Analyzing Exceptions in the Context of Test Data Generation Based on Symbolic Execution

Marcelo Medeiros Eler
University of Sao Paulo
Sao Paulo - SP - Brazil
E-mail: marceloeler@usp.br

Vinicius H. S. Durelli
University of Groningen
The Netherlands
E-mail: v.h.serapilha.durelli@rug.nl

Andre Takeshi Endo
Federal Technological University of Parana
Cornelio Procopio - PR - Brazil
E-mail: andreendo@utfpr.edu.br

***Abstract*—Testing exception scenarios is a challenging task in the context of test data generation based on symbolic execution. In such a context, test data is generated based on constraints explicitly declared in the code. However, constraints required to activate specific exceptions may not be directly declared in the code. In such a case, implicit constraints have to be inferred from exception handling mechanisms. Given that exceptions can be raised in several situations, finding constraints to generate test data to exercise all possible faulty scenarios can significantly increase the number of paths and constraints, which can cause or aggravate path explosion issues. This paper reports on an investigation that we carried out to gauge the cost (i.e., number of path constraints) of four data generation approaches aimed at covering exception dependent paths.**

Keywords – Exception handling; symbolic execution; test data generation; software analysis

I. Introduction

Automatic test data generation is a notorious complex problem. Symbolic execution and constraint solving have been used as an approach to generate test data that achieve high control-flow coverage [5, 9]. During symbolic execution, program elements are represented as functions of symbolic input values [5, 9]. Each path is represented by a path constraint, which is a sequence of constraints that should be satisfied so that the underlying path can be traversed. Constraint solvers are thus used to generate concrete input values (i.e., test data) that satisfy each set of constraints.

Often, the constraints required to traverse a path are explicitly declared in the code by means of control-flow statements, such as `if` and `while`. However, some constraints are not explicitly declared in the code because they do not stem from conventional control-flow statements. Some of these constraints implicitly derive from exception handling mechanisms such as Java's `try-catch-finally` blocks. For instance, a block that declares a `NullPointerException` will be executed only when such an exception is thrown. However, in general, there are no constraint indicating in which condition such an exception will be thrown.

We classify paths that depend on an exception being thrown as *exception-dependent* paths (EDPs) [7], as oppose to *exception-free* paths (EFPs) [14]. According to an analysis performed over a sample of 100 open source projects called SF100 [8][1], we discovered that almost one third of the methods have at least one EDP [7]. Although the number of EDPs of a program is high, and exception handling is an important topic in software development [4], the influence of exception mechanisms to unit test data generation using symbolic execution has not been widely explored [1, 3, 5].

Taking into account the implicit constraints that stem from exception handling mechanisms can significantly increase the number of path constraints. This has the potential to exacerbate a well-known issue faced by symbolic execution approaches: path explosion [1, 5, 6], which is usually caused by complex loop structures.

The contribution of this paper is twofold. First, we discuss the characteristics and possible approaches to identify constraints that exercise EDPs and faulty scenarios. Second, we investigate how different approaches to generate test data that cover EDPs may impact the number of path constraints.

This paper is organized as follows. Section II provides background on symbolic execution and EDPs. Section III discusses possible ways to identify constraints that lead to exceptions being raised and, in turn, the execution of EDPs. Section IV describes the investigation we conducted to measure the path constraint overhead brought by approaches that generate test data tailored to execute EDPs. Section V shows related work. Finally, Section VI presents concluding remarks.

II. Symbolic Execution and Exception Dependent Paths (EDPs)

Symbolic execution is a program-analysis technique that represents the elements of a program as symbolic input values [9]. Each path is represented by a path constraint, i.e., a set of constraints in a logical expression that must be satisfied in order to execute the underlying path. To generate unit test data, symbolic execution approaches resort to constraint solvers to yield solutions to the path constraints. These solutions provided by the constraint solvers are then used as test data to achieve high coverage on control-flow criteria.

