of program errors since they have to deeply analyze the semantics of each language for each kind of bug.

On the other hand, to indicate the *root cause* of a bug, many techniques [11, 22, 38, 51, 54] which is only for fault localization have been suggested. Among them, a well-known approach is spectrum-based fault localization (SBFL) technique which utilizes program spectrum (i.e., execution traces of the program) collected by the coverage measurement tool for the program. SBFL takes source codes and test cases as an input and then reports suspiciousness score of each program element to guide a good starting point to localize the bug.

However, existing SBFL techniques only focus on monolingual program bugs. If a developer applies existing SBFL techniques to multilingual programs, program spectrum written in different language is not collected. Since multilingual bugs may be involved with code fragments of different languages, SBFL techniques show bad precision for multilingual program bugs. Despite such weakness, until now, there is no work to extend SBFL techniques to cover multilingual programs.

In addition, since the accuracy of SBFL techniques depends on the quality of a test suite, SBFL techniques often show low accuracy under the poor quality of a test suite in real-world programs. If SBFL techniques cannot collect diverse execution paths of the program due to the poor quality of the test suite, SBFL techniques are more likely to assign the same suspiciousness score for multiple program elements. A developer does not know which program element is more suspicious among the program elements sharing same suspiciousness score, in the worst case, the developer may inspect all of the program elements in the score group.

In the view point of manual debugger, recently Blink [28] combining Java debugger and C debugger is suggested. However, multilingual bugs often have a long propagation path between the *root cause* of the bug and its symptom, A developer may consume a lot of effort to indicate the buggy line (i.e., *root cause*) without any suggestion of a good starting point to perform quickly debugging the bug.

1.2 Proposed approach

To overcome aforementioned limitations, I suggest a fault localization technique, called MUSEUM (MUtation baSEd fault localization for mUltilingual prograMs), which shows a promising results on real-world multilingual bugs and operates on the basis of two observations regarding the impact analysis of mutants. MUSEUM reports suspiciousness score of program elements precisely in multilingual bugs that can reduce the human effort in debugging activity. In the fault localization field, MUSEUM is a novel approach in terms of targeting multilingual bugs.

MUSEUM extends a mutation based fault localization technique, MUSE [38], which is restricted to target monolingual bugs. To add multilingual features, I have analyzed JNI specification, and the result is 15 new mutation operators that assist to localize the *root cause* of multilingual bugs. MUSEUM finally works together with conventional mutation operators for C, Java and 15 new mutation operators.

1.3 Contribution

This dissertation provide the folliwing contributions:

- New mutation operators targeting multilingual program errors which are highly effective at locating multilingual bugs (Section 3.3)
- Empirical demonstration of the high accuracy of the mutation-based fault localization technique for the six real-world multilingual bugs (Chapter 4)
- Detailed report on three case studies to determine why and how the proposed technique can precisely localize real-world multilingual bugs (Chapter 5, 6 and 7).

1.4 Structure of the dissertation

The remaining dissertation is structured as follows.

Chapter 2 describes the background on multilingual debugging and fault localization techniques. It begins by explaining safety rules in multilingual programs, then introduces existing multilingual bug checkers with static and dynamic approach. Next, I describe fault localization techniques with an illustrative example.

Chapter 3 explains our mutation based fault localization technique for multilingual bug. I first describe a motivating example for the proposed approach. Then, I deliver the details of the proposed approach with new mutation operators.

Chapter 4 provides an overview of the empirical study on the six real-world multilingual bugs. I first explain the experiment setup, then I demonstrate the superior accuracy of the proposed technique through the experiment results.

Chapter 5, 6, and **7** describe three case studies on real-world multilingual bugs in detail. For each bug, the bug overview and a buggy line causing the bug are described. Then, I illustrate actual mutants that assist the proposed technique to infer the buggy line.

Chapter 8 discusses observations made through the experiment. I address advantages of mutation based approach and effectiveness of new mutation operators for localizing multilingual bugs

Chapter 9 concludes this dissertation with future work.

Chapter 2. Background and Related Work

2.1 Multilingual Bugs

A multilingual program is composed of code segments in different languages that execute each others through language interfaces (e.g., JNI [33] and Python/C). These language interfaces require the multilingual programs to follow safety rules across language boundaries. Lee *et al.* [29] classifies safety rules in Java/C and Python/C programs into three classes: (1) state constraints, (2) type constraints, and (3) resource constraints.

- *State constraints* ensure that the runtime system of one language is in a consistent state before transiting to/from a system of another language. For instance, JNI requires the program to not propagate a Java exception before executing a JNI function from a native method in C. Python/C requires C code to not acquire a global lock in Python twice because the global lock causes deadlock.
- *Type constraints* ensure that the programs in different languages exchange valid arguments and return values of expected types at a language boundary. For instance, the `NewStringUTF` function in JNI expects its arguments not to be `NULL` in C.
- *Resource constraints* ensure that the program manages resources correctly. These resource constraints are comparable to the contracts of calling the `free` function for dynamically allocated memory in C. For example, a local reference l to an Java object obtained in a native method m_1 should not be reused in another native method m_2 since l becomes invalid when m_1 terminates [33] (see Chapter 6 as an example of a multilingual bug that violates this resource constraint).

A multilingual bug is caused by violating safety rules at language interface (i.e., foreign function interface (FFI) bugs), and/or by unintended interactions of code across language boundaries. When a program breaks an interface safety rule, the program crashes or generates undefined behaviors. Multilingual programs respecting all interface safety rules still can have multilingual errors when the cause-effect chain goes through languages interfaces. For instance, a program would leak a C object referenced by a Java object that is garbage collected at some point. The cause of the memory leak is in Java at the last reference to this Java object while the effect is in C because Java code is expected to free the C object (Section 3.1).

2.2 Multilingual Bug Checkers

There are several static or dynamic approaches to check the constraints described in Section 2.1. The majority of approaches targets the constraint violations in JNI because the most used language combination is Java/C.

2.2.1 Static approach

Furr *et al.* [18] presents multilingual type inference system (called JSaffire) to ensure the type safety of multilingual programs which use JNI. Some JNI functions (e.g., `FindClass`, `GetFieldID`, `GetMethodID`) use constant strings as a parameter to indicate a class name, a field name or a method signature in Java. Since the constant string does not contain any type information, it is easy to get wrong. JSaffire infers the type assumption of the constant strings in C and checks that the type assumption is consistent with actual Java definition. However, JSaffire only checks statically type constraint violations at JNI invocations while state constraint violations (e.g., pending exception) are still not resolved.

Tan *et al.* [49] proposes a framework (calling it ILEA) that enables existing Java analysis tools to understand the behavior of C code. To do that, ILEA partially compiles the target C code into a specification which is based on Java so that an existing Java analysis tools can understand the behavior of C code through the Java specification and finally the tools can be extended to cover C code. However, ILEA is suffer from its compilation precision, and also from the effectiveness of the Java analysis tools.

Although various static checkers for multilingual bug including above works exist, they are hard to use in practice because they have trouble with a lot of false positives [18,27]. Also, since the specification of interface languages natively include dynamic properties, static approaches themselves have limitations.

2.2.2 Dynamic approach

Dynamic approach detects interface errors by inserting codes at a languague boundary [48] or by monitoring language transitions between different languages [29].

Tan *et al.* [48] proposes a framework called Safe Java Native Interface (SafeJNI) to ensure type safety of multilingual programs which contain Java and C components. SafeJNI explains several pitfalls in JNI (e.g., direct access through Java references, exception handling) and then suggests three methodologies to prevent these vulnerabilities. (1) a pointer type system based on CCured [40], which statically enforces usage of C pointers according to their capability (e.g., read-only pointer) (2) inserting dynamic checks which are performed before and after a JNI function is invoked (3) a memory management scheme which ensures safe memory management by additional validity tag for C pointers referencing Java object. However, even though SafeJNI shows acceptable effectiveness and performance in its evaluation, SafeJNI is difficult to use in practice because it needs source-code rewriting.

