Exercising Java Exceptions Using Java Pathfinder and Program Instrumentation

Alexandre Locci Martins, Simone Hanazumi, and Ana C.V. de Melo

Department of Computer Science, University of São Paulo, São Paulo, Brazil
`{alelocci,hanazumi,acvm}@ime.usp.br`

Abstract. This paper presents an instrumentation technique to exercise the exceptional behavior of Java programs using the Java Pathfinder model checker. First, programs statements that are potentially able to throw exceptions are identified, and then a set of Java instructions are inserted into programs source code to throw exceptions, producing an instrumented program. Java Pathfinder is applied over this instrumented code to perform multiple executions of the system under verification and cover the program exceptional behavior. Additionally, our technique proposes a prototype of a new Java Pathfinder class to verify and test Java exceptions.

1 Introduction

Testing and *formal verification* are the main activities used by practitioners to certify the software quality [1,11]. To assess the software quality, we compare the observed software behavior and the software specification. If the software behavior and its specification do not match, then there is a strong indication that a software error was found. Therefore, testing and formal verification techniques provide strategies to evaluate the software quality.

Testing consists in dynamically analyzing the software behavior to compare the observed behavior at the implementation level against the expected behavior specified at the design level. Formal verification [3,6], in turn, corresponds to mathematically prove that a given system is in accordance with its design specifications. These activities use different approaches to certify software quality and some recent studies have tried to crossover both techniques to combine their strong features and minimize their weaknesses[7].

An exception is any unexpected or undesirable behavior detected by software or hardware that requires a special handling [15]. Whenever an exception is raised in a program, a change on the program normal execution path [14] occurs, driving to the so called *exceptional path.* Today an expressive number of Java programs have at least one kind of exception handling block statement [4,5]. Although the importance of analyzing exception handling structures is recognized [16,17,4,15,5], most researches on combining testing and verification are devoted to the normal behavior of software [12,16,17,2,9].

If testing or verification techniques only focus on the normal software behavior, an expressive number of potential execution paths may not be exercised. As a result, a number of software behaviors are not checked in testing or they are neglected as potential model elements in verification.

Aiming to contribute to the research of programs exceptional behavior testing and verification, our work suggests a code instrumentation technique that guides the Java Pathfinder (JPF) [13] verification to exercise exceptional paths of Java programs. We *force* programs to exercise their exceptional paths via source code instructions that are used by JPF to throw exceptions and return to predefined execution states.

In the instrumentation technique we address the exceptions given by programmers. The application of this technique with the JPF execution gives to programmers the opportunity to observe his/her program exceptional behavior when testing or checking properties to ensure that the exceptional paths are also reached.

This paper is organized as follows: Section 2 describes the main concepts that appear in our work; Section 3 presents the proposed instrumentation technique to exercise the exceptional paths of a program; Section 4 presents the related work; and, Section 5 shows the conclusion and future work.

2 Background

This section describes the main Java exceptions statements used in the present work and a brief overview of the JPF.

2.1 Handling Exceptions in Java

Java is one of the first modern languages in which exception handlers were presented as primary design elements; all exceptions are objects.

There are three basic syntactic structures that provide exception handling. The first one is the *try/catch/finally* blocks that are used to enclose the statement that may throw exception (try), provide the correct treatment for an exception thrown (catch) and release the used resources (finally). The second one is the *throws* clause used to indicate that a method may throw an exception. The last one is the *throw* statement used by the programmer to intentionally throw an exception at run time.

When an exception is thrown at run time the control flow is skipped to the appropriate syntactic handling structure (catch/finally) or the execution is halted. In the *try/catch/finally* blocks, whenever an exception is thrown in the *try* block, the control-flow is deviated to a *catch* and/or *finally* block and goes ahead, not returning to the previous statement. Also, when a *throw* statement is used, the method is killed and the flow goes up with the stack until a handling statement is found or the program stops. All these cases show that an exception (handled or not) changes the program behavior and must be tested/verified.

2.2 Java Pathfinder (JPF) and the *Verify* Class

Java Pathfinder (JPF) is a model checker written in Java and it is used to check program properties directly on the Java *bytecode* [13]. It performs an exploration of the state space to identify violations of properties. If a property violation is detected, then it presents values at state space that are responsible for reaching such violation, a counter-example. Otherwise, it can conclude that the program satisfies the proposed property.