Using symbolic execution and constraint solving to generate test data is appealing and straightforward. However, even though this research area has come a long way over the past

[1]Details of the SF100 corpus of classes are available at http://www.st.cs.uni-saarland.de/evosuite/SF100/

DOI reference number: 10.18293/SEKE2015-170

decades, several challenges still remain [1, 5–7, 10, 16], such as path explosion and EDPs.

Path explosion is the problem of having to cope with too many paths, which can overwhelm the constraint solver and reduce the performance of the overall process [5]. The path constraints assembled during symbolic execution are usually based on individual constraints explicitly declared in the source code by means of control-flow statements. EDPs, on the other hand, are paths that can only be traversed if a specific exception is thrown. The constraints required to raise the underlying exception of an EDP, however, are not explicit in the code. Therefore, the constraints that lead to the execution of EDPs have to be abstracted from exception handling mechanisms, viz., `try-catch-finally` blocks.

Listing 1 presents an illustrative example of EDPs. The Java method `evalBMI` classifies the weight of a person as underweight, healthy weight, or overweight, given its body mass index (BMI) calculated by `calcBMI`, which might throw an `ArithmeticException`.

Listing 1: Source code of `evalBMI`.

```
1 public void evalBMI(Dialog ud, float mass, float height) {
2  winlayout.setStyle("default");
3  float bmi = 0; 01
4  try {
5    bmi = calcBMI(mass, height); 02
6    String msg = String.valueOf(bmi); 02
7    ud.print(msg); 02
8    if (bmi < 18.5) 02
9      ud.print("Underweight"); 03
10   else
11   if (bmi < 25) 04
12     ud.print("Healthy weight"); 05
13   else{
14     ud.print("Overweight""); 06
15     ud.getLayout().setColor("red"); 06
16   }
17 } 07
18   catch (ArithmeticException e) {
19   ud.print("Height must be greater than zero"); 08
20 } catch (NullPointerException e) {
21   e.printStackTrace(); 09
22 } finally {
23   ControlBoard.addBMI(bmi); 10 – 11
24 }
25 } 12
```

Notice that `evalBMI` has a `try-catch-finally` construct with two exception handlers: each `catch` block is an exception handler whose argument declares the type of exception that the handler can treat. The first handler catches an unchecked exception: `ArithmeticException`. The second handler catches another sort of unchecked exception: `NullPointerException`. The `finally` statement ensures that all instructions within its block are executed regardless of what happens in the `try` block.

In Java, unchecked exceptions inherit from either `RuntimeException` or `Error`. In general, good programming practices can avoid raising unchecked exceptions. Therefore, the compiler does not force the programmer to handle such type of exceptions, albeit it is a common practice. Notice, for example, that both `winlayout.setColor` (line 2) and `ud.print` (lines 7, 9, 12, and 14) instructions may throw a `NullPointerException`, but only the latter are within a `try` block. As oppose to unchecked exceptions, the compiler force the programmer to catch or propagate checked exceptions.

We represent the methods under test as control-flow graphs (CFGs) [11, 17]. CFGs are directed graphs in which each node usually represents a block of instructions without flow deviation (i.e., a basic block) and directed edges represent transitions (i.e., unconditional branch or jump) in the control-flow.

Figure 1 shows the CFG generated for `evalBMI`. The numbers after each instruction in Listing 1 indicate their corresponding node in the CFG. Nodes related to `try`, `catch`, and `finally` blocks are also shown. Dashed edges represent branches that are executed when some exception occurs. The exception handling mechanism of the Java language is conservative: it considers that any instruction within a `try-catch-finally` block may throw an exception. Thus, there are edges from all nodes within the `try` statement to the exception handling nodes.