Lee *et al.* [29] suggests a JNI bug detector, called Jinn, which checks violation of all constraints in Section 2.1 dynamically at runtime. Jinn is based on 11 finite-state machines each of which describes JVM states regarding JNI function invocations. Each state in the finite-state machines is moved to another state according to a corresponding language transition in JNI function invocations. If a state is transited to predefined error state, Jinn reports an error message and the program is terminated. However, Jinn cannot detect unintended interactions of code across language boundaries (i.e., program semantic error which does not violate any pre-defined constraints)

JVMs, such as HotSpot [7] and IBM J9 [17], support a built-in dynamic JNI checker (calling it checkJNI) in a command-line option (i.e., -Xcheck:jni). checkJNI detects automatically some JNI rule violations (e.g., null value passed to JNI function, invalid JNI reference is used) at the cost of slower execution. Since checkJNI does not handle all JNI rule violations as a critical error, in some cases (e.g., pending exception error), checkJNI just reports warning messages and does not kill the program.

2.3 Fault Localization Techniques

Fault localization techniques [35,50] aim to locate the root cause of an error in the target program (i.e., the second step of debugging) by observing test runs. Fault localization has been extensively studied for monolingual programs both empirically [11,22,38] and theoretically [51,54]. Among the fault localization techniques, spectrum-based fault localization (SBFL) techniques are widely used due to its intuitive approach. However, recently mutation-based fault localization techniques are introduced to overcome the low accuracy of spectrum-based fault localization techniques.

2.3.1 Spectrum-Based Fault Localization

SBFL technique utilizes program spectrum, which is a collection of execution information of a program, to infer that a code entity of the program (i.e., a program element) is suspicious for an error if the code entity is likely executed when the error occurs. Note that SBFL techniques are *language agnostic* because they calculate the suspiciousness scores of target code entities by using information on the testing results (i.e., fail/pass) of test cases and the code coverage of these test cases without complex semantic analyses.

Suppose that a program $P = \{e_1, \ldots, e_n\}$, which contains a set of program elements e_i, is executed by a set of test cases $T = \{t_1, \ldots, t_m\}$, and the execution result is pass or fail. SBFL techniques' scoring metric that calculating suspiciousness score of a program element e can be expressed by 4 notations: $T_f(e)$, $T_p(e)$, T_f, T_p, where $T_f(e)$ is a set of test cases that covers e and fails on P, $T_p(e)$ is a set of test cases that covers e and passs on P, T_f is a set of test cases that fails on P, and T_f is a set of test cases that passes on P.

In the empirical study by Lucia *et al.* [34], effectiveness of 40 spectrum-based fault localization techniques is evaluated on programs from Siemens test suite and three larger programs from Software Infrastructure Repository (SIR) benchmark. As a result, *Ochiai* [41] outperforms other SBFL techniques in terms of the average percentage of code inspected for localizing the root cause of a buggy program. *Ochiai* is derived from a product term P(P fails|e covered) $\times$ P(e covered|P fails), where first term represents *precision*, second term indicates *recall*, and through the derivation the following metric is used for calculating suspiciousness of a program element e.

$$Ochiai(e) = \frac{|T_f(e)|}{\sqrt{|T_f| \times (|T_f(e)| + |T_p(e)|)}} \tag{2.1}$$

According to the other empirical evaluation for 42 SBFL techniques by Hofer *et al.* [20], *Jaccard* [21] as well as *Ochiai* is the best performing metric for fault localization in *spreadsheet* applications. *Jaccard* matric originates from an intuitive approach which is the number of matches over the number of attributes, that is $\frac{|T_f(e)|+|T_p(\neg e)|}{|T_f(e)|+|T_f(\neg e)|+|T_p(e)|+|T_p(\neg e)|}$, where $T_f(\neg e)$ is a set of test cases that does not cover e and fails on P and $T_p(\neg e)$ is a set of test cases that does not cover e and passes on P. For this formula, since *Jaccard* considers that $|T_p(\neg e)|$ is an irrelavent property to infer the root cause of a fault, $|T_p(\neg e)|$ is discarded and then finally *Jaccard* metric is defined as follows:

$$Jaccard(e) = \frac{|T_f(e)|}{|T_f(e)| + |T_f(\neg e)| + |T_p(e)|} \tag{2.2}$$

Instead of empirical evaluation for effectiveness of SBFL techniques, Naish *et al.* [39] theoretically analyzes 30 SBFLs on a sample program that simulates a single fault program. As a result, an optimal ranking matric (called *Op2*) is proposed, which is effective for the given model in terms of that the matric gives different ranks when necessary. *Op2* metric is defined as follows:

$$Op2(e) = |T_f(e)| - \frac{|T_p(e)|}{|T_p| + 1} \tag{2.3}$$

However, the accuracy of SBFL techniques is often too low to localize faults in large real-world programs. The reason is that although SBFL technique are premised on the availability of a high-quality test suite that provides good coverage of the program entities [14], the quality of a test suite in real-world programs is in general poor to support SBFL techniques.

2.3.2 Mutation-Based Fault Localization

To improve the accuracy of fault localization, mutation-based fault localization (MBFL) techniques have been proposed recently. These techniques can analyze diverse program behaviors using mutants (i.e., target program versions that are generated by applying simple syntactic code changes such as replacing `if(x>10)` with `if(x<10)`). MBFL techniques are also language agnostic since they utilize only information on the testing results (i.e., pass or fail) of test cases on the original target program and its mutants. Moon *et al.* [38] demonstrate that their MBFL technique (calling it MUSE) is 6.5 times more precise than state-of-the-art SBFL techniques such as Ochiai and Op2 on the 15 versions of the SIR subjects. The key idea of MUSE is as follows. Consider a faulty program P whose execution with some test cases results in error. Let m_f be a mutant of P that mutates the faulty statement, and m_c be one that mutates a correct statement. MUSE assesses the suspiciousness of a statement based on the following two observations:

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

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

For a statement s of P and given test suite T, let $T_f(s)$ be the set of tests that covered s and failed on P, and $T_p(s)$ the set of tests that covered s and passed on P. With respect to a fixed set of mutation operators, let $mut(s) = \{m_1, \dots, m_k\}$ be the set of all mutants of P that mutates s with observed changes in test results. For each mutant $m_i \in mut(s)$, let $T_f^{m_i}$ and $T_p^{m_i}$ be the set of failing and passing tests on m_i respectively. And let T_{f2p} and T_{p2f} be the set of tests that their results are changed from failure to pass and vice versa between before and after executing all mutants of P on the test suite T. Then, $MUSE(s)$ is defined as follows:

$$MUSE(s) = \frac{1}{|mut(s)|} \sum_{m \in mut(s)} \left(\frac{|T_f(s) \cap T_p^m|}{|T_{f2p}|} - \frac{|T_p(s) \cap T_f^m|}{|T_{p2f}|}\right) \tag{2.4}$$

The first term, $\frac{|T_f(s) \cap T_p^m|}{|T_{f2p}|}$, reflects the first observation: it is the proportion of the number of tests that failed on P but now pass on a mutant m that mutates s over the total number of all failing tests that pass on a some mutant (it increases the suspiciousness of s if mutating s causes failing tests to pass). Similarly, the second term, $\frac{|T_p(s) \cap T_f^m|}{|T_{p2f}|}$, reflects the second observation, being the proportion of the number of tests that passed on P but now fail on a mutant m that mutates s over the total number of all passing tests that fail on a some mutant (it decreases the suspiciousness of s if mutating s causes passing

		Execution Traces of Test Cases (x, y)							Jaccard		Ochiai		Op2	
	int max; void setmax(int x, int y){	TC_1 (3,1)	TC_2 (5,-4)	TC_3 (0,-4)	TC_4 (0,7)	TC_5 (-1,3)	$\lvert f_P(s)\rvert$	$\lvert p_P(s)\rvert$	Susp.	Rank	Susp.	Rank	Susp.	Rank
s_1:	max = −x; //should be 'max = x;'	●	●	●	●	●	2	3	0.40	5	0.63	5	1.25	5
s_2:	if (max < y){	●	●	●	●	●	2	3	0.40	5	0.63	5	1.25	5
s_3:	max = y;	●	●		●	●	2	2	0.50	2	0.71	2	1.50	2
s_4:	if (x*y<0)	●	●		●	●	2	2	0.50	2	0.71	2	1.50	2
s_5:	print(" diff .sign");}		●			●	1	1	0.33	6	0.50	6	0.75	6
s_6:	print(max);}	●	●	●	●	●	2	3	0.40	5	0.63	5	1.25	5
	Test Results	Fail	Fail	Pass	Pass	Pass								