Although JPF does not work directly with exceptions, it provides the *Verify* class to exercise an explicit interval of values, instead of leaving to JPF the task of choosing values to use in programs verification (e.g. sequence of integer values or boolean values).

The *Verify* class can be used similarly to exercise exceptional paths by combining its methods with the *throw* statement [2,9,13]. However, using this instrumentation, the start point of the exceptional path could be no longer the exact point in which the exceptions are really raised.

In the present work, the instrumentation technique preserves the capabilities of the JPF *Verify*. The difference is that our technique focuses on the analysis of the instrumented program behavior, and the instrumented code that actually throws an exception is the starting point of an exceptional path. Moreover, our instrumentation technique can be applied to Java programs and used to exercise exceptional paths in other test and verification tools besides the JPF model checker [10].

3 Source Code Instrumentation

In our work, we propose an alternative construction to the JPF *Verify* class, the *VerifyEx* class, and an alternative instrumentation approach to exercise exceptions. The following sections describe this new class and how to use it to exercise program exceptional paths.

3.1 The VerifyEx Class

The *VerifyEx* is a prototype Java class designed to exercise software exception behavior. It may be integrated directly in the software code under test or used as a library (.jar). This class provides methods to identify if one exception was, or was not, thrown. If the class detects that an exception was not thrown, it throws the exception to exercise the software exceptional behavior. In addition, the VerifyEx class stores the sequence of exceptions that were thrown to report to the user.

Listing 1.1. VerifyEx Class Example

```
1   public class ExampleClass {
2      public static void main(String[] args) {
3          try{
4              if(VerifyEx.exception("1")){
5                  throw new Exception();
6              }
7          }catch(Exception e){
8              System.out.println("Exception 1");
9          }
10         if(ExceptionList.exceptionList.size() < 1) ExampleClass.main(args)
               ;
11     }
12  }
13  class VerifyEx { ...  }
14  class ExceptionList {...  }
```

Line 4 (Listing 1.1) presents an example of instrumentation statement: the *exception* method receives a string as a parameter.

Different from *Verify*, this string is a label and can be used to identify the line where the exception was thrown, the sequence of thrown exceptions or any other kind of information that the developer needs for code analysis (testing).

The second difference is presented at line 10. The class attribute *exceptionList* is used to define if all exceptions have been thrown. This attribute stores the labels passed as parameters to the *exception* method. The *Verify* can be used to implement this functionality either, but this implies in more statements to be inserted directly into the source code. And this could make the code less readable.

After the exception at line 5 has been thrown, label `1` is added to the list and checked at line 10. If the list size is equal to 1, all executions are stopped. Otherwise, one or more exceptions were not exercised yet and the *reboot* process is started to cover the other values.

Although the *Verify* class can also execute a kind of reboot process, it does not allow the developer to have control over the process as with the *VerifyEx* class. For instance, the Verify class does not interrupt the program execution, even when the exceptional behavior has been exercised. This results in the coverage of a set of statements that does not have relation with the exceptional structure and may be accessed without any exception throw. The new class, VerifyEx, has the advantage of being more flexible because the reboot execution point can be programmed to anywhere in code, instead of always going to the `main` method. And the VerifyEx reboot process allows the interruption of the program execution after the exceptional statements have been exercised.This feature can drive the JPF execution to be more efficient in certain cases since the exceptional path does not need to start at the `main` method.

A full example of using the *VerifyEx* class is given in Section 3.3.

3.2 Instrumenting Exceptions Statements

The source code is instrumented by using a method that allows the identification of the lines that may throw an exception. This method is based on the activation/deactivation exception concept that was presented by Sinha and Harrold [17]. An exception object is activated if it is raised with a *throw* command or in a *try* block and deactivated when it is treated or a *null* value is given. Now, a sequence of actions is implemented:

1. *Build exception activation/deactivation model*: we create an abstract model that identifies the statements that may activate exceptions and the statements that may deactivate exceptions in the source code. After that, we establish the relation among these elements, obtaining the control flow model of exception activation/deactivation statements.
2. *Determine the line to insert the instrumentation code*:
   – *try* blocks: insert instrumentation code at the first line of the *try* block. This decision is based on the fact that the scope of *try* block is a potential place to activate an exception.
   – *throw*: the instrumentation is inserted immediately before this statement.
   – *throws*: the instrumentation is inserted at the method first line.
3. *Inserting the VerifyEx*: we insert into the source code the statement that is able to identify and throw the suitable exception.

