Acta Cybernetica **21** (2014) 331–352.

Runtime Exception Detection in Java Programs Using Symbolic Execution*

István Kádár†, Péter Hegedűs†, and Rudolf Ferenc†

Abstract

Most of the runtime failures of a software system can be revealed during test execution only, which has a very high cost. In Java programs, runtime failures are manifested as unhandled runtime exceptions.

In this paper we present an approach and tool for detecting runtime exceptions in Java programs without having to execute tests on the software. We use the symbolic execution technique to implement the approach. By executing the methods of the program symbolically we can determine those execution branches that throw exceptions. Our algorithm is able to generate concrete test inputs also that cause the program to fail in runtime.

We used the Symbolic PathFinder extension of the Java PathFinder as the symbolic execution engine. Besides small example codes we evaluated our algorithm on three open source systems: jEdit, ArgoUML, and log4j. We found multiple errors in the *log4j* system that were also reported as real bugs in its bug tracking system.

Keywords: Java runtime exception, symbolic execution, rule checking

1 Introduction

Nowadays, it is a big challenge of the software engineering to produce huge, reliable and robust software systems. About 40% of the total development costs go for testing [13], and the maintenance activities, particularly bug fixing of the system also require a considerable amount of resources [20]. Our purpose is to develop a new method and tool, which supports this phase of the software engineering lifecycle with detecting runtime exceptions in Java programs, and finding dangerous parts in the source code, that could behave as time-bombs during further development. The analysis will be done without executing the program in a real environment.

Runtime exceptions in the Java programming language are the instances of class java.lang.RuntimeException, which represent a sort of runtime error, for example

*This research was supported by the Hungarian national grant GOP-1.1.1-11-2011-0038 and the TÁMOP 4.2.4. A/2-11-1-2012-0001 European grant.

†University of Szeged, Department of Software Engineering Árpád tér 2. H-6720 Szeged, Hungary, E-mail: {ikadar|hpeter|ferenc}@inf.u-szeged.hu

DOI: 10.14232/actacyb.21.3.2014.4

an invalid type cast, an array over indexing, or division by zero. These exceptions are dangerous because they can cause a sudden stop of the program, as they do not have to be handled by the programmer explicitly.

Exploration of these exceptions is done by using a technique called symbolic execution [12]. When a program is executed symbolically, it is not executed on concrete input data but input data is handled as symbolic variables. When the execution reaches a branching condition containing a symbolic variable, the execution continues on both branches. This way, all of the possible branches of the program will be executed in theory. Java PathFinder (JPF) [10] is a software model checker which is developed at NASA Ames Research Center. In fact, Java PathFinder is a Java virtual machine that executes Java bytecode in a special way. Symbolic PathFinder (SPF) [14] is an extension of JPF, which can perform symbolic execution of Java bytecodes. The presented work is based on these tools.

The paper explains how the detection of runtime exceptions of the Java programming language was implemented using Java PathFinder and symbolic execution. Concrete input parameters of the method resulting a runtime exception are also determined. It is also described how the number of execution branches, and the state space have been reduced to achieve a better performance. The implemented tool called *Jpf Checker* has been tested on real life projects, the log4j, ArgoUML, and jEdit open source systems. We found multiple errors in the *log4j* system that were also reported as real bugs in its bug tracking system. The performance of the tool is acceptable since the analysis was finished in a couple of hours even for the biggest system used for testing.

The remainder of the paper is organized as follows. We give a brief introduction to symbolic execution in Section 2. After that in Section 3 we present our approach for detecting runtime exceptions. Section 4 discusses the results of the implemented algorithm on different small examples and real life open source projects. Section 5 collects the works that related to ours. Finally, we conclude the paper and present some future work in Section 6.

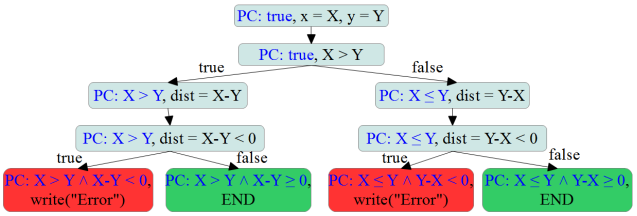
2 Symbolic Execution

During its execution, every program performs operations on the input data in a defined order. Symbolic execution [12] is based on the idea that the program is operated on symbolic variables instead of specific input data, and the output will be a function of these symbolic variables. A symbolic variable is a set of the possible values of a concrete variable in the program, thus a symbolic state is a set of concrete states. When the execution reaches a selection control structure (e.g. an if statement) where the logical expression contains a symbolic variable, it cannot be evaluated, its value might be also true and false. The execution continues on both branches accordingly. This way we can simulate all the possible execution branches of the program.

During symbolic execution we maintain a so-called *path condition (PC)*. The path condition is a quantifier-free logical formula with the initial value of true, and

```
1. int x, y, dist;
2. ...
3. if (x > y) {
4.   dist = x - y;
5. } else {
6.   dist = y - x;
7. }
8. if (dist < 0)
9.   write("Error");
```

(a)



(b)

Figure 1: (a) Sample code that determines the distance of two integers on the number line (b) Symbolic execution tree of the sample code handling variable x and y symbolically

its variables are the symbolic variables of the program. If the execution reaches a branching condition that depends on one or more symbolic variables, the condition will be appended to the current PC with the logical operator *AND* to indicate the true branch, and the negation of the condition to indicate the false branch. With such an extension of the PC, each execution branch will be linked to a unique formula over the symbolic variables. In addition to maintaining the path condition, symbolic execution engines make use of the so called *constraint solver* programs. Constraint solvers are used to solve the path condition by assigning values to the symbolic variables that satisfy the logical formula. Path condition can be solved at any point of the symbolic execution. Practically, the solutions serve as test inputs that can be used to run the program in such a way that the concrete execution follows the execution path for which the PC was solved.