		Test Result Changes							MUSE	
Statements	Mutants	TC_1 (3,1)	TC_2 (5,-4)	TC_3 (0,-4)	TC_4 (0,7)	TC_5 (-1,3)	$\lvert f_P(s) \cap p_m\rvert$	$\lvert p_P(s) \cap f_m\rvert$	Suspiciousness	Rank
s_1: max = −x;	m1: max −= x−1;			P→F			0	1	0.475	1
	m2: max=x;	F→P	F→P				2	0		
s_2: if (max < y){	m3: if (!(max<y)){			P→F	P→F	P→F	0	3	-0.125	6
	m4: if(max==y){				P→F	P→F	0	2		
s_3: max = y;	m5: max = −y;				P→F	P→F	0	2	-0.100	4
	m6: max = y+1;				P→F	P→F	0	2		
s_4: if (x*y<0){	m7:if (!(x*y<0))				P→F	P→F	0	2	-0.075	3
	m8:if(x/y<0)					P→F	0	1		
s_5: print(" diff .sign");}	m9:return;					P→F	0	1	-0.050	2
	m10:;					P→F	0	1		
s_6: print(max);}	m11:printf(0);}				P→F	P→F	0	2	-0.125	6
	m12:;}			P→F	P→F	P→F	0	3		

Figure 2.1: Example of how fault localization techniques work

tests to fail). When averaged over $|mut(s)|$, the first term and the second term become the probability of test result change per mutant, from failing to passing and vice versa respectively.

There exist a few other MBFL approaches. Papadakis *et al.* [42, 43] utilizes the similarity between the behavior of generated mutants and the behavior of a program with an "unknown fault". If a mutant (i.e., a mutated program where single statement is transformed) behaves like non-mutated program (i.e., test results are same), then the mutated statement is suspected of faulty. This approach is different from *MUSE* in that not considering test result changes comes from mutant executions.

2.3.3 An Illustrative Example

Figure 2.1 [1]presents an example of how SBFL techniques (e.g., *Jaccard*, *Ochiai*, *Op2*) and MBFL technique (e.g., *MUSE*) localize a fault. The example code is a function called setmax(), which sets a global variable max (initialized to 0) with y if x < y, or with x otherwise. Statement s_1 contains a fault, as it should be max=x. Let us assume that we have five test cases ($tc1$ to $tc5$): the coverage of individual test cases are marked with black bullets (●). TC_1 and TC_2 fail because `setmax()` updates max with the smaller number, y. The remaining test cases pass. Thus, $|f_P| = 2$ and $|p_P| = 3$.

First, MUSE generates mutants by mutating only one statement at a time. For the sake of simplicity, here we assume that MUSE generates only two mutants per statement, resulting in a total of 12 mutants, $\{m_1, \ldots, m_{12}\}$ (listed under the "Mutants" column of Figure 2.1). Test cases change their results after the mutation as noted in the middle column. For example, TC_1, which used to fail, now passes on the

[1]This example is originally presented in [38]

mutant $m2$.

Since there are two changes from failure to pass, $f2p = 2$ (TC$_1$ and TC$_2$ on m_2) while $|f_P| = 2$. Similarly, $p2f = 20$ (see the changed results of TC$_3$, TC$_4$, and TC$_5$), while $|p_P| = 3$.

MUSE calculates the suspiciousness of s$_1$ as $\frac{1}{2} \cdot \{(\frac{0}{2} - \frac{1}{20}) + (\frac{2}{2} - \frac{0}{20})\} = 0.475$, where $|f_P(s_1) \cap p_{m_1}| = 0$ and $|p_P(s_1) \cap f_{m_1}| = 1$ for m_1 and $|f_P(s_1) \cap p_{m_2}| = 2$ and $|p_P(s_1) \cap f_{m_2}| = 0$ for m_2. MUSE calculates the suspiciousness scores of the other five statements as -0.125, -0.100, -0.075, -0.050, and -0.125. The suspiciousness of the s$_1$ is the highest. In contrast, Jaccard, Ochiai, and Op2 choose s$_3$ and s$_4$ as the most suspicious statements, while assigning the fifth rank to the actual faulty statement s$_1$. The example shows that MUSE can precisely locate certain faults that the state-of-the-art SBFL techniques cannot.

Chapter 3. MUSEUM: Mutation-Based Fault Localization for Real-world Multilingual Programs

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

MUSEUM targets both monolingual and multilingual bugs. To localize multilingual bugs precisely, MUSEUM utilizes conventional mutation operators and new mutation operators designed for directly mutating interactions between language interfaces. These new mutation operators (Section 3.3) improve the accuracy of MBFL by generating mutants whose testing results are informative to locate multilingual bugs (Chapter 6).

3.1 Motivating Example

Target program

Figure 3.1 presents a target Java/C program with a memory leak bug failing the assertion at Line 71 [1]. 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 [33, p. 123] of wrapping native data structures, which is widely used in language bindings for legacy C libraries. The `peer` field (Line 4) is an opaque pointer from Java to C to point to a dynamically allocated integer object in C. The `CPtr` constructor (Line 9) executes the `nAlloc` native method (Lines 17–21) to allocate an integer object in C and stores the address of the integer object in `peer`. While JVMs automatically reclaim a `CPtr` object once the object becomes unreachable in the Java heap, the clients of `CPtr` are required to dispose manually the integer object by executing `dispose` (Line 12) on the `CPtr` object. If the client does not dispose an `CPtr` object before it becomes unreachable, the peer integer object becomes a unreachable memory leak in C.

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

`ClientTest` (Lines 48–73) is a Java class of driving test cases directly for `Client` and indirectly for `CPtr`. It contains one passing test `passingTest` (Lines 55–63) and one failing test `failingTest` (Lines 64–73). The testing oracle validates a program execution by using (1) the assertion statements (Lines 59 and 69) and (2) the exception handler statements (Lines 61 and 71). The assertion statements

[1]This example is a simplified version of a real-world bug found in Azureus 3.0.4.2 (Bug1 in Table 4.1).

```
/* CPtr.java */
public class CPtr {
  static {System.loadLibrary("CPtr");}
  private final long peer;
  private native long nAlloc();
  private native void nFree(long pointer);
  private native int nGet(long pointer);
  private native void nPut(long pointer, int x);
  public CPtr(){peer = nAlloc();}
  public int get(){return nGet(peer);}
  public void put(int x){nPut(peer, x);}
  public void dispose(){nFree(peer);} }

/* CPtr.c */
#include <jni.h>
#include <stdlib.h>
jlong Java_CPtr_nAlloc(JNIEnv *env,jobject o){
  jint *p;
  p =(jint *)malloc(sizeof (jint)); /*Mutant m1*/
  return (jlong)p;
}
void Java_CPtr_nFree(JNIEnv *env,jobject o,jlong p){
  free((void *)p);
}
jint Java_CPtr_nGet(JNIEnv *env,jobject o,jlong p){
    return *(jint *)p;
}
void Java_CPtr_nPut(JNIEnv *env,jobject o,jlong p,
jint x){
    *((jint *)p) = x;
}

/* Client.java*/
public class Client {
  CPtr m = null;
  void add(int x){
    m = new CPtr(); /*Mutant m2*/
    m.put(x);
  }
  int remove(){
    int x = m.get();
    m.dispose();
    m = null;
    return x;  /*Mutant m3*/
} }

/* ClientTest.java */
import java.util.*;
public class ClientTest {
  static final List pinnedObj=new LinkedList();
  public static Object pinObject(Object o){
    pinnedObj.add(o);
    return o;
  }
  void passingTest(){ // passing test case
    try {
      Client d = new Client() ;
      d.add(1) ;
      assert d.remove() == 1;
    } catch(VirtualMachineError e) {
      assert false; /*potential memory leak in C*/
    }
  }
  void failingTest(){ // failing test case
    try {
      Client d = new Client() ;
      d.add(1) ;
      d.add(2) ;
      assert d.remove() == 2;
    } catch (VirtualMachineError e) {
      assert false; /*potential memory leak in C*/
    }
} }
```

Figure 3.1: A Java/C program leaking memory in C after garbage collection in Java

at Line 59 and Line 69 validate the program state after executing a sequence of `add` and `remove` by checking if `remove` correctly returns the last value given by `add`. On the other hand, the exception handler statements at Line 60 and Line 70 detect failures at arbitrary locations. For instance, runtime monitors such as QVM [13] and Jinn [29] would throw an asynchronous Java exception either at GC safe points or at language transitions.