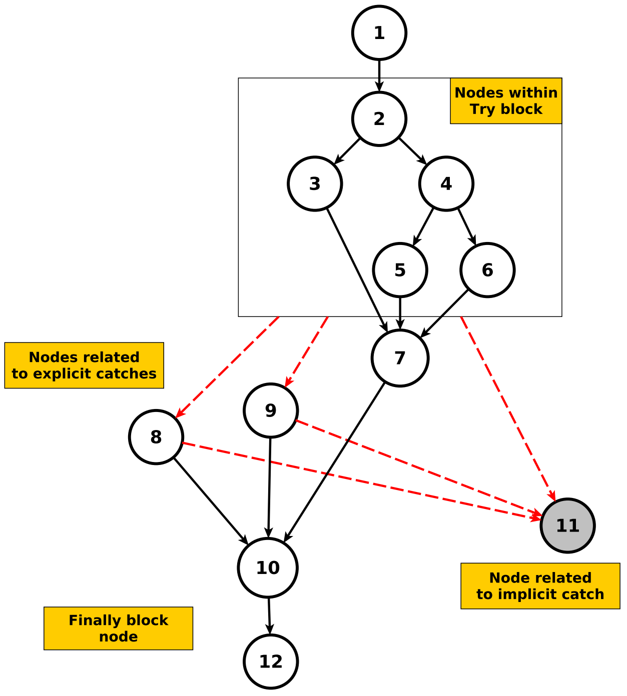
1
2
3
4
5
6
7
8
9
10
11
12
Nodes within Try block
Nodes related to explicit catches
Node related to implicit catch
Finally block node

Fig. 1: CFG for `evalBMI`.

Notice that nodes 2–6 have edges to nodes 8, 9, and 11; we use only one edge from the `try` box for improving the legibility of the graph. The exception handling notation we use is adapted from the notations proposed by Sinha and Harrold [13] and Vincenzi et al. [14].

Nodes within the `try` block can reach any exception nodes (8, 9, and 11), depending on the type of the thrown exception. Node 8 catches an `ArithmeticException` and node 9 captures a `NullPointerException`. Node 11 is an implicit catch included by the compiler to capture any uncaught[2] exception, even those that might be thrown within `catch` statements. Node 10 represents the `finally` statement, which is reached by the exception nodes (8 and 9) and by node 7. It is important to mention that Node 11 is a copy of the `finally` block that was automatically created by the compiler to assure that the instructions under the `finally` block are executed even when an uncaught exception is thrown. Node 11 is also an exit node since it throws an exception.

During symbolic execution, we adopted a breadth-first algorithm whose goal is to generate paths that cover all branches of a CFG. Table I shows all paths of `evalBMI`, which may be either *(i)* EDP, when the path includes an exception edge (dashed), or *(ii)* EFP.

TABLE I: EDPs and EFPs of `evalBMI`.

Path ID	EDP
1	{1, 2, 11}
2	{1, 2, 4, 11}
3	{1, 2, 3, 11}
4	{1, 2, 9, 11}
5	{1, 2, 8, 11}
6	{1, 2, 4, 5, 11}
7	{1, 2, 4, 6, 11}
8	{1, 2, 8, 10, 12}
9	{1, 2, 3, 8, 10, 12}
10	{1, 2, 4, 8, 10, 12}
11	{1, 2, 4, 6, 8, 10, 12}
12	{1, 2, 4, 5, 8, 10, 12}
13	{1, 2, 9, 10, 12}
14	{1, 2, 3, 9, 10, 12}
15	{1, 2, 4, 9, 10, 12}
16	{1, 2, 4, 5, 9, 10, 12}
17	{1, 2, 4, 6, 9, 10, 12}
	EFP
18	{1, 2, 3, 7, 10, 12}
19	{1, 2, 4, 5, 7, 10, 12}
20	{1, 2, 4, 6, 7, 10, 12}

Symbolic execution approaches yield path constraints based on the constraints explicitly declared along the underlying path. However, there is no constraint associated with the exception edges. Therefore, approaches that want to generate test data to exercise EDPs must implement mechanisms to derive constraints from exception handling mechanisms, otherwise they will not be covered. For instance, edge 2–9 is executed only when a `NullPointerException` is raised. In such a case, an analysis of the code would show that the constraint `(ud==null)` would result in the execution of that path.

Given that the constraints that cover EDPs are not explicit in the code, many symbolic execution techniques ignore EDPs, thereby building path constraints that take into account only explicit constraints. A clear advantage of not dealing with implicit constraints is that the number of paths to be processed is low, which can speed up test data generation. The disadvantage, however, is that many possible execution paths remain uncovered.