All of the possible execution paths define a connected and acyclic directed graph called *symbolic execution tree*. Each point of the tree corresponds to a symbolic state of the program. An example is shown in Figure 1.

Figure 1 (a) shows a sample code that determines the distance of two integers x and y. The symbolic execution of this code is illustrated on Figure 1 (b) with the corresponding symbolic execution tree. We handle x and y symbolically, their symbols are X and Y respectively. The initial value of the path condition is true. Reaching the first if statement in line 3, there are two possibilities: the logical expression can be true or false; thus the execution branches and the logical expression and its negation is added to the PC as follows:

$$true \wedge X > Y \Rightarrow X > Y, \quad \text{and} \quad true \wedge \neg(X > Y) \Rightarrow X \leq Y$$

The value of variable *dist* will be a symbolic expression, X-Y on the true branch and Y-X on the false one. As a result of the second if statement (line 8) the execution branches, and the appropriate PCs are appended again. On the true branches we get the following PCs:

$$X > Y \wedge X - Y < 0 \Rightarrow X > Y \wedge X < Y,$$

$$X \leq Y \wedge Y - X < 0 \Rightarrow X \leq Y \wedge X > Y$$

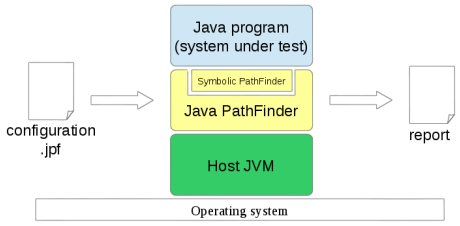


Figure 2: Java PathFinder as a virtual machine itself runs on a JVM, while performing a verification of a Java program

It is clear that these formulas are unsolvable, we cannot specify such X and Y that satisfy the conditions. This means that there are no such x and y inputs with which the program reaches the *write("Error")* statement. As long as the PC is unsatisfiable at a state, the sub-tree starting from that state can be pruned, there is no sense to continue the controversial execution.

It is impossible to explore all the symbolic states. It takes unreasonably long time to execute all the possible paths. A solution for this problem can be e.g. to limit the depth of the symbolic execution tree or the number of states which, of course, inhibit to examine all the states. The next subsection describes what are the available techniques in Symbolic PathFinder to address this problem.

2.1 Java PathFinder and Symbolic PathFinder

Java PathFinder (JPF) [10] is a highly customizable execution environment that aims at verifying Java programs. In fact, JPF is nothing more than a Java Virtual Machine which interprets the Java bytecode in a special way to be able to verify certain properties. It is difficult to determine what kind of errors can be found and which properties can be checked by JPF, it depends primarily on its configuration. The system has been designed from the beginning to be easily configurable and extendable. One of its extensions is *Symbolic PathFinder (SPF)* [14] that provides symbolic execution of Java programs by implementing a bytecode instruction set allowing to execute the Java bytecode according to the theory of symbolic execution.

JPF (and SPF) itself is implemented in Java, so it also have to run on a virtual machine, thus JPF is actually a middleware between the standard JVM and the bytecode. The architecture of the system is illustrated on Figure 2.

To start the analysis we have to make a configuration file with .jpf extension in which we specify different options as key-value pairs. The output is a report that contains e.g. the found defects. In addition to the ability of handling logical, integer and floating-point type variables as symbols, SPF can also handle complex types symbolically with the lazy initialization algorithm [11], and allows the symbolic execution of multi-threaded programs too.

SPF supports multiple constraint solvers and defines a general interface to communicate them. *Cvc3* is used to solve linear formulas, *choco* can handle non-linear

logical formulas too, while *IASolver* use interval arithmetic techniques to satisfy the path condition. Among the supported constraint solvers, *CORAL* proved to be the most effective in terms of the number of solved constraints and the performance [19].

To reduce the state space of the symbolic execution SPF offers a number of options. We can specify the maximum depth of the symbolic execution tree, and the number of elementary formulas in the path condition can also be limited. Further possibility is that with options *symbolic.minint*, *symbolic.maxint*, *symbolic.minreal*, and *symbolic.maxreal* we can restrict the value ranges of the integer and floating point types. With the proper use of these options the state space and the time required for the analysis can be reduced significantly.

3 Detection of Runtime Exceptions

We developed a tool that is able to automatically detect runtime exceptions in an arbitrary Java program. This section explains in detail how this analysis program, the JPF checker works.

To check the whole program we use symbolic execution, which is performed by Symbolic PathFinder. However, we do not execute the whole program symbolically to discover all of the possible paths, instead we symbolically execute the methods of the program one by one. Starting the analysis from the *main* method has several drawbacks. For example, the state space would be too large and we would need to cut it when the execution reaches the defined maximal depth in the symbolic execution tree. Our approach results in a significant reduction in the state space of the symbolic execution.

An important question is which variables to be handled symbolically. In general, execution of a method mainly depends on the actual values of its parameters and the referred external variables. Thus, these are the inputs of a method that should be handled symbolically to generally analyze it. Currently, we handle the parameters and data members of the class of the analyzed method symbolically.

Our goal is not only to indicate the runtime exceptions a method can throw (its type and the line causing the exception), but also to determine a parameterization that leads to throwing those exceptions. In addition, we determine this parameterization not only for the analyzed method which is at the bottom of the call stack, but for all the other elements in the call stack (i.e. recursively for all the called methods).

Our work can be divided into two steps:

1. It is necessary to create a runtime environment which is able to iterate through all the methods of a Java program, and start their symbolic execution using Symbolic PathFinder.

2. We need a JPF extension which is built on its listener mechanism, and which is able to indicate potential runtime exceptions and related parameterization while monitoring the execution.