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`.

Failing test

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

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* [2]

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

[2] See `Pin-Java-Object` mutation operator in Table 3.1

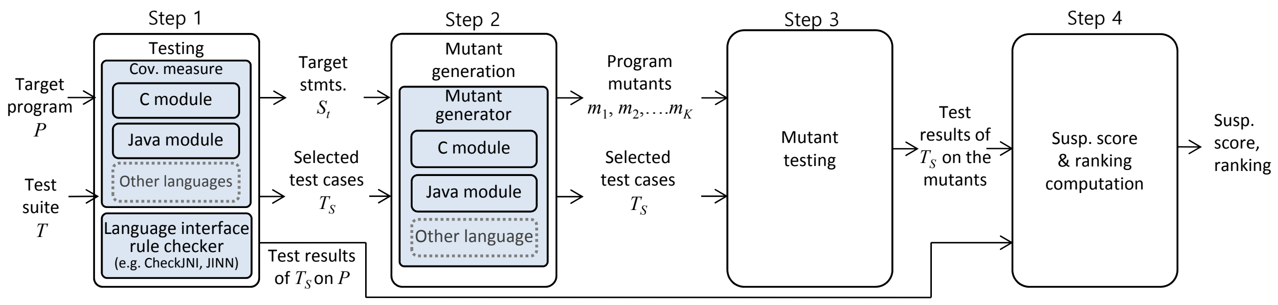
Step 1
Testing
Cov. measure
C module
Java module
Other languages
Language interface rule checker (e.g. CheckJNI, JINN)
Target program P
Test suite T
Target stmts. S_t
Selected test cases T_S
Test results of T_S on P
Step 2
Mutant generation
Mutant generator
C module
Java module
Other language
Program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

Figure 3.2: Fault localization process of MUSEUM

m_3, *a mutant obtained by replacing the return value with 0 in Line 44*

This mutation replaces the variable `x` with an integer constant 0 at Line 44. This mutation fails the assertion at Lines 59 and 69 since the return value of `remove` is always `0`.

From these testing results, MUSEUM concludes that Line 37 is more suspicious than Line 19 and Line 44 because the failing test case passes only with m_2 and the passing test case fails with m_1 and m_3 (see Step 4 of Section 3.2).

Locating the root cause of this memory leak poses challenges in runtime monitoring and fault localization techniques. Memory leak detectors [23, 53] locate memory leaks and their allocation sites not the cause of the leaks in general. While some leak chasers [13, 15, 52] locate the cause of memory leak, they do not scale well across language boundaries since they do not track opaque pointers and their staleness values across languages. SBFL techniques cannot localize the bug because both `passingTest` and `failingTest` cover the same branches/statements in their executions. Consequently, SBFL techniques cannot indicate any code element that is more correlated with the failure than the others.

3.2 Fault Localization Process of MUSEUM

Figure 3.2 describes how MUSEUM localizes faults. MUSEUM takes the target source code and the test cases of the target program as input, and returns the suspiciousness scores of the target code lines as output. MUSEUM has the following basic assumptions on a target program P and test suite T:

1. Existence of test oracles

   A target program has explicit or implicit test oracle mechanism (i.e., user-specified `assert`, runtime failure such as null-pointer dereference, and/or runtime monitor such as Jinn [29]) 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 [29], and QVM [13] (Section 2.1).

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 [12]. [3]

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

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

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

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

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

The first term, $\frac{|f(s) \cap p_{m_i}|}{f2p}$, reflects the first observation: it is the proportion of the number of tests that failed on P but now pass on a mutant m_i that mutates s over the total number of all failing tests that pass on a some mutant (the suspiciousness of s increases if mutating s causes failing tests to pass). Similarly, the second term, $\frac{|p(s) \cap f_{m_i}|}{p2f}$, reflects the second observation, being the proportion of the number of tests that passed on P but now fail on a mutant m_i that mutates s over the total number of all passing tests that fail on a some mutant (the suspiciousness of s decreases if mutating s causes passing tests to fail). After dividing the sum of the first term and the second term by $|mut(s)|$, $Susp(s)$ indicates the probability of s to be a faulty statement based on the changes of test results on P and $mut(s)$. [4]

3.3 New Mutation Operators for Multilingual Bugs

In addition to the conventional mutation operators, MUSEUM utilizes new mutation operators to effectively localize multilingual bugs because these mutation operators can directly mutate interactions between language interfaces. We have made 15 new mutation operators, which change semantics of a target program regarding the JNI constraints based on the previous JNI bug studies [5, 17, 29, 48] and JNI specifications [33]. Table 3.1 shows the list of the new mutation operators. The description of the new mutation operators are as follows:

1–3. These mutation operators clear, propagate, or generate a pending exception in a native method to ensure the JVM state constraints. Targets of the three mutation operators are all JNI function calls (i.e., `(*env)->`$<$*JNIFunction*$>$`(...);`). For example, `Clear-pending-exceptions` clears a pending exception in currrent thread by inserting

[3]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 [38].

[4] If a target statement has no mutant (i.e., $|mut(s)|$=0), $Susp(s)$ is defined as 0. MUSEUM defines the first term as 0 if $f2p$ is 0. Similarly, the second term is defined as 0 if $p2f$ is 0. For a concrete example of how to calculate the suspiciousness score of MBFL, see Section II.C of Moon *et al.* [38].

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

immediately before a JNI function call and immediately after a JNI function call that may throws a Java exception.[5] `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 throws 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 throws a Java exception. The first and the second mutation operators are defined based on a best practice in JNI programming [17] and a general solution for JNI exception bugs [30]. The third mutation operator is motivated by a case of a real-world multilingual bug regarding exception handling across language boundaries [10].

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. [6] In other words, `Type-cast-to-jboolean` changes an assignment `jbool_var = int_expr;` with

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

   This mutation operation is motivated by the common pitfall of JNI programming [33, pp.132–133].

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

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

   This mutation operator is motivated by a report of a real-world bug found in Eclipse 3.4 [29].

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 is inspired by a real-world bug with an incorrect array data transfer from Java to C [6].

