Test of Exception Handling in Different Java Virtual Machine

Tingting Li [1, a]

[1]School of Software Engineering ,Shanghai Jiao Tong University, China

[a]litingting@sjtu.edu.cn

Keywords: testing, exception handling, java virtual machine

Abstract. Exception Handling in Java and other languages make it possible to raise exception and attaching exception handlers to particular blocks of code. In most research, more people focus on testing of exception handling of programs, Alexandre L. Martins [1] force the execution of exceptional path to improve code coverage analysis, Felipe Ebert [2] study exception handling bugs in Java programs. Exception handling is important to guarantee program quality. However, several studies compare the exception handling result of different Java Virtual Machine(JVM). We focus on studying the exception handling in different Java virtual machine, and develop a tool to automatically insert exception code to test the consistency of Java virtual machine exception handling.

Introduction

Exception Handling in Java and other languages make it possible to raise exception and attaching exception handlers to particular blocks of code. Some research focus on testing of exception handling of programs, Alexandre L. Martins [1] force the execution of exceptional path to improve code coverage analysis, Felipe Ebert [2] study exception handling bugs in Java programs.

However, several studies compare the exception handling result of different Java Virtual Machine(JVM). In this work we present an exploratory study on exception handling in different JVM.

High reliability of software is important to software quality, while high reliability always require programs with complete exception handling constructs. Java exceptions can be raised implicitly or explicitly. Many developers use exception handling construct to satisfy their requirements, but sadly lacks experience of exception handling knowledge [3]. These make it difficult in documenting and testing exception handling of program. To improve the robustness of programs, researchers develop many testing methods of exception handlings. Changbin JI et al. [4] present mutant operators for Java exception handling constructs.

However, no one takes focus on the implementation of exception handling in JVM. Different JVM may have different response with the same exception conditions. Although exception handling construct behavior excellent in current JVM, but it will bring unpredictable result if the other JVM's handles exception differently.

Apparently it is necessary to validate the consistency of exception handling process in different JVM. Our work determines to validate the consistency with exception code instrument method.

This work can be divide into three parts. First trace the execution path of programs, second instrument exception throw and handling code into execution path randomly, at last compare the execution result in different JVM. In our work, Jimple code play an important role. Class file will first translate into Jimple, then collect execution path and instrument exception code block (in Jimple), at last translate result Jimple code into class file and run it with different JVM.

This paper is organized as follows: Section 2 presents related backgrounds; Section 3 presents our method with instrumentation technique; Section 4 illustrate the result of our tool; Section 5 shows Conclusion and our future work.

Background

This section describes background of Java Exception Handling and introduction to Jimple. As exception handling is important in software quality, it is necessary to validate the consistency of exception handling in different JVM [1][5].

Java Exception Handling Structure. An exception is said to be thrown from the point where it occurred and is said to be caught at the point to which control is transferred. There are three kind of exception handling structure in Java.

try/catch/finally block. Catch and finally block is optional, one try block can follow by zero or more catch block and/or zero or one finally block. Try block may throw exceptions, catch block can catch exceptions that compatible its catchable exception classes or it’s subclasses. Finally block usually used to free resources while it always executed whether exceptions are thrown or not.

throws clause. Throws clause follow with the declaration of the method parameters, it is used to declare which exception classes can be explicitly thrown by the method. It is not necessary to handle with these exceptions.

throw statement. It usually appears inside a decision statement to inform under which conditions an exception should be thrown [1].

Methodology

Trace execution path.

1) Insert trace jimple code into target file

We choose to insert definition code before the main method. The reason is complicated, at first we try to insert it before initiation method, but some class’s initiation is private and it can be invoke in initiation method.