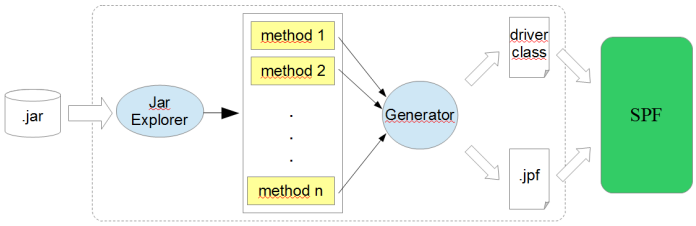


Figure 3: Architecture of the runtime environment

3.1 The Runtime Environment

The concept of the developers of Symbolic PathFinder was to start running the program in normal mode like in a real life environment, than at given points, e.g. at more complex or problematic parts in the program switch to symbolic execution mode [15]. The advantage of this approach is that, since the context is real, it is more likely to find real errors. E.g. the values of the global variables are all set, but if these variables are handled symbolically we can examine cases that never occur during a real run. A disadvantage is that it is hard to explore the problematic points of a program, it requires prior knowledge or preliminary work. Another disadvantage is that you have to run the program manually namely, that the control reach those methods which will be handled symbolic by the SPF.

In contrast, the tool we have developed is able to execute an arbitrary method or all methods of a program symbolically. The advantage of this approach is that the user does not have to perform any manual runs, the entire process can be automated. Additionally, the symbolic state space also remains limited since we do not execute the whole program symbolically, but their parts separately. The approach also makes it possible to analyze libraries that do not have a *main* method such as log4j. One of the major disadvantages is the that we back away from the real execution environment, which may lead to false positive error reports.

For implementing such an execution environment we have to achieve somehow that the control flow reaches the method we want to analyze. However, due to the nature of the virtual machine, JPF requires the entry point of the program, which is the class containing the main method. Therefore, we generate a driver class for each method containing a main method that only passes the control to the method we want to execute symbolically and carries out all the related tasks. Invoking the method is done using the Java Reflection API. We also have to generate a JPF configuration file that specifies, among others, the artificially created entry point and the method we want to handle symbolically. After creating the necessary files, we have to compile the generated Java class and finally, to launch Symbolic PathFinder.

The architecture of the system is illustrated in Figure 3. The input *jar* file is processed by the *JarExplorer*, which reads all the methods of the classes from the jar file and creates a list from them. The elements of the list is taken by the *Generator* one by one. It generates a driver class and a JPF configuration file for

```
1. exceptionThrown() {
2.   exception = getPendingException();
3.   if (isInstanceOfRuntimeException(exception)) {
4.     pc = getCurrentPc();
5.     solve(pc);
6.     summary = new FoundExceptionSummary();
7.     summary.setExceptionType(exception);
8.     summary.setThrownFrom(exception);
9.     summary.setParameterization(parsePc(pc, analyzedMethod));
10.    invocationChain = buildInvocationChain();
11.    foreach(Method m : invocationChain) {
12.      summary.addStackTraceElement(m, parsePc(pc, m));
13.    }
14.    foundExceptions.add(summary);
15.  }
16.}
```

Figure 4: Pseudo code of the exceptionThrown event

each method. After the generation is complete, we start the symbolic execution.

3.2 Implementing a Listener Class

During functioning, JPF sends notifications about certain events. This is realized with so-called listeners, which are based on the observer design pattern. The registered listener objects are notified about and can react to these events. JPF can send notifications of almost every detail of the program execution. There are low-level events such as execution of a bytecode instruction, as well as high-level events such as starting or finishing the search in the state space. In JPF, basically two listener interfaces exist: the *SearchListener* and *VMListener* interface. While the former includes the events related to the state space search, the latter reports the events of the virtual machine. Because these interfaces are quite large and the specific listener classes often implement both of them, adapter classes are introduced that implement these interfaces with empty method bodies. Therefore, to create our custom listener we derived a class from this adapter and implemented the necessary methods only.

Our algorithm for detecting runtime exceptions is briefly summarized below. By performing symbolic execution of a method all of its paths are executed, including those that throw exceptions. When an exception occurs, namely when the virtual machine executes an ATHROW bytecode instruction, JPF triggers and *exceptionThrown* event. Thus, we implemented the exceptionThrown method in our listener class. Its pseudo code is shown in Figure 4.

First, we acquire the thrown Exception object (line 2), then we decide whether it is a runtime exception (i.e. whether it is an instance of the class RuntimeException) (line 3). If it is, we request the path condition related to the actual path and

use the constraint solver to find a satisfactory solution (lines 4-5). Lines 6-9 set up a summary report that contains the type of the thrown exception, the line that throws it and a parameterization which causes this exception to be thrown. The parameterization is constructed by the *parsePC()* method, which assigns the satisfactory solutions of the path condition to the method parameters. Lines 10-13 take care of collecting and determining parameterization for the methods in the call stack. If the source code does not specify any constraint for a parameter on the path throwing an exception (i.e. the path condition does not contain the variable), then there is no related solution. This means that it does not matter what the actual value of that parameter is, as it does not affect the execution path, and the method is going to throw an exception due to the values of other parameters. In such cases parsePc() method assigns the value "any" to these parameters.

It is also possible that a parameter has a concrete value. Figure 5 illustrates such an example. When we start the symbolic execution of method *x()*, its parameter *a* is handled symbolically. As *x()* calls *y()* its parameter *a* is still a symbol, but *b* is a concrete value (42). In a case like this, parsePc() have to get the concrete value from the stack of the actual method.

```
1. void x(int a) {
2.   short b = 42;
3.   y(a, b);
4. }
```

```
5. void y(int a, short b) {
6.   ...
7.   throw new NullPointerException();
8.   ...
9. }
```

Figure 5: An example call with both symbolic and concrete parameters

We note that the presented algorithm reports any runtime exceptions regardless of the fact whether it is caught by the program or not. The reason of this is that we think that relying on runtime exceptions is a bad coding practice and a runtime exception can be dangerous even if it is handled by the program. Nonetheless, it would be easy to modify our algorithm to detect uncaught exceptions only.