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);
   ```

[5] 154 among total 229 JNI functions may throw an exception [33].

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

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 [33, pp.131–132].

8–13. These mutation operators increase or decrease the life time of a reference to a Java object (and probably the life time of the referenced Java objects too). For example, `Make-global-reference` increases the life time of a local reference l by making the reference as a global one. In other words, `Make-global-reference` inserts the following statement after an assignment statement to a local reference l (i.e., l = *expr*):

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

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

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

    We have developed four other mutation operators for local references and weak global references. These mutation operators are related to a bug fix pattern regarding reference errors in native code [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 the class variable, and Java garbage collectors cannot reclaim the object. This mutation operator intends to extend the lifetime of Java objects in a target program and influence interactions of Java and native memory managements. This mutation operator is inspired by a safe memory management scheme of SafeJNI [48].

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 [17].

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 [16] to obtain the coverage information on C code and Java code of a target program, respectively. MUSEUM uses existing mutation tools Proteum [36] and PIT, together

Table 3.1: New mutation operators of MUSEUM

No.	Mutation operator	Corresponding language interface rule (Section 2.1)
1	`Clear-pending-exceptions`	JVM state constraints
2	`Propagate-pending-exceptions`	
3	`Throw-new-exceptions`	
4	`Type-cast-to-jboolean`	Type constraints
5	`Type-cast-to-superclass`	
6	`Replace-array-elements-with-constants`	
7	`Replace-target-Java-member`	
8	`Make-global-reference`	Resource constraints
9	`Remove-global-reference`	
10	`Make-weak-global-reference`	
11	`Remove-weak-global-reference`	
12	`Make-local-reference`	
13	`Remove-local-reference`	
14	`Pin-Java-object`	
15	`Switch-array-release-mode`	

with the 15 new mutation operators for multilingual bugs (Section 3.3). Proteum implements 107 mutation operators defined in Agrawal *et al.* [12] which mutate C code in source level. Among the 107 mutation operators, MUSEUM uses 75 mutation operators that change only one statement. To reduce the runtime cost of the experiments, MUSEUM generates only one mutant for every applicable operator at each mutation point [7]. MUSEUM generates Java mutants by using PIT which mutates Java bytecode. MUSEUM uses all 14 mutation operators of PIT. Among the 15 new mutation operators, 14 new mutation operators for C code are implemented with Clang, and the one new mutation operator for Java (i.e., `Pin-Java-object`) is built with the ASM bytecode engineering tool.

[7] For example, `if(x+2>y+1)` has one mutation point (`>`) for ORRN (mutation operator on relational operator) and two points (`2` and `1`) for CCCR (mutation operator for constant to constant replacement) [12]. MUSEUM generates only one mutant like `if(x+2<y+1)` using ORRN and only `if(x+0>y+1)` and `if(x+2>y+0)` using CCCR. The selection of a mutant to generate using a mutation operator depends on the Proteum implementation. Note that MUSEUM generates multiple mutants for a code location when multiple mutation operators are applied to the code location .

Table 4.1: Target multilingual Java/C bugs, sizes of the target code, the number of test cases used, and references

Bug	Target program	Symptom	Size of target program				# of TC used	Bug report or bug-fixing revision
			Java		NativeC			
			Files	LOC	Files	LOC		
Bug1	Azureus 3.0.4.2	Memory leak in C	2,705	340.6K	N/A	N/A	8	CVS revision 1.64 of ListView.java [1]
Bug2	`sqlite-jdbc` 3.7.8	Assertion violation in Java	20	4.6K	3	1.8K	150	Issue 16 [8]
Bug3	`sqlite-jdbc` 3.7.15	Assertion violation in Java	19	4.2K	2	1.7K	159	Issue 36 [9]
Bug4	`java-gnome` 4.0.10	Invalid JNI reference in C	1,097	64.2K	496	65.6K	170	Bug 576111 in Bugzilla database [3]
Bug5	`java-gnome` r-658	Double free in C	1,134	67.1K	514	69.2K	184	Subversion revision 659 [2]
Bug6	`SWT` 3.7.0.3	Segmentation fault in C	582	118.7K	29	34.4K	50	Bug 322222 in the Eclipse bug repo. [4]

Chapter 4. Empirical Evaluation

This chapter evaluates MUSEUM on the six bugs in four real-world multilingual programs to demonstrate its effectiveness. Section 4.1 describes the experiment setup, and Section 4.2 presents the fault localization results. [1]

4.1 Experiment Setup

4.1.1 Real-world multilingual program bugs

Table 4.1 presents the six multilingual bugs in four real-world programs 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. As described in the assumption 1 for fault localization (Section 3.2), the bug reports and commit logs in the last column describe the symptoms of the target bugs so that our test oracle detects test failures. A corresponding bug report indicates both buggy version and its fixed version. We have applied MUSEUM to Java code and native C code of the target program, not library code nor external system code. All target programs are written in Java and C except for `Azureus`. While `Azureus` is a pure Java program, it triggers a memory leak in C when it misuses the application program interface of the Eclipse SWT library written in Java and C.

4.1.2 Real-world Test Cases

Regarding test cases, we have used the test cases maintained by the developers of the target programs. We utilize the test cases of the fixed version, at least one of which reveals the target bug in the buggy version (see the assumption 2 in Section 3.2). If the fixed version does not have a test case that fails on the buggy version (e.g., `Azureus` memory leak bug), we create a failing test case based on the bug report. In addition, to localize a fault precisely, we focus to localize one bug at a time by building a new test suite out of the original test suite. The new test suite consists of one failing test case and all passing test cases that cover at least one statement executed by the failing test case.

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

4.1.3 System Platform

The experiments were performed on the 30 machines equipped with Intel i5 3.4 GHz with 8 GB main memory (we performed experiment on one core per machine). All machines run Ubuntu 8.10 32-bits, gcc 4.3.2, and OpenJDK 1.6.0. MUSEUM distributes tasks of testing each mutant to the 30 machines. [2]

4.2 Experiment Results

Table 4.2 reports the experiment data on the six bugs. The second row counts the number of the source target lines which are executed by the failing test case (see Step 1 of Section 3.2). The third row shows the total number of the mutants generated by MUSEUM, and the forth row describes the total number of the target lines on which at least one mutant is generated. The fifth and sixth rows show the number of the mutants on which testing results have changed. The last row describes the runtime cost. For example, to localize the fault in Bug4 (an invalid JNI reference in C), we built a test suite containing one failing test case and 169 passing test cases out of the original test suite (see the eighth column of the fifth row at Table 4.1). For Bug4, MUSEUM generated 718 mutants with at least one mutant for 70% of the target lines (=130/186). Among the 718 mutants, there are two mutants on which the failing test case passes (see the sixth row of Table 4.2). [3] 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). However, these mutants contribute significantly to localize a fault precisely because a partial fix increases the first term of the suspiciousness metric formula much (see the metric formula in the Step 4 of Section 3.2).

Regarding time cost, although MUSEUM consumes large amount of computing resources to compile and execute a large number of mutants, the overall elapsed time can be modest. This is because tasks of testing mutants can be distributed to a large number of machines (e.g., Amazon EC2) as these tasks are independent to each other. For example, it takes around 90 minutes (=(12+66+54+96+60+252)/6) to localize each bug of the six multilingual bugs on average by utilizing 30 machines.

Table 4.3 compares the fault localization results of MUSEUM and the cutting-edge SBFL techniques including Jaccard [21], Ochiai [41], and Op2 [39]. Each entry reports the suspiciousness score ranking which is the maximum number of statements to examine until finding a faulty statement described in the bug report. The percentage number in the parentheses indicates the normalized rank of the faulty statement out of the total target statements (i.e., $\frac{\text{rank}}{\text{\# of the target statements}}$). The second row of the table clearly shows that MUSEUM precisely identifies the buggy statement. MUSEUM ranks the buggy statements in Bug1, Bug3, and Bug4 as the most suspicious statements (i.e., the first rank). 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, respectively (e.g., for Bug5, the suspiciousness scores of the eight statements including the buggy statement are equal). Thus, from these experiments, we conclude that MUSEUM localizes a multilingual bug precisely.

In contrast, SBFL techniques fail to localize multilingual bugs precisely. For Bug6, Op2 ranks the buggy statement as the 3,494th among the 3,494 target statements (see the fifth row of Table 4.2), which means that a developer has to examine all target statements (i.e., 100%) to identify the bug. One main

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

[3] 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 4.2: Overview of the experiment data

	Bug1	Bug2	Bug3	Bug4	Bug5	Bug6
# of the target lines (executed by the failing test case)	1,939	299	443	186	186	3,494
# of mutants	2,861	691	965	718	369	9,479
# of lines which have a mutant	1,575	219	327	132	103	2,524
# of mutants that make a passing test case fails (breaking)	305	555	793	358	311	3,617
# of mutants that make a failing test case passes (partial fix)	1	3	7	2	51	32
Time cost (min)	12	66	54	96	60	252

Table 4.3: The ranks of the buggy line identified by MUSEUM and other SBFL techniques

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

deficiency of traditional SBFL techniques is the low resolution in fault localization (i.e., all statements in the same branch have same suspiciousness scores because the statements in the same branch are covered by the same test cases). This is one reason why those SBFL techniques assign the same suspicious ranks to the buggy statements in the experiments. In contrast, MUSEUM mutates each statement in multiple different ways and can assign different suspiciousness scores to the statements in the same branch.

Chapter 5. Case Study 1: Azureus Memory Leak Bug (Bug1)

5.1 Bug Overview

Azureus is a popular P2P file-sharing application. Azureus 3.0.4.2 has a memory leak bug because Azureus may allocate native memory for an `Image` object, but never de-allocate the memory. In more detail, `Image` class uses native C methods to allocate/de-allocate native memory for its member objects. Such allocated native memory should be explicitly freed by calling `Image.disposed()` before an `Image` object is not used anymore and garbage collected. Otherwise (i.e., `Image` object is garbage collected without calling `Image.dispose()`), the allocated native memory is not reclaimed and the native memory leak occurs. A problem is that `ListView.handleResize()` of Azureus allows an `Image` object to be garbage collected without calling `Image.disposed()`. The memory leak bug in Azureus 3.0.4.2 was fixed on Jan 12, 2008 [1] (used as an example in the QVM case study [13]).

The QVM parper [13] describes that the following code in `ListView.java` as the memory leak bug because the code misses `imgView.dispose()` before the line 523 where the old `Image` object referenced by `imgView` is not used anymore.