Cause every method we choose to instrument has main method, and the definition of main method are the same, this is excellent for instrumenting trace method here. The definition of main is like Fig.1.

```
    public static void main(java.lang.String[])
    {
       ……
    }
```

Figure .1 Main method in Java

This is instrumented trace definition code. It will output the trace of the execution path, including the execution method name and label name. This information will be important for us to instrument exception code later as Fig.2.

```
public static void instrument(java.lang.String)
   {
      java.lang.String r0, $r6, $r10;
      java.io.PrintStream $r1;
      java.lang.StringBuilder $r2, $r7, $r8, $r9;
      java.lang.Thread $r3;
      java.lang.StackTraceElement[] $r4;
      java.lang.StackTraceElement $r5;

      r0 := @parameter0: java.lang.String;
      $r1 = <java.lang.System: java.io.PrintStream out>;
      $r2 = new java.lang.StringBuilder;
      specialinvoke $r2.<java.lang.StringBuilder: void <init>(java.lang.String)>("==>");
      $r3 = staticinvoke <java.lang.Thread: java.lang.Thread currentThread()>();
      $r4 = virtualinvoke $r3.<java.lang.Thread: java.lang.StackTraceElement[] getStackTrace()>();
      $r5 = $r4[2];
      $r6 = virtualinvoke $r5.<java.lang.StackTraceElement: java.lang.String getMethodName()>();
      $r7 = virtualinvoke $r2.<java.lang.StringBuilder: java.lang.StringBuilder
append(java.lang.String)>($r6);
```

```
        $r8 = virtualinvoke $r7.<java.lang.StringBuilder: java.lang.StringBuilder
append(java.lang.String)>(":");
        $r9 = virtualinvoke $r8.<java.lang.StringBuilder: java.lang.StringBuilder
append(java.lang.String)>(r0);
        $r10 = virtualinvoke $r9.<java.lang.StringBuilder: java.lang.String toString()>();
        virtualinvoke $r1.<java.io.PrintStream: void println(java.lang.String)>($r10);
        return;
    }
```

Figure .2 Jimple code example

2) Insert trace invoke into Jimple

This is the trace invoke code in Jimple, it will be insert after every label, except exception caught definition. ClassName will be replace by the class name of this file, and the argument label will be replace with current label and it's number as Fig.3.

```
staticinvoke <ClassName: void instrument(java.lang.String)>("label0:");
```

Figure .3 Instrument code

3) Change Jimple into Class and collect execution path

Use soot change jimple into class, then run class file with JVM, collect the console output.

Insert exception code.

1) Find every execution path

This step will collect execution path of java class. We define LabelTrace class to store the trace, it contains method name and a list of label name. This will be use to locate position of exception instrument.

2) Random insert exception code (pseudo code)

Random choose one/some execution point, insert one/some exception code, produce many different jimple test files.

Exception code can be simple throw exception and some more complicate condition.

3) change Jimple to Class

With the help of soot, change every jimple result into class file. Then collect every success result class file.

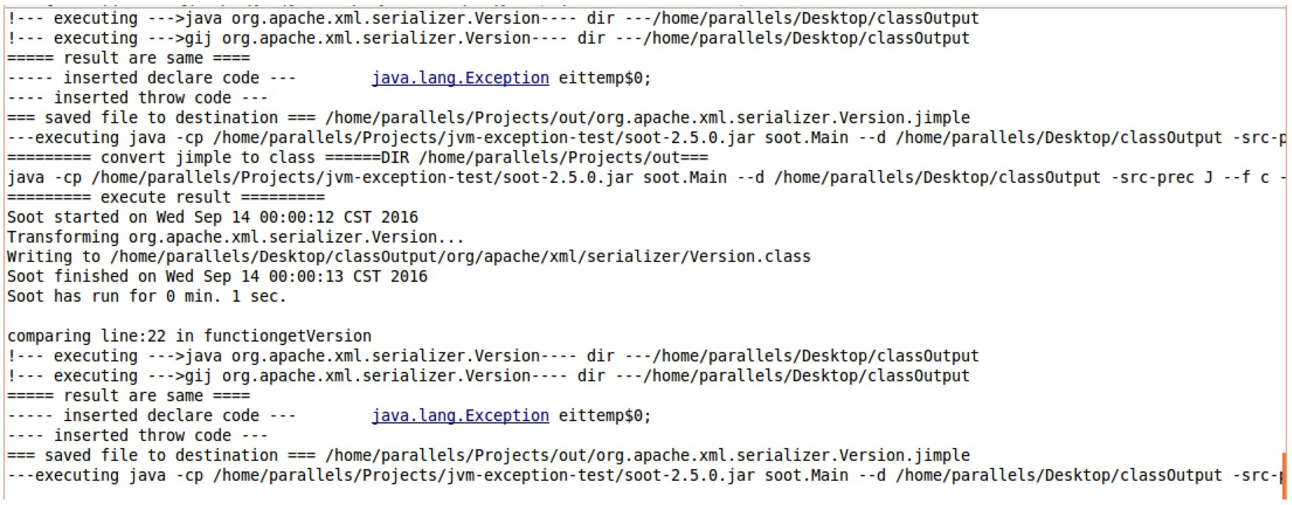
Run in different java virtual machine. Choose different java virtual machine to run testing code. In our paper, we choose OpenJdk and Gij run the testing code.