4 Results

The developed tool was tested in a variety of ways. The section describes the results of these test runs. We analyzed manually prepared example codes containing instructions that cause runtime exceptions on purpose; then we performed analysis on different open-source software to show that our tool is able to detect runtime exceptions in real programs, not just in artificially made small examples. The subject systems are the log4j (`http://logging.apache.org/log4j/`) logging library, the ArgoUML modeling tool (`http://argouml.tigris.org/`), and the jEdit text editor program (`http://www.jedit.org/`). We prove the validity of the detected exceptions by the bug reports, found in the bug tracking systems of these projects, that describe program faults caused by those runtime exceptions that are also found by the developed tool.

```
    public class Example5 {
    ...
8.  void callRun(int x, int y) {
9.    Integer i = null;
10.   if (x > 6) {
11.     int b = 9;
12.     run(b, y);
13.     i = Integer.valueOf(b);
14.     System.out.println(i);
15.   } else {
16.     i = Integer.valueOf(3);
17.     System.out.println(i);
18.   }
19. }
```

```
20. public void run(int x, int y) {
21.   if (y > 10) {
22.     int[] arr = new int[5];
23.     for (int i = 0; i < x; i++) {
24.       arr[i] = i;
25.     }
26.   } else {
27.     Integer i = null;
28.     if (y < 5) {
29.       i = Integer.valueOf(4);
30.       i.floatValue();
31.     } else {
32.       System.out.println(
33.                   i.floatValue());
34.     }
35.   }
36. }}
```

Figure 6: Manually prepared example code with the analysis of method callRun()

4.1 Manually Prepared Examples

A small manually prepared example code is shown on Figure 6. The method under test is *callRun()* which calls method *run()* in line 12. Running our algorithm on this code gives two hits: the first is an ArrayIndexOutOfBoundsException, the second is a NullPointerException. The first exception is thrown by method run() at line 24. A parametrization leading to this exception is callRun(7, 11). Method run() will be called only if $x > 6$ (line 10) that is satisfied by 7 and it is called with the concrete value 9 and symbol y. At this point there is no condition for y. Method run() can reach line 24 only if $y > 10$, the indicated value 11 is obtained by satisfying this constraint. Throwing of the ArrayIndexOutOfBoundsException is due to the fact that in line 22 we declare a 5-element array but the following for loop runs from 0 to x. The value of x at this point is 9 which leads to an exception.

The train of thought is similar in the case of the second exception. The problem is that variable i created in line 27 initialized only in line 29 to a value different form *null*, but not in the else block, therefore line 33 throws a NullPointerException. This requires that the value of y not to be greater than 10 and not to be less than 5. These restrictions are satisfied by e.g. 5, and value 7 for x is necessary to invoke run(). So the parametrizations are callRun(7, 5) and run(9, 5). The analysis is finished in less than a second.

A second example code is presented in Figure 7. The resulting report refers to an ArithmeticException, which is thrown at line 39 and the stack trace highlights that the problematic method is *expand()* which is invoked at line 30 by *run()*. The control flow reaches line 30 only if variable b is false. For example, if n is -999, and *check* has the value true, as the parameter list in the error report included, b will be false and the *expand()* method on the else branch will be executed. At

```
   ...
3. public class Example3 {
     ...
8.   public void run(int n,
          boolean check, A a) {
9.     boolean b = check && n >= 0;
10.     int max = Integer.MIN_VALUE;
11.     if (b) {
12.       if (a != null) {
13.         int l = n;
14.         int r = 2*n + 1;
15.         if (a.getMember() > 120) {
16.           if (l <= a.getMember()) {
17.             max = a.getMember();
18.           } else {
19.             max = l;
20.           }
21.           if (r > max) {
22.             max = r;
23.           }
24.           while (max < n) {
25.             max = expand(n, 0);
26.           }
27.         }
28.       }
29.     } else {
30.       max = expand(n, 0);
31.     }
32.     System.out.println("Maximum"
33.                   + value: " + max);
34.   }
```

```
35. private int expand(int n, int m) {
36.   double res = count(m);
37.   if (res > n) {
38.     do {
39.       res = n / res;
40.       res -= 2;
41.     } while (res >= 0);
42.     return n + m;
43.   } else {
44.     return (int)res;
45.   }
46. }
47.
48. private int count(int l) {
49.     int count = l;
50.     for (int i=100; i>0; i--) {
51.       if (i % 3 == 0) {
52.         count++;
53.       }
54.     }
55.     return count;
56.   }
57.
58. }
```

```
1. public class A extends Letter {
2.   ...
3.   public int member;
4.
5.   public int getMember() {
6.     return member;
7.   }
8.   ...
9. }
```

Figure 7: Manually prepared example code with the analysis of method run()

line 36, variable *res* has a concrete value because method *count()* will be executed. It can be seen that *res* is definitely a non-negative integer, thus the condition at line 37 is true if *n=-999*. Then the loop begins to execute, and variable *res* will be reduced to 0 after a number of iterations, leading to a division by 0 fault. In the report, the third parameter of the examined *run()* method is "any". That is because this parameter does not play a role in whether or not the program runs onto the discussed ArithmeticException.

Line 25 in method *run()* also calls *expand()*, but there is no corresponding error report. In fact, due to the instructions at lines 13-23, the condition at line 24 is always false, thus this *expand()* call will never be executed. Actually, line 25 is unreachable code.