```
496:public void handleResize(boolean bForce) {
...
508:  if (imgView == null || bForce){
...
523:    imgView = new Image(...) ;
```

A developer might introduce this bug because he or she might fail to recognize the necessity of freeing the native memory in Java code, which is necessary for multilingual programs written in both Java and C.

5.2 Detailed Experiment Result

Since Azureus code has no test case, we created a failing test based on the bug report. Also, we built additional seven passing test cases which covers `handleResize()`. In addition, we add the following test oracle to detect memory leak on `Image` objects by using the `isDisposed()` method of `Image` class which checks if all members get freed:

```
public final class Image ... {
  protected void finalize() throws Throwable {
    if(!isDisposed()) reportLeak();
    super.finalize();}}
```

Finally, we make all test cases to invoke `System.gc()` and `Thread.sleep(10)` so that JVM can garbage collect all unused objects before the execution terminates.

MUSEUM identifies the line 523 of `ListView.java` as the most suspicious statement (i.e., the first rank). As shown in the second column of Table 5.1, the line 523 has four mutants. Mutant $m4$ is generated by the new mutation operator `Pin-Java-Object` designed for multilingual bugs (particularly targeting resource constraints, see Table 3.1). $m4$ adds the corresponding new statement in Table 5.1 before the target statement. Mutants $m1$, $m2$ and $m3$ are generated by the mutation operators provided by PIT which replace the target statement with the statements in the table.

$m4$ makes the failing test case as a passing one (the third column) because it adds an extra reference to the old object pointed by `imgView`, which prevents the old object from being garbage collected. As a result, the native memory leak does not occur. $m4$ make the failing test case as a passing one (the third column), which makes the first term of the MUSEUM suspiciousness metric large and increases the suspiciousness score significantly. In contrast, $m1$ and $m3$ make two and four passing test cases as failing ones (the fourth column), which increases the second term but in only limited degree due to the large denominator (i.e., $p2f$=1961).

Among the 1939 target statements, only the line 523 has a mutant that fixes the bug (i.e., $m4$) with regard to the given test cases and the given test oracle (i.e., making the failing test case passes). Consequently, the line 523 has the highest suspiciousness score. Thus, through the case study for the bug 1, we confirm that the new mutation operators such as `Pin-Java-Object` can increase the precision of MUSEUM.

Table 5.1: The four mutants generated at the buggy statement of the bug 1

No.	Mutant generated from `imgView = new Image(listCanvas.getDisplay(), clientArea); at line 523 of ListView.java`	$\lvert f(s) \cap p_m \rvert$	$\lvert p(s) \cap f_m \rvert$
$m1$	`imgView = null;`	0	2
$m2$	`imgView = new Image(null, clientArea);`	0	0
$m3$	`new Image(listCanvas.getDispaly(),clientArea);`	0	4
$m4$	`global_ref_list.add(imgView);`	1	0

Chapter 6. Case Study 2: Locating The Cause of Invalid Use of JNI References (Bug4)

This case study illustrates how MUSEUM localizes the cause of dangling JNI references (Bug4) accurately by using the new mutation operators (Table 3.1).

6.1 Bug Overview

Dynamic error detectors [29] detect Bug4 and report the calling context at the fault location of using the dangling JNI reference as an argument to a JNI function. However, they cannot report the cause location where the JNI reference was stored into a callback object in C heap, which occurs at Line 524 of `binding_java_signal.c` as indicated as the buggy statement in the bug report:

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

When `bindings` at Line 387 is invoked, the `receiver` parameter is assigned with a local JNI reference. Line 524 stores the local reference in a data structure in the C heap pointed by `bjc`. However, once `bindings` returns back to Java, the local reference stored in `bjc->rec` is not valid anymore (i.e., becoming a dangling reference, see Resource constraints in Section 2.1). Later, when the application calls a JNI function with an argument containing the dangling reference, the application crashes with a JNI invalid argument error.

6.2 Detailed Experiment Result

MUSEUM localizes the fault *exactly* by ranking Line 524 as the most suspicious statement (i.e., the first rank without a tie). Table 6.1 describes nine mutants ($m1$ to $m9$) that are generated by mutating Line 524. The second column shows the changed statement of each mutant. The third and the forth columns report the number of tests that failed on the original program but pass on the mutant (i.e., $|f(s) \cap p_m|$), and the number of tests that passed on the original program but fail on the mutant (i.e., $|p(s) \cap f_m|$), respectively (Section 3.2). $m1$, $m2$, $m4$, $m6$, $m8$, and $m9$ are generated by applying our new multilingual mutation operators in Table 3.1. These mutants are generated by inserting the statements of changing the life time of JNI references right after the target statement. $m3$, $m5$ and $m7$ from Proteum terminate the control flow at the level of procedure, statement, and whole program.

In the testing runs, our new mutation operators prevent mutated programs $m1$ and $m4$ from crashing in the failing test case (i.e., `Make-weak-global-reference` and `Make-global-reference` in Table 3.1, respectively). $m1$ and $m4$ turn the failing test case into a passing one (the third column) because they keep `bjc->rec` to store a weak global reference and a global reference respectively and eliminate the

Table 6.1: The nine mutants generated by mutating the buggy statement of Bug4

No.	Mutant generated by mutating Line 524 of `bindings_java_signal.c`	$\|f(s)$ $\cap p_m\|$	$\|p(s)$ $\cap f_m\|$
$m1$	`bjc->rec=receiver;` `bjc->rec=(*env)->NewWeakGlobalRef(env,bjc->rec);`	1	0
$m2$	`bjc->rec=receiver;` `bjc->rec=(*env)->NewLocalRef(env,bjc->rec);`	0	0
$m3$	`return; // return back to the caller.`	0	0
$m4$	`bjc->rec=receiver;` `bjc->rec=(*env)->NewGlobalRef(env,bjc->rec);`	1	0
$m5$	`; // remove a statement at Line 524`	0	2
$m6$	`bjc->rec=receiver;` `(*env)->DeleteGlobalRef(env, bjc->rec);`	0	2
$m7$	`kill(getpid(), 9); //terminate the process`	0	2
$m8$	`bjc->rec=receiver;` `(*env)->DeleteLocalRef(env, bjc->rec);`	0	2
$m9$	`bjc->rec=receiver;` `(*env)->DeleteWeakGlobalRef(env,bjc->rec);`	0	2

dead reference problem caused by the short-lived local reference. On the other hand, the conventional mutation operators (i.e., $m3$, $m5$, and $m7$) do not affect the test results. $m1$ and $m4$ make the first term of the MUSEUM suspiciousness metric large and increase the suspiciousness score of Line 524 significantly because the denominator of the first term is small (i.e.,$f2p$=2) (Section 3.2). In contrast, each of the mutants $m5$ to $m9$ make two passing test cases fail (the fourth column), which increases the second term but in only limited degree due to the large denominator (i.e., $p2f$=6034).