With the exception code injecting program, we insert one exception one time, and then change files back to class file, then run codes on different Java virtual machine and compare the result.

Result and Summary

In experiment, we develop a tool can run different testing in different Java virtual machine automatically. With an input of jar file, the program will extract it and change it into jimples file, then find main-method file. Insert exception code automatically and package it back to class. At last run .class in different Java virtual machine and compare the results. The output is as Fig.4.

In future work, we will try to make this tool more complete. Will consider add some more strategy on exception insert positioning, lift insert exception types and complexity, run on more Java virtual machine and so on.



```
!--- executing --->java org.apache.xml.serializer.Version---- dir ---/home/parallels/Desktop/classOutput
!--- executing --->gij org.apache.xml.serializer.Version---- dir ---/home/parallels/Desktop/classOutput
===== result are same ====
----- inserted declare code ---        java.lang.Exception eittemp$0;
---- inserted throw code ---
=== saved file to destination === /home/parallels/Projects/out/org.apache.xml.serializer.Version.jimple
---executing java -cp /home/parallels/Projects/jvm-exception-test/soot-2.5.0.jar soot.Main --d /home/parallels/Desktop/classOutput -src-p
========= convert jimple to class ======DIR /home/parallels/Projects/out===
java -cp /home/parallels/Projects/jvm-exception-test/soot-2.5.0.jar soot.Main --d /home/parallels/Desktop/classOutput -src-prec J --f c -
========= execute result =========
Soot started on Wed Sep 14 00:00:12 CST 2016
Transforming org.apache.xml.serializer.Version...
Writing to /home/parallels/Desktop/classOutput/org/apache/xml/serializer/Version.class
Soot finished on Wed Sep 14 00:00:13 CST 2016
Soot has run for 0 min. 1 sec.

comparing line:22 in functiongetVersion
!--- executing --->java org.apache.xml.serializer.Version---- dir ---/home/parallels/Desktop/classOutput
!--- executing --->gij org.apache.xml.serializer.Version---- dir ---/home/parallels/Desktop/classOutput
===== result are same ====
----- inserted declare code ---        java.lang.Exception eittemp$0;
---- inserted throw code ---
=== saved file to destination === /home/parallels/Projects/out/org.apache.xml.serializer.Version.jimple
---executing java -cp /home/parallels/Projects/jvm-exception-test/soot-2.5.0.jar soot.Main --d /home/parallels/Desktop/classOutput -src-
```

Figure .4 Program run result

References

[1] Martins, Alexandre L., Simone Hanazumi, and Ana CV De Melo. "Testing Java Exceptions: An Instrumentation Technique." Computer Software and Applications Conference Workshops (COMPSACW), 2014 IEEE 38th International. IEEE, 2014.

[2] Ebert, Felipe, Fernando Castor, and Alexander Serebrenik. "An exploratory study on exception handling bugs in Java programs." Journal of Systems and Software 106 (2015): 82-101.

[3] Robillard, Martin P., and Gail C. Murphy. "Designing robust Java programs with exceptions." ACM SIGSOFT Software Engineering Notes. Vol. 25. No. 6. ACM, 2000.

[4] Ji, Changbin, et al. "A new mutation analysis method for testing Java exception handling." Computer Software and Applications Conference, 2009. COMPSAC'09. 33rd Annual IEEE International. Vol. 2. IEEE, 2009.

[5] Deitel, Paul, and Harvey Deitel. Java How to program. Prentice Hall Press, 2011.