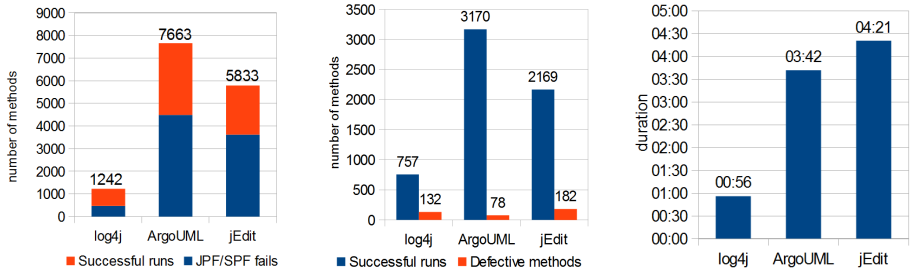


Figure 8: (a)Number of methods examined in the programs and the number of JPF or SPF faults (b) Number of successfully analyzed methods and the number of defective methods (c) Analysis time

4.2 Analysis of Open-source Systems

Analysis of log4j 1.2.15, ArgoUML 0.28 and jEdit 4.4.2 were carried out on a desktop computer with an Intel Core i5-540M 2.53 GHz processor and 8 GB of memory. In all three cases the analysis was done by executing all the methods of the release jar files of the projects symbolically.

Figure 8 (a) displays the number of methods we analyzed in the different programs. We started analyzing 1242 methods in log4j of which only 757 were successful, in 474 cases the analysis stopped due to the failure of the Java PathFinder (or Symbolic PathFinder). There are a lot of methods in ArgoUML which also could not be analyzed, more than half of the checks ended with failure. In case of jEdit the ratio is very similar. Unfortunately, in general JPF stopped with a variety of error messages.

Despite the frequent failures of JPF, our tool indicated a fairly large number of runtime exceptions in all three programs. Figure 8 (b) shows the number of successfully analyzed methods and the methods with one or more runtime exceptions. The hit rate is the highest for log4j and despite its high number of methods, relatively few exceptions were found in ArgoUML.

The analysis times are shown in Figure 8 (c). Analysis of log4j completed within an hour, while analysis of ArgoUML, that contains more than 7500 methods, took 3 hours and 42 minutes. Although jEdit contains fewer methods than ArgoUML, its full analysis were more time-consuming. The performance of our algorithm is acceptable, especially considering that the analysis was performed on an ordinary desktop PC not on a high-performance server. However, it can be assumed that the analysis time would grow with less failed method analysis.

It is important to note, that not all indicated exceptions are real errors. This is because the analysis were performed in an artificial execution environment which might have introduced false positive hits. When we start the symbolic execution of a method we have no information about the circumstances of the real invocation. All parameters and data members are handled symbolically, that is, it is considered

```
    public class SimpleLayout extends Layout {
      ...
58.   public String format(LoggingEvent event) {
59.
60.     sbuf.setLength(0);
61.     sbuf.append(event.getLevel().toString());
62.     sbuf.append(" - ");
63.     sbuf.append(event.getRenderedMessage());
64.     sbuf.append(LINE_SEP);
65.     return sbuf.toString();
66.   }
      ...
    }
    public class LoggingEvent implements java.io.Serializable {
      ...
      transient public Priority level;
       ...
255.   public Level getLevel() {
256.     return (Level) level;
257.   }
    }
    public class Level extends Priority implements Serializable{
      ...
    }
```

Figure 9: Method org.apache.log4j.SimpleLayout.format() and its environment.

that their value can be anything although it is possible that a particular value of a variable never occurs.

Despite the fact that not all the reported exceptions are real program errors they are definitely representing real risks. During the modification of the source code there are inevitably changes that introduce new errors. These errors often appear in form of runtime exceptions (i.e. in places where our algorithm found possible failures). So the majority of the reported exceptions do not report real errors, but potential sources of danger that should be paid special attention.

In the following, we are going to show some interesting faults found by our tool in the above systems.

The first example method is *org.apache.log4j.SimpleLayout.format()* of log4j, which is shwon in Figure 9. In this method three possible runtime exceptions are found by the tool. The first two are NullPointerExceptions, both thrown at line 61. The produced report says that the first NPE will be thrown if the parameter is *null*, and the second when this parameter differs from *null*. In the first case, when the parameter is null, expression *event.getLevel()* causes the exception, since a method of a null reference cannot be called. When parameter *event* is not null, the code gets the *level* data member and calls its *toString()* method. The second NullPointerException is caused by the fact that the requested *level* data member

```
     public class FindDialog extends ArgoDialog ... { ... }
     class PredicateMType extends PredicateType {
       ...
727.   public static PredicateType create(Object c0, Object c1, Object c2) {
728.     Class[] classes = new Class[3];
729.     classes[0] = (Class) c0;
730.     classes[1] = (Class) c1;
731.     classes[2] = (Class) c2;
732.     return new PredicateMType(classes);
733.   }
       ...
     }
```

Figure 10: Method org.argouml.ui.PredicateMType.create()

can also be null, thus using operator '.' may raise the exception.

The third exception is a ClassCastException. As shown, at line 256 in class *LoggingEvent* there is a type cast which tries to convert the *level* member which has a type *Priority* to a *Level* object. According to the listing in the bottom of Figure 9, class *Level* is a descendant of class *Priority*, thus the cast at line 256 is a downcast, which is incorrect in case the dynamic type of the member is not *Level*.

Three possible ClassCastExceptions are revealed in method *PredicateMType.create()* that is depicted in Figure 10. Lines 729, 730 and 731 cast down the three parameters from *Object* to *Class* without performing any type check. The first entry in the report says that *create(null, null, !null)* parametrization can lead to an exception thrown at line 731. If *c0* and *c1* parameters are null, lines 729 and 730 are executed without any problem, because casting a null reference to any class is permitted in Java. It is important that this does not mean that *c0* and *c1* have to be necessarily null, the report just gives a sample parametrization which leads the execution to the exception. As long as the third parameter is not null a ClassCastException can be raised. Of course, to achieve this it is necessary that the parameter type is different from *Class*. Parametrization *create(null, !null, "any")* leads to potential fault at line 730. The reasoning is similar to the previous one: if *c0* is null and *c1* is non-null (and of course it is not a *Class*) ClassCastException will be thrown. The third parameter is completely irrelevant. In case of the third ClassCastException, occurring at line 729, the values of *c1* and *c2* do not matter.