Among the 186 target statements, only Line 524 has mutants that fix Bug4 with regard to the given test cases and the given test oracle (i.e., making the failing test case pass). Consequently, Line 524 has the highest suspiciousness score due to the new mutation operators which generate partial fixes. Thus, through the case study for Bug4, we confirm that the new mutation operators such as `Make-global-reference` can increase the accuracy of MUSEUM (Section 8.2).

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

Note that these CCTs do not make adverse effect to MUSEUM. This is because the mutants (i.e., $m1$ to $m9$) obtained by mutating the buggy statement (i.e., Line 524) do not make these two CCTs fail as the target program and the mutants do not use `bjc->rec` as an argument to a JNI function call later with these CCTs (i.e., the mutation on the buggy statement is inactive with CCTs because the buggy statement is dormant with CCTs). Thus, these CCTs do not increase the second term of the MUSEUM suspiciousness metric (Section 3.2) and do not lower the suspiciousness score of the buggy statement.

Chapter 7. Case Study 3: Locating the Cause of a Segmentation Fault in Eclipse SWT (Bug6)

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

7.1 Bug Overview

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

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

The origin of `NULL` is the native C function (`callback`) that acts as a gateway from C to Java in the SWT library. `callback` returns `NULL` when a Java exception is pending in the current thread. While the detection of this bug is trivial, locating the root cause took a heroic debugging effort for more than a year with hundreds of comments from dozens of programmers. This bug was difficult for experts to debug since the cause-effect chain goes through Java exception propagation and language transitions. Although the multilingual debuggers [28] aid programmers to locate the origin of `NULL`, they do not locate the root cause of the bug.

The root cause is turned out to be an immature implementation of a callback handler at Line 2602 of `Display.java`. [1]

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

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

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

Table 7.1: The top four statements of the SWT target code whose suspiciousness scores are high

Rank	Susp. score	$\lvert f(s) \cap p_m \rvert$	$\lvert p(s) \cap f_m \rvert$	Statement	Mutant
3	0.0313	1	0	`/*Display.java:2602*/` `OS.G_OBJ_SET_CONSTRUCTOR(PLClass,NewProc);`	`; /* the function call is removed */`
3	0.0313	1	0	`/*OS.java:8115*/` `return _major_version;`	`return 0;`
3	0.0313	1	0	`/*OS.java:8125*/` `return _minor_version;`	`return 0;`
4	0.0306	1	1	`/*Display.java:2392*/` `initializeSubclasses();`	`; /* the function call is removed */`

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

7.2 Detailed Experiment Result

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

Table 7.1 presents the top four suspicious statements and their mutants, which increase the suspiciousness scores. [2] MUSEUM ranks the first three statements in a tie as rank 3 that include the location of the root cause of the failure: `Display.java:2602`. The mutants for the top three statements change the failing test case into passing one without affecting passing test cases (see the third and the fourth columns).

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

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

Table 8.1: Statistics on the mutation operators that generate mutants in the experiments

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

Chapter 8. Discussions

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

One of the issues that make debugging real-world programs difficult is the poor quality of a test suite because fault localization can be more accurate if a test suite covers more diverse execution paths. For large real-world programs, however, it is challenging to build test cases that exercise diverse execution paths because it is non-trivial to understand and control a target program. In addition, generating diverse test cases for multilingual programs has additional burden to learn and satisfy constraints for foreign function interface such as JNI constraints. Therefore, it is often the case that multilingual programs are developed with a set of similar test cases. As a result, as shown in Table 4.3, the SBFL techniques fail to precisely localize the six real-world multilingual programs.

For example, the statement coverages of the test suites used for Bug2 and Bug3 are around 85% and 86% and the SBFL techniques localize these bugs somehow precisely (i.e., the suspiciousness rank of Bug2 and Bug3 are 4 and 5, respectively). However, the statement coverages of the test suites used for Bug1, Bug4, Bug5, and Bug6 are around 1%, 22%, 24%, and 19% and the accuracy of the SBFL techniques for these bugs are very low (Table 4.3). In contrast, MUSEUM can overcome this limitation by achieving the effect of diverse test cases through the diverse mutants with limited test cases. Thus, MUSEUM can be a promising technique for debugging complex real-world multilingual programs.

8.2 Effectiveness of New Mutation Operators for Localizing Multilingual Bugs

Table 8.1 presents the information on the mutation operators that generate mutants on which the failing test case passes (i.e., partially-fixing mutants). The second row shows the number of tests where the failing test case on the target program passes on a mutant. The third row represents the number of mutation operators that generate a mutant on which the failing test case changes to pass. The fourth and the fifth rows show the similar information to the second and the third rows but on the mutants generated by the new mutation operators for multilingual programs (Section 3.3).

Table 8.1 shows that only the new mutation operators generate partially fixing mutants for Bugs 1 and 4 (i.e., since the numbers in the third row and the fifth row are the same). For Bug1, only the `Pin-Java-Object` mutation operator generates a mutant on which the failing test case passes, which indicates that the target statement of the mutant is closely related to the bug (i.e., memory leak in this case). Similarly, for Bug4, only `Make-global-reference` and `Make-weak-global-reference` generate the mutants that make the failing test case pass.

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

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

Chapter 9. Conclusion and Future Work

9.1 Summary

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

9.2 Future Work

As future work, I will focus on developing our technique in the direction of improving the effectiveness, the effeciency, and application to other language combinations (e.g., Python/C)

9.2.1 Improving Effectiveness

Although current MUSEUM localizes JNI bugs precisely in our case studies by employing 15 new mutation operators for JNI, in order to guarentee the promising results to other JNI bugs, I will add more mutation operators to cover more features in multilingual programs.

9.2.2 Improving Efficiency