III. Handling EDPs and Uncaught Exceptions

Current symbolic execution tools and approaches do detail whether or not and how they handle EDPs. Therefore, we devised four possible approaches to generate test data that cover EDPs and uncaught exceptions in order to increase the likelihood of covering more faulty scenarios. The impact of using these approaches is discussed in Section IV.

A. Analyzing `try-catch` Statements: Single Constraint

This approach analyzes instructions declared within `try` statements and tries to identify constraints that would throw exceptions caught by the `catch` clauses. For example, if the target exception is a `NullPointerException`, instructions that access methods and fields of an object are considered. Even though there are several instructions that could raise the target exception, only one constraint is selected for each block of instructions.

The advantage of this approach is that test data is generated considering both explicit and implicit constraints. The drawback is the increase in the number of path constraints. In addition, complex analysis techniques are required to derive constraints from exception handling mechanisms, mainly because each type of exception requires different constraints to be raised. Also, selecting only one constraint to raise the target exception may not be enough since it can be unsolvable, i.e., it is not possible to find a concrete solution to satisfy all constraints. For instance, if the constraint `(ud==null)` is selected to execute the exception edge 6–9, path {1, 2, 4, 6, 9, 10, 12} would remain uncovered since edges 2–4 and 4–6 are executed only if `(ud!=null)`.

B. Analyzing `try-catch` Statements: Multiple Constraints

The analysis performed in this approach is similar to the previous approach (Subsection III-A). However, instead of selecting only one constraint for each block or node, all possible constraints are used according to the target exception. In such a case, one new path constraint is generated for each new constraint identified. For instance, two path constraints would be generated for EDP $\{1, 2, 4, 6, 9, 10, 12\}$: one considering the constraint `(ud== null)`, and other considering the constraint `(ud.getLayout()==null)`.

The advantage of this approach is that it explores all possible situations in which an exception can be raised within an exception handling environment. Even though many of the path constraints generated may turn out to be unsolvable, choosing several constraints increases the chances of finding test data to cover the target EDP. Nevertheless, the main drawback is that the number of path constraints yielded for each EDP may be too high, leading to path explosion.

C. Beyond `try-catch` Statements

Both aforementioned approaches are based in the fact that, in theory, programmers generally employ exception handling mechanisms in scenarios where exceptions are more likely to be thrown. According to Cabral and Marques [4], however, programmers do not catch enough unchecked exceptions making applications crash even on minor error situations. Therefore, deriving constraints only from instructions within `try-catch-finally` blocks may let several faulty scenarios uncovered due to how programmers write their code [4, 12].

[2]An uncaught exception is an exception thrown that is not captured by an exception handling mechanism. In Java, only unchecked exceptions may be uncaught.

In this context, we devised a thorough approach that identifies constraints that traverse EDPs by analyzing exception handling mechanisms and also constraints aimed at raising uncaught exceptions that can be raised outside the boundaries of `try-catch` statements. Yet considering a `try-catch` environment, the constraints generated are not limited to the declared exception. For each constraint identified, the underlying path constraint is replicated and the new constraint is added.

The main advantage of this approach is that it allows for yielding path constraints to generate test data that traverse EDPs and also raise uncaught exceptions. The test data generated are not limited to exercise EDPs abstracted from exception handling mechanisms. Rather, the test data generated by this approach exercises every possible faulty scenario by activating all possible exceptions.

The main drawback of this approach is the huge number of path constraints to process. This alternative can clearly aggravate the path explosion problem. Furthermore, the complexity of finding a concrete constraint to raise an exception is greater in this context since the analysis must take into account any type of exception, not only the target exceptions within `catch` statements.

Another drawback of yielding a huge number of path constraints is that many of them may be unsolvable. In such cases, resources will be spent to process constraints that will not generate any test data. One possible solution to mitigate this problem is to apply static analysis techniques to identify and eliminate unsolvable path constraints prior to sending them to the constraint solver.