The last example is a tiny method, *MRUFileManager.getFile()* shown in Figure 11. At line 98, *getFile()* checks whether the *index* parameter is less then the size of the *_mruFileList* LinkedList. If so, the return value is the corresponding element of the LinkedList, otherwise *null*. Our report shows that the *index* can be a negative number, too. This case is not handled, and *LinkedList.get()* will throw an IndexOutOfBoundsException if method *getField()* is called for example with -999. Calling *getField()* with a negative number seems unreasonable and of course it is, but possible.

```
    public class MRUFileManager {
      ...
      private LinkedList _mruFileList;
      ...
      public int size() {
        return _mruFileList.size();
      }
97.   public Object getFile(int index) {
98.     if (index < size()) {
99.       return _mruFileList.get(index);
100.    }
101.
102.    return null;
103.  }
      ...
    }
```

Figure 11: Method org.apache.log4j.lf5.viewer.configure.MRUFileManager.getFile()

4.3 Real Errors

In this subsection a few defects are presented which are reported in bug tracking systems, and caused by runtime exceptions found also by our tool. The first affected bug[1] reports the termination of an application using log4j version 1.2.14 caused by a NullPointerException. The reporter got the Exception from line 59 of *ThrowableInformation.java* thrown by method *org.apache.log4j.spi.ThrowableInformation.getThrowableStrRep()* as shown in the given stack trace. The code of the method and the problematic line detected by our analysis is shown in Figure 12.

The problem here is that the initialization of the *throwable* data member of class *ThrowableInformation* is omitted, its value is null causing a NullPointerException at line 59. This causes that the *log()* method of log4j can also throw an exception which should never happen. Our tool found other errors as well which demonstrate its strength of being capable of detecting real bugs.

The next exception is also a NullPointerException, which occurred in log4j 1.2.15. The bug report[2] explains that the runtime exception causing the halt comes form method *org.apache.log4j.NDC.remove()*, at line 377. Figure 13 shows the corresponding piece of code. The fault here is that the *ht* static data member is null. Although the data member is initialized as Figure 13 shows, it is possible that during the execution its value is set to null. The report in the log4j bug tracking system sheds light to this. The reporter also mentions that according to his observations, the other methods of class *NDC*, which use the *ht* member, first check whether it is null or not, but in method *remove()* there is no such investigation.

[1] https://issues.apache.org/bugzilla/show_bug.cgi?id=44038
[2] https://issues.apache.org/bugzilla/show_bug.cgi?id=45335

```
    public class ThrowableInformation implements java.io.Serializable {
      private transient Throwable throwable;
      ...
54.   public String[] getThrowableStrRep() {
55.     if(rep != null) {
56.       return (String[]) rep.clone();
57.     } else {
58.       VectorWriter vw = new VectorWriter();
59.       throwable.printStackTrace(vw);
60.       rep = vw.toStringArray();
61.       return rep;
62.     }
63.   }
    ...
    }
```

Figure 12: Method org.apache.log4j.spi.ThrowableInformation.getThrowableStrRep()

We describe one more error that was also found in log4j version 1.2.15[3]. The error is at line 312 of the class *org.apache.log4j.net.SyslogAppender*. The line is inside the method *append()* in which there is a NullPointerException again. The code snippet is shown in Figure 14.

The reason of this runtime error is that the *layout* data member, which is inherited from class *AppenderSkeleton*, stays uninitialized. Our report also includes a ClassCastException thrown by method *getLevel()* at line 294. This fault is the same that we already described explaining Figure 9 in the previous subsection.

5 Related Work

In this section we present works that are related to our research. First, we introduce some well-known symbolic execution engines, then we show the possible applications of the symbolic execution. We also summarize the problems that have been solved successfully by Symbolic PathFinder that we used for implementing our approach. Finally, we present the existing approaches and techniques for runtime exception detection.

The idea of symbolic execution is not new, the first publications and execution engines appeared in the 1970's. One of the earliest work is by King that lays down the fundamentals of symbolic execution [12] and presents the EFFIGY system that is able to execute PL/I programs symbolically. Even though EFFIGY handles only integers symbolically, it is an interactive system with which the user is able to examine the process of symbolic execution by placing breakpoints and saving and restoring states. Another work from the 1970's by Boyer et al. presents a similar system called SELECT [1] that can be used for executing LISP programs

[3] https://issues.apache.org/bugzilla/show_bug.cgi?id=46271

```
     public class NDC {
       ...
       static Hashtable ht = new Hashtable();
       ...
374.   static
375.   public
376.   void remove() {
377.     ht.remove(Thread.currentThread());
378.
379.     // Lazily remove dead-thread references in ht.
380.     lazyRemove();
381.   }
       ...
     }
```

Figure 13: Source code of method org.apache.log4j.NDC.remove()

symbolically. The users are allowed to define conditions for variables and return values and get back whether these conditions are satisfied or not as an output. The system can be applied for test input generation; in addition, for every path it gives back the path condition over the symbolic variables.

Starting from the last decade the interest about the technique is constantly growing, numerous programs have been developed that aim at dynamic test input generation using symbolic execution. The EXE (EXecution generated Executions) [3] presented by Cadar et al. at the Stanford University is an error checking tool made for generating input data on which the program terminates with failure. The input generation is done by the STP built-in constraint solver that solves the path condition of the path causing the failure. EXE achieved promising results on real life systems. It found errors in the package filter implementations of *BSD* and *Linux*, in the *udhcpd* DHCP server and in different Linux file systems. The runtime detection algorithm presented in this work solves the path condition to generate test input data similarly to EXE. The basic difference is that for running EXE one needs to declare the variables to be handled symbolically while for Jpf Checker there is no need for editing the source code before detection.