All these steps are automatically done by a tool that takes a program source code as input and returns the instrumented source code. To do so, the tool recognizes a program statement that is responsible for activating and deactivating exceptions, finds the places where the instrumentation code must be inserted, and inserts the instrumentation code to activate exceptions and reboot the program path. An example in Section 3.3 illustrates the instrumentation technique.

3.3 Instrumenting a Code and Running JPF

To illustrate how the exceptional paths are exercised with the proposed instrumentation, which uses the *VerifyEx* class, we use the *CheckValue* program (Listing 1.2). We first present JPF running on the non-instrumented code to show how far it goes on the exercise of exceptions. Then, we apply the presented instrumentation technique to instrument this code and run JPF on the instrumented code to summarize the technique effects on program paths.

Listing 1.2. *CheckValue* Program - original code

```
public class CheckValue {
 public static void main(String[] args) {
  try{
   int num = (int)(Math.random()*200);
   if (num == 95) throw new Exception();
    checkValue(num);
    s.o.p("Valid Values");
   }catch(ArithmeticException e){
     CheckValue.exitError("Value below the limit");
   }catch(ArrayIndexOutOfBoundsException e){
     CheckValue.exitError("Value over the limit");
   }catch(Exception e){
     CheckValue.exitError("Reserved");
   }finally{
     s.o.p("Value Checked");
   }
     s.o.p("Stop");
   }
   private static void checkValue(int num) throws ArithmeticException,
    ArrayIndexOutOfBoundsException{
     ValueInferior    vi = new ValueInferior();
     CheckUpper       vu = new CheckUpper();
      vi.inferiorValue(num);
      vu.upperValue(num);
   }
   private static void exitError(String str){
    s.o.p(str);
   }
}
class ValueInferior{
    public void inferiorValue(int a){
        if(a < 10){
            throw new ArithmeticException();
        }
        s.o.p("Inferior Limit Checked");
    }
}
class CheckUpper{
    public void upperValue(int a){
        if(a > 100){
            throw new ArrayIndexOutOfBoundsException();
        }
        s.o.p(Upper Limit Checked");
    }
}
```

3.4 Running JPF on the Non-instrumented Code

Despite the broad use of JPF to model check programs, it mainly operates on the normal flow of control and data. For example, if the *CheckValue* program is checked by JPF, only the normal paths are exercised and no exception is thrown intentionally (Listing 1.3). As a result, JPF performs only one execution of the program *CheckValue*.

Listing 1.3. JPF traces

```
JavaPathfinder v6.0 - (C) RIACS/NASA Ames Research Center
======================================================= system under test
application: CheckValue.java

========================================== search started: 2/27/14 4:41 PM
Inferior Limit Checked
Upper Limit Checked
Valid Values
Value Checked
Stop

======================================================= results
no errors detected

======================================================= statistics
...
============================================ search finished: 2/27/14 4:41 PM
```

However, a model checker can be used to exercise the whole behavior of programs, including the exceptional behavior. If so, properties on the exceptional behavior are better covered and broader test cases are exercised. The forthcoming section shows JPF running on an instrumented code of the *CheckValue* program.

3.5 Running JPF on the Instrumented Code

Here we present an example of how to instrument a code with the *VerifyEx* class. We use for this purpose the *CheckValue* program (instrumented code in Listing 1.4). To illustrate how the exceptional paths are exercised with this instrumentation, we present the results obtained using the tool JPF, Listing 1.5.