D. Beyond `try-catch` Statements: An Optimized Version

We devised an optimized version of the previous approach (Subsection III-C) in which new path constraints are only created when the new constraint that activates an exception follows these rules: *(i)* it does not contradict any constraint in the underlying path constraint; *(ii)* it is not yet in the underlying path constraint; *(iii)* it does not raise an uncaught exception before reaching the block where the target exceptions is supposed to be thrown. The advantage of this approach is that the amount of path constraints is kept in check.

IV. Study of the Overhead Brought by Approaches to Generate Test Data to Faulty Scenarios

A. Study Setup and Procedure

The main goal of this study is to investigate the impact of generating unit test data to cover EDPs and uncaught exceptions, taking into account the number of path constraints for each of the four approaches presented above. Specifically, we want to find out the overhead brought on the number of path constraints.

To perform such an investigation, we selected a third party benchmark named SF100 to be the object of our investigation [8]. SF100 is made up of a collection of 100 open source Java projects that differ considerably in size, complexity, and application domains. Altogether, these 100 Java projects contain 18,344 classes and 136,156 methods.

We used a tool called CP4SE (Constraint Profiling for Symbolic Execution) [7] to analyze the SF100 benchmark. CP4SE can symbolically execute a program under test based on its bytecode and provide the path constraints generated for each execution path. It uses a breadth-first search to find all paths considering only one loop iteration and also the aim of covering all branches. In order to tailor CP4SE for our purposes, we implemented the four test data generation approaches to cover EDPs and uncaught exceptions discussed in Section III. We adopted CP4SE as a static analysis tool, focusing on the analysis of path constraints associated to EFPs and EDPs. It is worth mentioning that the generation of test data is out of the scope of this paper.

In this study, we investigate the effects of generating path constraints related to four out of the seven most common exceptions used in the Java language according to Cabral and Marques [4]. The four exceptions we investigated are presented as follows:

- `NullPointerException`: instructions such as `obj.field` or `obj.method(...)` generate the constraint `(obj==null)`, where `object` is a program element (e.g., variable or method return) whose type is an object.
- `NegativeArraySizeException`: instructions such as `array = new type[size]` generate the constraint `(size<0)`, where `size` is any numeric element (e.g., variable, method return or arithmetic expression).
- `ArrayIndexOutOfBoundsException`: instructions such as `array[i]` generate the constraint `(i>=array.length)` or `(i<0)`, where `array` is any array structure (e.g.,variable or method return) and `i` is any numeric element.
- `ArithmeticException`: instructions such as `(x/y)` generate the constraint `(y==0)`, where `y` is a numeric element (e.g, variable or arithmetic expression).

It is important to highlight that the first two approaches employed (Subsections III-A and III-B) only identify constraints for specific exception. If an instruction such as `(x/y)` is within a `try` statement caught by a `NullPointerException`, no constraint will be identified. On the other hand, all types of constraints are identified in handling mechanisms that catch generic exceptions such as `java.lang.Exception` or `AnyException`.

It is also worth mentioning that all instructions are analyzed by CP4SE after the symbolic execution of the program under test. Thus, CP4SE only identifies constraints to instructions that follow the structure defined for each exception. For example, consider a method with the instruction `(x/y)`, but the following assignment is always executed before: `y=5`. In such a case, the constraint `(y==0)` is not identified since the analyzed instruction becomes `(x/5)` after symbolic execution. The same principle holds for the other types of instructions.

TABLE II: Path constraint overhead.

	No EDPs	Approach A		Approach B		Approach C		Optimized Approach C	
Exception–Approach	**# PC**	**# PC**	**Overhead**	**# PC**	**Overhead**	**# PC**	**Overhead**	**# PC**	**Overhead**
NullPointerException	115,305	134,490	16.6%	149,991	30.1%	646,186	460.40%	295,701	156,5%
ArrayIndexOutOfBoundsException	115,305	125,099	8.5%	127,560	8.5%	191,414	66%	158,078	37,1%
ArithmeticException	115,305	124,150	7.7%	124,160	7.7%	126,758	9.9%	126,487	9,7%
NegativeArraySizeException	115,305	124,724	8.2%	124,768	8.2%	136,906	18.7%	130,580	13,2%
All four exceptions	115,305	134,560	16.7%	154,085	33.6%	724,233	528.1%	323,185	180,3%

B. Results

We executed CP4SE several times to generate path constraints to SF100 according to the four approaches presented in Section III and the target exceptions commented in Section IV-A. Each run of CP4SE considered a particular approach and a particular exception. Table II summarizes the results of these runs.