The DART [7] (Directed Automata Random Testing) by Godefroid et al. tries to eliminate the shortcomings of the symbolic execution e.g. when it is unable to handle a condition due to its unlinear nature. DART executes the program with random or predefined input data and records the constraints defined by the conditions on the input variables when it reaches a conditional statement. In the next iteration taking into account the recorded constraints it runs the program with input data that causes a different execution branch of the program. The goal is to execute all the reachable branches of the program by generating appropriate input data. The CUTE and jCUTE systems [16] by Sen and Agha extend DART with multithreading and dynamic data structures. The advantage of these tools is that they are capable of handling complex mathematical conditions due to concrete

```
    public abstract class AppenderSkeleton {
      protected Layout layout;
      ...
    }
    public class SyslogAppender extends AppenderSkeleton {
      SyslogQuietWriter sqw;
      private boolean layoutHeaderChecked = false;
      ...
291.  public
292.  void append(LoggingEvent event) {
293.
294.    if(!isAsSevereAsThreshold(event.getLevel()))
295.      return;
296.
297.    // We must not attempt to append if sqw is null.
298.    if(sqw == null) {
299.      errorHandler.error("No syslog host is set for SyslogAppedender"
300.                                    + named " + this.name + ".");
301.      return;
302.    }
303.
304.    if (!layoutHeaderChecked) {
305.        if (layout != null && layout.getHeader() != null) {
306.            sendLayoutMessage(layout.getHeader());
307.        }
308.        layoutHeaderChecked = true;
309.    }
310.
311.
312.    String packet = layout.format(event);
313.    String hdr = getPacketHeader(event.timeStamp);
314.
315.    if(facilityPrinting || hdr.length() > 0) {
316.        StringBuffer buf = new StringBuffer(hdr);
317.        if(facilityPrinting) {
318.            buf.append(facilityStr);
319.        }
320.        buf.append(packet);
321.        packet = buf.toString();
322.    }
        ...
      }}
```

Figure 14: Source code of method org.apache.log4j.net.SyslogAppender.append()

executions. This can be also achieved in Jpf Checker by using the concolic execution of SPF; however, symbolic execution allows a more thorough examination of the source code. Further description and comparison of the above mentioned tools can be found e.g. in the work of Coward [4].

There are also approaches and tools for generating test suites for .NET programs using symbolic execution. Pex [21] is a tool that automatically produces a small test suite with high code coverage for .NET programs using dynamic symbolic execution, similar to path-bounded model-checking. Jamrozik et al. introduce an extension of the previous approach called augmented dynamic symbolic execution [9], which aims to produce representative test sets with DSE by augmenting path conditions with additional conditions that enforce target criteria such as boundary or mutation adequacy, or logical coverage criteria. Experiments with the Apex prototype demonstrate that the resulting test cases can detect up to 30% more seeded defects than those produced with Pex.

Song et al. applied the symbolic execution to the verification of networking protocol implementations [18]. The SymNV tool creates network packages with which a high coverage can be achieved in the source code of the daemon, therefore potential rule violations can be revealed according to the protocol specifications.

The SAFELI tool [6] by Fu and Qian is a SQL injection detection program for analyzing Java web applications. It first instruments the Java bytecode then executes the instrumented code symbolically. When the execution reaches a SQL query the tool prepares a string equation based on the initial content of the web input components and the built-in SQL injection attack patterns. If the equation can be solved the calculated values are used as inputs which the tool verifies by sending a HTML form to the server. According to the response of the server it can decide whether the found input can be a real attack or not.

The main application of the Java PathFinder and its symbolic execution extension is the verification of the internal projects in NASA. Bushnell et al. describes the application of Symbolic PathFinder in TSAFE (Tactical Separation Assisted Flight Environment) [2] that verifies the software components of an air control and collision detection system. The primary target is to generate useful test cases for TSAFE that simulates different wind conditions, radar images, flight schedules, etc.

The detection of design patterns can be performed using dynamic approaches as well as with static program analysis. With the help of a monitoring software the program can be analyzed during manual execution and conclusions about the existence of different patterns can be made based on the execution branches. In his work, von Detten [22] applied symbolic execution with Symbolic PathFinder supplementing manual execution. This way, more execution branches can be examined and the instances found by traditional approaches can be refined.

Ihantola [8] describes an interesting application of JPF in education. He generates test inputs for checking the programs of his students. His approach is that functional test cases based on the specification of the program and their outcome (successful or not) is not enough for educational purposes. He generates test cases for the programs using symbolic execution. This way the students can get feedbacks like “the program works incorrectly if variable a is larger than variable b plus 10”.

Sinha et al. deal with localizing Java runtime errors [17]. The introduced approach aims at helping to fix existing errors. They extract the statement that threw the exception from its stack trace and perform a backward dataflow analysis starting from there to localize those statements that might be the root causes of the exception.

The work of Weimer and Necula [23] focuses on proving safe exception handling in safety critical systems. They generate test cases that lead to an exception by violating one of the rules of the language. Unlike Jpf Checker they do not generate test inputs based on symbolic execution but solving a global optimization problem on the control flow graph (CFG) of the program.

The JCrasher tool [5] by Csallner and Smaragdakis takes a set of Java classes as input. After checking the class types it creates a Java program which instantiates the given classes and calls each of their public methods with random parameters. This algorithm might detect failures that cause the termination of the system such as runtime exceptions. The tool is capable of generating JUnit test cases and can be integrated to the Eclipse IDE. Similarly to Jpf Checker JCrasher also creates a driver environment but it can analyze public methods only and instead of symbolic execution it generates random data which is obviously not feasible for examining all possible execution branches.