Listing 1.4. CheckValue Program - instrumented code

```
public class CheckValue {
 public static void main(String[] args) {
   try{
     if (VerifyEx.exception("1")) throw new Exception();
     int num = (int)(Math.random()*200);
     checkValue(num);
     s.o.p("Valid Values");
   }catch(ArithmeticException e){
      CheckValue.exitError("Value below the limit");
   }catch(ArrayIndexOutOfBoundsException e){
      CheckValue.exitError("Value over the limit");
   }catch(Exception e){
      CheckValue.exitError("Reserved");
   }finally{
      s.o.p("Value Checked");
   }
      s.o.p("Stop");
      if(ExceptionList.exceptionList.size() < 3) CheckValue.main(args);
 }
 private static void checkValue(int num) throws ArithmeticException,
   ArrayIndexOutOfBoundsException{
     ValueInferior vi = new ValueInferior ();
     CheckUpper vu = new CheckUpper ();
     vi. inferiorValue (num);
     vu. upperValue (num);
 }
 private static void exitError(String str){
   s.o.p(str);
 }
}
class ValueInferior{
  public void inferiorValue(int a) throws ArithmeticException{
   if(VerifyEx.exception("2")){
     throw new ArithmeticException();
   }
   s.o.p("Inferior Limit Checked");
 }
}
class CheckUpper{
 public void upperValue(int a) throws ArrayIndexOutOfBoundsException{
   if(VerifyEx.exception("3")){
     String msg = "Invalid";
     throw new ArrayIndexOutOfBoundsException(msg);
   }
   s.o.p("Upper Limit Checked");
 }
}
```

In the *CheckValue* program (Listing 1.4), we have three possible exceptional paths. For each path, one of these three types of exceptions may be thrown: (1) *Exception*; (2) *ArithmeticException*; and (3) *ArrayIndexOutOfBoundsException*. To exercise all these exceptional paths, we instrument the code using the *VerifyEx* class, following the steps of Section 3.2:

1. *Build exception activation/deactivation model*: in *CheckValue* program we have identified in its abstract model three exceptions activation/deactivation statements. They are located at lines:
   – (3, 12): reference to the *try* block where an *Exception* may be thrown, treatment of *Exception* in a *catch* block;
   – (32, 8): reference to *ArithmeticException* throw, treatment of *ArithmeticException* in a *catch* block; and
   – (40, 10): reference to *ArrayIndexOutOfBoundsException* throw, treatment of *ArrayIndexOutOfBoundsException* in a *catch* block.
2. *Determine the line to insert the instrumentation code*: according to the identified exceptions activation/deactivation, the appropriate instrumentation statements are:
   – Line 4: inside a *try* block;
   – Line 33: first line of a method that throws exceptions; and
   – Line 41: first line of a method that throws exceptions.
3. *Inserting the VerifyEx*: Listing 1.4 presents the instrumentation code inserted at lines 4, 33 and 41, as we have determined in the previous step.

In addition, the last element to be considered in the instrumentation is the possibility of performing a *reboot* of the code after the release of a given exception that involves the program execution halt. To do this, we insert a *VerifyEx* reboot statement just before the halt statement. In the example (Listing 1.4), we put a reboot statement at line 18 to guarantee that the three exceptional paths of the program were exercised by JPF. After the execution of all the abnormal paths, the program execution returns to its `main` method.

The JPF output of the instrumented *CheckValue* code is presented in Listing 1.5. The exceptional paths of the program are exercised in the following order: (1) *Exception*: lines 6-8; (2) *ArithmeticException*: lines 9-11; and (3) *ArrayIndexOutOfBoundsException*: lines 13-15.

Listing 1.5. CheckValue Program - JPF output

```
1   JavaPathfinder v6.0 - (C) RIACS/NASA Ames Research Center
2  ============================================= system under test
3  application: CheckValue.java
4
5  ========================================== search started: 2/27/14 4:44
       PM
6  Reserved
7  Value Checked
8  Stop
9  Value below the limit
10 Value Checked
11 Stop
12 Inferior Limit Checked
```

```
Value over the limit
Value Checked
Stop

========================================================= results
no errors detected

================================================== statistics
...
================================== search finished: 2/27/14 4:44 PM
```

With this *CheckValue* example, we observe that the *VerifyEx* class allows the automation of the process of exercising exceptional paths (Listing 1.5), using source code instrumentation (Listing 1.4). This instrumentation allows multiple executions of a program by the use of verification or testing tools. For each run, the chosen tool provides a possible exception throw. For each thrown exception we have an exceptional path that is exercised. Thus, any check that was previously restricted to program normal paths is applied to exceptional paths. Additionally, the *VerifyEx* class allows overcoming situations in which the execution of a given path leads to a program halt. This is done through a simple recursive call (*reboot* process) that is inserted before the instruction that causes the program halt. Comparing these results to traditional programs exceptional behavior testing/verification, we can conclude that our technique provides a more effective and faster manner to test/verify programs abnormal execution paths.

4 Related Work

The main researches related to our work are briefly described in this section. They are divided in two categories: researches that dealt with the test of system exceptional behavior, and researches that used the JPF Verify class (Section 2.2) to instrument the code in order to exercise new execution paths.