Mutation based fault localization requires a lot of computing power to compile mutants and execute the mutants on given test suite. To reduce the computational cost, I will apply selective mutation testing, such as using selective mutation operator, reducing mutation points, and test case selection.

9.2.3 Application to Different Languages

MUSEUM can be applied to multilingual programs written in other language combinations such as Python-C, PHP-JavaScript, etc. To adapt and apply MUSEUM, a target language should provide a statement coverage measurement tool and a mutation tool. I believe that one can easily add a support for a new language in MUSEUM using those tools since required tools are available for the popular programming languages (e.g., mutations tools such as Nester for C#, Humbug for PHP, MutPy for Python, Heckle for Ruby, Grunt-mutation-testing for Java Script and coverage tools such as NCover for C#, Xdebug for PHP, JSCover for JavaScript, etc.).

References

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

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

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

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

[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] JNI Get<Type>ArrayElements fails with zero length arrays. https://bugs.openjdk.java.net/browse/JDK-4804447, 2012.

[7] Other Command-Line Options: The -Xcheck:jni Option. https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/clopts002.html, 2012.

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

[9] 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.

[10] Firefox Bug 958706 - Dont́ hide JNI exceptions. https://bugzilla.mozilla.org/show_bug.cgi?id=958706, 2014.

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

[12] 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.

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

[14] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed test generation for effective fault localization. In *ISSTA*, 2010.

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

[16] H. Coles. PIT - mutation testing tool for Java. `http://pitest.org`, 2010.

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

[18] M. Furr and J. S. Foster. Polymorphic type inference for the jni. In *Programming Languages and Systems*, pages 309–324. Springer, 2006.

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

[20] B. Hofer, A. Perez, R. Abreu, and F. Wotawa. On the empirical evaluation of similarity coefficients for spreadsheets fault localization. *Automated Software Engineering*, 22(1):47–74, 2014.

[21] 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.

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

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

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

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

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

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

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

[29] B. Lee, B. Wiedermann, M. Hirzel, R. Grimm, and K.S. McKinley. Jinn: Synthesizing dynamic bug detectors for foreign language interfaces. In *ACM Conference on Program Language Design and Implementation (PLDI)*, 2010.

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

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

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

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

[34] L. Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi. Extended comprehensive study of association measures for fault localization. volume 26, pages 172–219. Wiley Online Library, 2014.

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

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

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

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

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

[40] G. C. Necula, S. McPeak, and W. Weimer. Ccured: Type-safe retrofitting of legacy code. In *ACM SIGPLAN Notices*, volume 37, pages 128–139. ACM, 2002.

[41] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

Summary

Mutation-based Debugging of Real-world Multilingual Programs

오늘날 많은 소프트웨어는 기존 라이브러리(legacy library)를 재사용하거나 각 프로그래밍 언어가 갖는 장점을 최대한 활용하기 위해 다중 언어로 구성되는 경우가 많이 있다. 다중 언어로 프로그램을 개발하게 되면 겪는 어려움 중 하나는 언어 간의 인터페이스 규칙을 정확히 이해하고 개발하지 않으면 고치기 어려운 결함이 발생할 수 있다는 점이다. 다중 언어 프로그램에서 발생하는 결함은 실행 흐름이 언어의 경계선을 넘나들기 때문에 개발자가 결함의 위치를 추적하기가 쉽지 않다. 또한, 기존의 결함 위치추정 방법은 단일 언어로 구성된 프로그램을 대상으로 하기 때문에, 다중 언어 프로그램에서는 정확도가 매우 떨어져서 개발자에게 도움을 주지 못한다.

본 논문에서는 다중 언어 프로그램의 결함을 효과적으로 디버깅할 수 있는 프로그램 변이를 활용한 결함 위치추정 기법인 MUSEUM을 제시한다. MUSEUM은 두 가지 관찰된 사실에 기반한 프로그램 변이 분석을 통해 결함의 위치를 추정한다. 두 개의 관찰된 사실은 변이된 프로그램에 원본 프로그램과 동일한 테스트 케이스를 실행 했을 때 테스트의 결과가 달라지는 경우를 이용하기 위한 이론적 근거를 제공해준다. MUSEUM의 결과는 실행 가능한 구문에 대한 결함 의심도 순위이다. 이 결과는 개발자가 디버깅을 수행할 때 살펴보아야 할 구문의 우선순위를 정하는 데에 도움을 준다. 기존에 제시되었던 단일 언어 프로그램에 대한 변이 규칙만 이용하는 것은 다중 언어 프로그램의 결함 위치를 추적하는데 한계점이 있으므로, 본 논문에서는 프로그램 변이 분석 방법을 다중 언어 프로그램에 효과적으로 적용하기 위하여 다중 언어의 특징을 반영한 새로운 15개의 변이 규칙을 제시하여 프로그램 변이 기반 방식의 정확도를 높였다.

MUSEUM을 평가하기 위해 실제 사용 중인 오픈 소스 프로젝트의 버그 저장소에서 6개의 결함을 수집하였고 해당 프로젝트의 크기는 6KLOC에서 341KLOC까지이다. MUSEUM이 알려주는 각 프로그램 구문의 의심도 순위에 따라 실제 결함의 원인이 되는 구문을 찾아내기 위해 살펴보아야 하는 구문의 수를 측정하였고, 이 결과를 대표적인 결함 위치 추정 방법인 스펙트럼에 기반한 결함 위치추정 기법(SBFL: Spectrum-Based Fault Localization)의 결과와 비교하였다. MUSEUM의 결과에 따르면 최악의 경우에서도 8개의 구문(전체의 4.3%)을 살펴보면 실제 결함 구문을 찾아낼 수 있었지만, SBFL의 결과를 이용하면 최악의 경우에는 모든 구문을 살펴보아야 결함 구문을 찾아낼 수 있음을 실험 결과를 통해 보였다. 특히, MUSEUM은 구문에 대한 의심도 순위를 제공할 뿐만 아니라, 의심도 순위를 추론하기 위해 사용되는 프로그램 변이를 분석하면 실제 결함을 고치는 데 도움을 줄 수 있음을 구체적인 사례를 통해 보였다.

감 사 의 글

석사 과정 2년 동안 저의 연구를 도와주시고 응원해준 분들께 감사의 말씀을 드립니다. 먼저 편안한 집에서 매일 재충전을 할 수 있도록 해주시고 학교에서도 따뜻한 집밥을 먹을 수 있도록 도시락을 싸주신 부모님 감사합니다. 부모님의 든든한 지원을 받은 덕분에 무사히 연구를 마칠 수 있었습니다. 또한, 여러 방면으로 대학원 생활을 지원해준 동생 곽진주에게도 고맙다는 말을 전하고 싶습니다.

연구자로서 갖춰야 할 소양들에 대해 알려주시고 올바른 방향으로 나아갈 수 있도록 지도해주신 김문주 교수님 감사합니다. 소프트웨어 테스팅에 대해 자세히 알지 못했던 저를 받아주시고, 최우수 국제 학술대회인 ASE에 발표된 논문과 SCI급 저널에 제출한 논문의 저자로 참여할 수 있도록 저를 성장시켜 주셨습니다. 부족한 점이 정말 많이 있었지만, 저를 위한 맞춤형 지도를 해주시고 많은 시간을 할애해 주신 점 깊이 감사드립니다. 정직하게 생활하시는 교수님을 보면서, 연구뿐만 아니라 연구 외적으로도 많은 부분을 배울 수 있었습니다. 그리고 공동 연구를 진행한 GIST의 이병철 교수님께도 감사드립니다.

저의 연구에 많은 도움을 주신 홍신 박사님께 감사드립니다. 높은 수준의 연구를 하는 것이 어떤 것인지 직접 배울 수 있었습니다. 같은 주제에 대해 함께 의논할 때 저의 부족함도 많이 느꼈지만, 이는 더 성장할 수 있는 계기가 되었습니다. 구현이 어려울 때마다 손쉽게 해결해준 김윤호 박사과정에게도 감사합니다. 설명하기 어려운 문제 상황도 쉽게 이해하고 해결 방안을 제시해준 덕분에 MUSEUM 도구를 완성할 수 있었습니다. 종종 다가와서 어깨를 다독여주는 점도 큰 힘이 되었습니다.

적응에 어려움을 겪던 석사 1년 차를 함께 보낸 연광흠, 송지영 학생에게 감사합니다. 혼자 적응하려 했으면 버티지 못했을 것입니다. 연구실 후배인 전이루, 김태진 학생에게도 감사합니다. 특히, 저의 실수 때문에 실험을 반복하게 되어도 웃으면서 다시 해준 전이루 학생에게 고맙습니다. 그리고 바쁜 와중에도 도구 구현을 도와준 문석현 졸업생, 연구실에서 겪는 어려움을 정말 빠르게 해결해준 박용배 졸업생, 버그 재현을 도와준 GIST의 고봉석 학생, KAIST에 입학했을 때 저에게 많은 도움을 주신 이명준 선배님에게도 감사합니다.

서로 각자의 연구 분야를 발표하여 공유하고 시야를 넓히는 데 도움을 준 석사 동기 스터디 구성원들에게 감사합니다. 저의 연구를 다른 분야 사람들에게 설명하는 연습할 수 있어서 좋았습니다. 대학원 생활을 하면서 겪는 힘든 일을 털어놓을 수 있었던 동기 전진성, 최종석 형, 대학교 친구 한가람에게 감사합니다. 혼자 고민하기 어려운 일을 함께 고민할 수 있어서 좋았습니다. 항상 같은 자리에서 격려해주고 응원해준 고등학교 친구인 김상섭, 류성근, 이상호, 차승민, 중학교 친구 장광희에게도 진심으로 감사합니다. 늘 바쁘다는 핑계를 대고 연락도 잘 못 하고 만나기도 어려웠지만 이해해주고 배려해주었습니다.

이 외에도, 여기에 적지 못한 많은 분들께 감사드립니다. 이러한 도움이 더욱 빛나도록 앞으로도 최선을 다하겠습니다. 감사합니다.