6 Conclusions and Future Work

The introduced approach for detecting runtime exceptions works well not just on small, manually prepared examples but it is able to find runtime exceptions which are the causes of some documented runtime failures (i.e. there exists an issue for them in the bug tracking system) in real world systems also. However, not all the detected possible runtime exceptions will actually cause a system failure. There might be a large number of exceptions that will never occur running the system in real environment. Nonetheless, the importance of these warnings should not be underrated since they draw attention to those code parts that might turn to real problems after changing the system. Considering these possible problems could help system maintenance and contributes to achieving a better quality software. As we presented in Section 4 the analysis time of real world systems are also acceptable, therefore our approach and tool can be applied in practice.

Unfortunately the Java PathFinder and its Symbolic PathFinder extension – which we used for implementing our approach – contain a lot of bugs. It made the development very troublesome, but the authors at the NASA were really helpful. We contacted them several times and got responses very quickly; they fixed some blocker issues particularly for our request. Although JPF and SPF have several bugs, it is under constant development and becoming more and more stable.

The achieved results are very promising and we continue the development of our tool. Our future plan is to eliminate the false positive and those hits that are irrelevant. We would also like to provide more details about the environment of the method in which the runtime exception is detected. The implemented tool gives

only the basic information about the reference type parameters whether they are *null* or not, and we cannot tell anything about the values of the member variables of the class playing a role in a runtime exception. These improvements of the algorithm are also in our future plans.

The presented approach is not limited to runtime exception detection. We plan to utilize the potentials of the symbolic execution by implementing other types of error and rule violation checkers. E.g. we can detect some special types of infinite loops, dead or unused code parts, or even SQL injection vulnerabilities.

References

[1] Boyer, Robert S., Elspas, Bernard, and Levitt, Karl N. SELECT – a Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, New York, NY, USA, 1975. ACM.

[2] Bushnell, D., Giannakopoulou, D., Mehlitz, P., Paielli, R., and Păsăreanu, Corina S. Verification and Validation of Air Traffic Systems: Tactical Separation Assurance. In *Aerospace Conference, 2009 IEEE*, pages 1–10, 2009.

[3] Cadar, Cristian, Ganesh, Vijay, Pawlowski, Peter M., Dill, David L., and Engler, Dawson R. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[4] Coward, P. David. Symbolic Execution Systems – a Review. *Software Engineering Journal*, 3(6):229–239, November 1988.

[5] Csallner, Christoph and Smaragdakis, Yannis. JCrasher: an Automatic Robustness Tester for Java. *Software Practice and Experience*, 34(11):1025–1050, September 2004.

[6] Fu, Xiang and Qian, Kai. SAFELI: SQL Injection Scanner Using Symbolic Execution. In *Proceedings of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications*, TAV-WEB '08, pages 34–39, New York, 2008. ACM.

[7] Godefroid, Patrice, Klarlund, Nils, and Sen, Koushik. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[8] Ihantola, Petri. Test Data Generation for Programming Exercises with Symbolic Execution in Java PathFinder. In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research*, Baltic Sea '06, pages 87–94, New York, 2006. ACM.

[9] Jamrozik, Konrad, Fraser, Gordon, Tillman, Nikolai, and Halleux, Jonathan. Generating Test Suites with Augmented Dynamic Symbolic Execution. In *Tests and Proofs*, volume 7942 of *Lecture Notes in Computer Science*, pages 152–167. Springer Berlin Heidelberg, 2013.

[10] Java PathFinder Tool-set. `http://babelfish.arc.nasa.gov/trac/jpf`.

[11] Khurshid, Sarfraz, Păsăreanu, Corina S., and Visser, Willem. Generalized Symbolic Execution for Model Checking and Testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'03, pages 553–568, Berlin, Heidelberg, 2003. Springer-Verlag.

[12] King, James C. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[13] Pressman, Roger S. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Science/Engineering/Math, November 2001.

[14] Păsăreanu, Corina S. and Rungta, Neha. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 179–180, New York, NY, USA, 2010. ACM.

[15] Păsăreanu, Corina S., Mehlitz, Peter C., Bushnell, David H., Gundy-Burlet, Karen, Lowry, Michael, Person, Suzette, and Pape, Mark. Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 15–26, New York, NY, USA, 2008. ACM.

[16] Sen, Koushik and Agha, Gul. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-checking Tools. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 419–423, Berlin, 2006. Springer-Verlag.

[17] Sinha, Saurabh, Shah, Hina, Görg, Carsten, Jiang, Shujuan, Kim, Mijung, and Harrold, Mary Jean. Fault Localization and Repair for Java Runtime Exceptions. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, ISSTA '09, pages 153–164, New York, NY, USA, 2009. ACM.

[18] Song, JaeSeung, Ma, Tiejun, Cadar, Cristian, and Pietzuch, Peter. Rule-Based Verification of Network Protocol Implementations Using Symbolic Execution. In *Proceedings of the 20th IEEE International Conference on Computer Communications and Networks (ICCCN'11)*, pages 1–8, 2011.

[19] Souza, Matheus, Borges, Mateus, d'Amorim, Marcelo, and Păsăreanu, Corina S. CORAL: Solving Complex Constraints for Symbolic Pathfinder. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 359–374, Berlin, Heidelberg, 2011. Springer-Verlag.

[20] Tassey, G. The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical report, National Institute of Standards and Technology, 2002.

[21] Tillmann, Nikolai and De Halleux, Jonathan. Pex: White Box Test Generation for .NET. In *Proceedings of the 2nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[22] von Detten, Markus. Towards Systematic, Comprehensive Trace Generation for Behavioral Pattern Detection Through Symbolic Execution. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11, pages 17–20, New York, NY, USA, 2011. ACM.

[23] Weimer, Westley and Necula, George C. Finding and Preventing Run-time Error Handling Mistakes. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 419–431, New York, NY, USA, 2004. ACM.