Information and Software Technology 82 (2017) 80–95

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

# MUSEUM: Debugging real-world multilingual programs using mutation analysis

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

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

A R T I C L E I N F O

*Article history:*
Received 29 July 2016
Revised 12 October 2016
Accepted 13 October 2016
Available online 14 October 2016

*Keywords:*
Debugging
mutation analysis
Language interoperability
Foreign function interface

A B S T R A C T

**Context:** The programming language ecosystem has diversified over the last few decades. Non-trivial programs are likely to be written in more than a single language to take advantage of various control/data abstractions and legacy libraries.

**Objective:** Debugging multilingual bugs is challenging because language interfaces are difficult to use correctly and the scope of fault localization goes beyond language boundaries. To locate the causes of real-world multilingual bugs, this article proposes a mutation-based fault localization technique (MUSEUM).

**Method:** MUSEUM modifies a buggy program systematically with our new mutation operators as well as conventional mutation operators, observes the dynamic behavioral changes in a test suite, and reports suspicious statements. To reduce the analysis cost, MUSEUM selects a subset of mutated programs and test cases.

**Results:** Our empirical evaluation shows that MUSEUM is (i) effective: it identifies the buggy statements as the most suspicious statements for both resolved and unresolved non-trivial bugs in real-world multilingual programming projects; and (ii) efficient: it locates the buggy statements in modest amount of time using multiple machines in parallel. Also, by applying selective mutation analysis (i.e., selecting subsets of mutants and test cases to use), MUSEUM achieves significant speedup with marginal accuracy loss compared to the full mutation analysis.

**Conclusion:** It is concluded that MUSEUM locates real-world multilingual bugs accurately. This result shows that mutation analysis can provide an effective, efficient, and language semantics agnostic analysis on multilingual code. Our light-weight analysis approach would play important roles as programmers write and debug large and complex programs in diverse programming languages.

## 1. Introduction

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

Despite the advance of automated testing techniques for complex real-world programs, debugging multilingual bugs (e.g., a bug

* Corresponding author.
*E-mail addresses:* hongshin@handong.edu (S. Hong), thkwak@kaist.ac.kr (T. Kwak), byeong@gist.ac.kr (B. Lee), podray@kaist.ac.kr (Y. Jeon), bsk@gist.ac.kr (B. Ko), kimyunho@kaist.ac.kr (Y. Kim), moonzoo@cs.kaist.ac.kr (M. Kim).

http://dx.doi.org/10.1016/j.infsof.2016.10.002
0950-5849/© 2016 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

whose cause-effect execution chain crosses language boundaries) in real-world programs is still challenging and consumes significant human effort. For instance, Bug 322222 in the Eclipse bug repository crashes JVMs with a segmentation fault in C as an effect when the program throws an exception in Java as the cause [21]. Locating and fixing this bug took a heroic debugging effort for more than a year from 2009 to 2010 with hundreds of comments from dozens of programmers before the patch went into Eclipse 3.6.1 in September 2010. The existing bug detectors targeting multilingual bugs [20,22,24,25,40,41,44] are not effective in debugging this case, because they can only report violations of predefined interface safety rules, but cannot indicate the location of the bug, especially when the bug does not involve any known safety rule violations explicitly. Moreover, these bug detectors do not scale well to a large number of languages and various kinds of program bugs since they have to deeply analyze the semantics of each language for each kind of bug.

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

Our empirical evaluation on the eight real-world Java/C bugs (Sections 4– 7) demonstrates that MUSEUM locates the bugs in non-trivial real-world multilingual programs far more accurately than the state-of-the-art spectrum based fault localization techniques. MUSEUM identifies the buggy statements as the most suspicious statements for all eight bugs (Section 4). For example, for Bug 322222 in the Eclipse bug repository, MUSEUM indicates the statement at which the developer made a fix as the most suspicious statement among total 3494 candidates (Table 2). Furthermore, one case study on an unresolved Eclipse bug (i.e., an open bug whose fix is not yet made) clearly demonstrates that MUSEUM generates effective information for developers to identify and fix the bug (Section 7).

In summary, this article's contributions are:

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

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

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

## 2. Background and related work

### 2.1. Multilingual bugs

A multilingual program is composed of several pieces of code in different languages that execute each others through language interfaces (e.g., JNI [26] and Python/C). These multilingual programs introduce new classes of programming bugs which obsolete the existing monolingual debugging tools and require much more debugging efforts of programmers than monolingual programs [21,43]. We classify multilingual bugs into *language interface bugs* and *cross-language bugs*.

#### 2.1.1. Language interface bugs

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

- *State constraints* ensure that the runtime system of one language is in a consistent state before transiting to/from a system of another language. For instance, JNI requires that the program is not propagating a Java exception before executing a JNI function from a native method in C.
- *Type constraints* ensure that the programs in different languages exchange valid arguments and return values of expected types at a language boundary. For instance, the `NewStringUTF` function in JNI expects its arguments not to be `NULL` in C.
- *Resource constraints* ensure that the program manages resources correctly. For example, a local reference $l$ to a Java object obtained in a native method $m_1$ should not be reused in another native method $m_2$ since $l$ becomes invalid when $m_1$ terminates [26].

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

#### 2.1.2. Cross-language bugs

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

### 2.2. Mutation-based fault localization (MBFL)

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

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