4.1 Testing Programs Exceptional Behaviors

Ji et al. [8] presented mutant operators for Java exception handling constructs. They are used with mutation testing to help in the evaluation and improvement of test sets used to ensure that these constructs were arranged properly. Zhang and Elbaum [19] dealt with the testing of exceptional paths in programs that use external resources. In their approach, the test suite executes a broader space of exceptional behavior associated with an external resource that is used by the program under test. Consequently, most of program faults and abnormal behaviors can be detected in an automated manner.

Our work intends to help the test of system exceptional behavior as the above works. But our approach uses code instrumentation to exercise the exceptional paths of a program. When exercising these paths using a testing tool, the software tester can use the collected data to ensure, for instance, that an exception was handled in a correct manner and that the exception was raised accordingly.

4.2 Code Instrumentation with JPF Verify Class

Staats [18] rebuilt the *Verify* class to allow JPF working with the *modified condition/decision coverage* (MC/DC). The code instrumentation presented in his work allows the automatic generation of test cases and its execution with the use of JPF to cover the MC/DC criterion. Since the MC/DC addresses the normal behavior, the exception statements are not instrumented. Li et al. [9] presented a technique to verify whether Java exceptions are used in a safe way. This technique combines a static analysis of the Java source code to identify where exceptions can be raised, with model checking technique visiting all possible exception execution paths. The exercise of exceptional paths is done by using the JPF *Verify* class to make JPF performing multiple executions of the same program. For each potential exception, all possible combinations of the normal or exceptional paths are exercised. Using static analysis, they can automatically find exceptions and insert instrumentation code to exercise the exceptional paths. The main contributions of this work are pointing out the possibility of using JPF to check exceptional paths and using the *Verify* class to instrument code. Artho and Sommer [2] presented a fault model for model checking networked programs. For that, the network program is modeled to eliminate undesired behaviors, such as deadlock. Each exceptional statement in the source code is instrumented by using JPF *Verify* class. The program is then checked by the JPF. The exceptional behavior is observed to determine how the program deals with faults on communications caught by the exceptional structures.

The works by Artho and Sommer [2] and Li et al. [9] address the feasibility of using JPF *Verify* class to exercise exceptions. The current work has a similar objective but differs from them in two aspects. First, we suggest a set of changes to the JPF *Verify* class to better handle exceptional paths. With our approach, the state to return after executing an exception path can be chosen by programmers. Second, our prototype class can be used in any testing or verification tool instead of being restricted to JPF execution.

5 Conclusions and Future Work

Exercising the exceptional behavior of programs is not an easy task. Most of the testing and verification tools were designed for handling normal control flow rather than exceptional flow.

This work proposes an approach to exercise exceptional paths by combining code instrumentation and the use of a testing and/or verification tool. In this paper we used the verification tool JPF, but we could have used other tools like the JUnit [10]. The result achieved here is a technique that: (1) exercises exceptional paths; (2) allows the developer to build logs of sequences of thrown exceptions and; (3) implements the reboot process that allows the programmer to choose which method should be called to perform a new execution of the program under verification. The latter streamlines the process of exercising exceptional behavior paths since it can ignore all code regions that have already been explored. Another contribution is the possibility of using our technique to instrument Java source code that can be executed in any kind of testing and verification tool.

VerifyEx allows the automation of the process of exercising exceptional paths using source code instrumentation. This instrumentation allows multiple executions of a program by the use of verification or testing tools. For each run, the chosen tool provides a possible exception throw. For each thrown exception we have an exceptional path that is exercised. Thus, any check that was previously restricted to the normal path is applied to the exceptional paths. Additionally, the *VerifyEx* class allows overcoming situations in which the execution of a given path leads to a program halt through a simple recursive call, *reboot* process, before the instruction that causes the halt program.

In addition, the last element to be considered is the possibility of performing a *reboot* of the code after the release of a given exception that involves the program execution halt. The task of testing or verifying the exceptional paths of programs with halt statements becomes exhaustive because the program will stop whenever a halt statement is executed (a new execution must be set for each path). A simple solution for this problem is to insert a *VerifyEx* reboot statement just before the halt statement. Then this halt statement will not be executed and the *model checker* goes ahead exercising other exceptional paths.