Rows 1 to 4 of Table II show the results for each type of exception individually, while row 5 presents the results of the four exceptions combined. The columns show the results obtained by the execution of the four approaches discussed in Section III. The first column shows how many path constraints (#PC) were identified using CP4SE when no EDP or faulty scenario is considered. This particular information is used in the rest of the table as the basis to measure the overhead brought by the implemented approaches. For each approach, we present the number of path constraints (#PC) identified and the overhead measure in percentage.

The results show that the `NullPointerException` type brings more overhead than the other three types we investigated. The overhead ranges from about 16% to 30%, when only exception handling mechanisms are analyzed. On the other hand, the overhead is increased by up to 460% when all blocks of instructions are considered. The optimizations introduced by the approach described in Subsection III-D seem to be a possible solution to this problem since the overhead dropped from 460% to 156%.

Although the number of path constraints with array elements is low, as in previous results [7], the number of instructions that uses arrays is high. As a result, the overhead brought by exceptions related to arrays is relatively high (up to 66%). The overhead regarding arithmetic exceptions is low, which is consistent with the fact that complex and nonlinear arithmetic expressions involving divisions are more frequent in specific applications according to Barr et al. [2], such as mathematical and scientific applications.

The overhead brought by the constraints identified within a `try-catch-finally` block (Subsections III-A and III-B) is significantly lower than the overhead brought by the analysis of all instructions (Subsections III-C and III-D). This means that there are several scenarios in which an uncaught exception may be thrown. This seems to agree with the analysis of Cabral and Marques [4], in which they state that, since programmers are not forced by the compiler, they do not catch unchecked exceptions properly, making applications crash even on minor error situations.

When all exceptions are considered, the overhead brought by Approach A (Subsection III-A) is not high. As Cabral and Marques [4] remark, developers tend to catch generic exceptions. In such a case, only one exception is enough to exercise that particular path. When Approach B (Subsection III-B) is used, the overhead is a bit higher (around 33%). Considering Approach C (Subsection III-C), the overhead is extremely high (528%), which exacerbates the path explosion issue. However, when Approach D (Subsection III-D) is considered, the overhead is about 180%.

The results of our investigation show that, even considering only four types of exceptions, the overhead of thoroughly generating test data for most exception scenarios is prohibitive for many symbolic execution approaches. Practitioners and researchers must perform a carefully analysis in hopes of deciding which approach or which combination of approaches should be used in each situation.

V. Related Work

Researchers have been investigating how exception handling mechanisms have been used by programmers and how these mechanisms can be tested properly. Cabral and Marques [4] carried out a quantitative study on how programmers use exception handling mechanisms. They looked at 32 projects, written in Java and .NET, and found that although the apt exceptions are thrown in most situations, programmers are not concerned with writing specialized handling code. Hindered by inflexible handling mechanisms [12], programmers fall back on writing generic exception handlers which are empty, exclusively dedicated to re-throw exceptions, or halting the method or program.

Xiaoquan et al. [15] proposed a static method to detect faults related to erroneous exception handling in Java programs. Their method combines two types of analysis: a forward flow-sensitive analysis to detect unsafe use of variables and a backward path feasibility analysis to prune false positives.

Few studies have been conducted to understand the characteristics of real-world software regarding exception handling from a symbolic execution point of view [7, 10, 16]. Xiao et al. [16] investigated path explosion in the context of dynamic symbolic execution. They analyzed the characteristics of loops in 16 open source projects written in the C# language, but their study focused on the characteristics of loops rather than their overall presence and relation with exceptions.