All the instrumentation process is supported by a tool that automatically do the instrumentation. In practice, developers may wish to initiate a deviation from a normal to an exceptional path before or after a particular instruction, it depends on what he/she wants to observe in testing or verifying a program. To automate the whole instrumentation process, however, a decision on where to insert instrumentation code to exercise exceptions activation and deactivation was taken. In a future work we will incorporate a decision process which aims to improve the semi-automatic choice of the points at which exceptions could be thrown. Finally, studies will be conducted to verify if we can use the listener *ExceptionInsert* available in the JPF. If this is possible, exceptions will be thrown without direct instrumentation of the source code.

Acknowledgments. This project has been funded by the Ministry of Education Research Agency (CAPES- Brazil), and the State of São Paulo Research Foundation (FAPESP) - Processes: 2011/01928-1, 2012/23767-2, 2013/22317-6 and 2013/25309-4.

References

1. Ammann, P., Offutt, J.: Introduction to Software Testing, 1st edn. Cambridge University Press, New York (2008)
2. Artho, C., Sommer, C., Honiden, S.: Model Checking Networked Programs in the Presence of Transmission Failures. In: Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, TASE 2007, pp. 219–228. IEEE Computer Society, Washington, DC (2007)
3. Baier, C., Katoen, J.-P.: Principles of Model Checking (Representation and Mind Series). The MIT Press (2008)

4. Cabral, B., Marques, P.: Exception Handling: A Field Study in Java and .NET. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 151–175. Springer, Heidelberg (2007)
5. Dúulaigh, K.O., Power, J.F., Clarke, P.J.: Measurement of Exception-Handling Code: An Exploratory Study. In: 5th International Workshop on Exception Handling, Zurich, Switzerland (June 9, 2012)
6. Fisher, M.: An Introduction to Practical formal Methods Using Temporal Logic, 1st edn. Wiley (2011)
7. Fraser, G., Wotawa, F., Ammann, P.E.: Testing with model checkers: a survey. Softw. Test. Verif. Reliab. 19, 215–261 (2009)
8. Ji, C., Chen, Z., Xu, B., Wang, Z.: A New Mutation Analysis Method for Testing Java Exception Handling. In: 2012 IEEE 36th Annual Computer Software and Applications Conference, vol. 2, pp. 556–561 (2009)
9. Li, X., Hoover, H.J., Rudnicki, P.: Towards Automatic Exception Safety Verification. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 396–411. Springer, Heidelberg (2006)
10. Martins, A.L., Hanazumi, S., de Melo, A.C.V.: Testing Java Exceptions: An Instrumentation Technique. In: Proceedings of the 2014 IEEE 38th Annual Computer Software and Applications Conference Workshops, COMPSACW 2014, IEEE Computer Society, Väasteråas (to appear, 2014)
11. Mathur, A.P.: Foundations of Software Testing, 1st edn. Addison-Wesley Professional (2008)
12. Nagar, P., Soni, N.: Optimizing Program-States using Exception-handling Constructs in Java. International Journal of Engineering Science & Advanced Technology 2(2), 192–199 (2012)
13. NASA, What is JPF, `http://babelfish.arc.nasa.gov/trac/jpf/wiki/intro/what_is_jpf` (accessed: November 25, 2012)
14. Oracle. What Is an Exception? `http://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html` (accessed: November 25, 2012)
15. Sebesta, R.W.: Concepts Programming Languages (7th ed.). Addison-Wesley Longman Publishing Co., Inc., Boston (2005)
16. Sinha, S., Harrold, M.J.: Analysis of Programs with Exception-Handling Constructs. In: ICSM 1998: Proceedings of the International Conference on Software Maintenance, p. 348. IEEE Computer Society, USA (1998)
17. Sinha, S., Harrold, M.J.: Analysis and Testing of Programs with Exception Handling Constructs. IEEE Trans. Softw. Eng. 26(9), 849–871 (2000)
18. Staats, M.: Towards a Framework for Generating Tests to Satisfy Complex Code Coverage in Java Pathfinder. In: Proceedings of NASA Formal Methods Symposium 2009, p. 116 (2009)
19. Zhang, P., Elbaum, S.: Amplifying tests to validate exception handling code. In: Proceedings of the 2012 International Conference on Software Engineering, ICSE 2012, pp. 595–605. IEEE Press, Piscataway (2012)