In a previous paper [7], we studied the distribution of path explosion, constraint complexity, dependency, and EDPs over the SF100 benchmark [8]. Regarding path explosion, the impact caused by loops and nested loops was investigated. Concerning

EDPs, we found out that 36% of the analyzed methods of SF100 had at least an EDP, but the impact on the number of path constraints generated has not been analyzed.

The main difference between our study and the related research is the investigation of how generating test data to cover EDPs can impact path explosion according to three different approaches. To the best of our knowledge, no other studies on this subject has been carried out.

VI. Concluding Remarks

Although symbolic execution has been extensively investigated as a promising approach for test data generation, little research has taken into account the generation of test data that cover exception-related paths. Despite the fact that many paths in a program are *exception-dependent* (i.e., EDPs), most approaches have focused on *exception-free* paths (i.e., EFPs). In this paper, we investigated this topic by looking at the increase in the number of path constraints resulted from four different test data generation approaches that cover exception scenarios.

The results would seem to suggest that the overhead caused by common exceptions, as `NullPointerException` and `ArrayIndexOutOfBoundsException`, is high, while the overhead caused by the other two exceptions is relatively low. When the four investigated exceptions are considered together, the overhead may be manageable if constraints are derived only from `try-catch-finally` statements (around 33%). However, it may be impracticable if constraints are derived from all instructions of the program under test (around 180%).

In conclusion, we believe that practitioners and researchers that want to generate test data tailored to cover exception-based scenarios should evaluate the trade-offs of using a thorough approach: generating test data for most likely exception-based scenarios results in a considerable overhead; on the other hand, focusing only on instructions declared within exception handling mechanisms or eschewing certain exceptions (e.g., `NullPointerException`) may leave many faulty scenarios uncovered.

Acknowledgments

The authors would like to thank the financial support provided by CAPES (BEX 1714/14-7), FAPESP (2014/08713-9), and CNPq (445958/2014-6).

References

[1] S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn. An orchestrated survey on automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[2] E. T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. In *Proc. of the 40th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 549–560, New York, NY, USA, 2013.

[3] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 122–131, Piscataway, NJ, USA, 2013. IEEE Press.

[4] B. Cabral and P. Marques. Exception handling: A field study in java and .net. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, pages 151–175, Berlin, Heidelberg, 2007. Springer-Verlag.

[5] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[6] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: Preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1066–1071. ACM, 2011.

[7] M. M. Eler, A. T. Endo, and V. H. S. Durelli. Quantifying the Characteristics of Java Programs that May Influence Symbolic Execution from a Test Data Generation Perspective. In *The 38th Annual Int. Computers, Software & Applications Conference*, pages 181–190, 2014.

[8] G. Fraser and A. Arcuri. Sound Empirical Evidence in Software Testing. In *Proc. of the 2012 Int. Conf. on Software Engineering*, pages 178–188, 2012.

[9] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[10] X. Qu and B. Robinson. A Case Study of Concolic Testing Tools and their Limitations. In *International Symposium on Empirical Software Engineering and Measurement*, pages 117–126, 2011.

[11] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, 1985.

[12] M. P. Robillard and G. C. Murphy. Designing Robust Java Programs with Exceptions. *ACM SIGSOFT Software Engineering Notes*, 25(6):2–10, 2000.

[13] S. Sinha and M. Harrold. Criteria for Testing Exception-Handling Constructs in Java Programs. In *Proc. of the Int. Conf. on Sw Maintenance*, pages 265–274, 1999.

[14] A. M. R. Vincenzi, M. E. Delamaro, J. C. Maldonado, and W. E. Wong. Establishing structural testing criteria for java bytecode. *Software Practice & Experience*, 36 (14):1513–1541, 2006.

[15] X. Wu, Z. Xu, and J. Wei. Static Detection of Bugs Caused by Incorrect Exception Handling in Java Programs. In *11th International Conference on Quality Software (QSIC)*, pages 61–66, 2011.

[16] X. Xiao, S. Li, T. Xie, and N. Tillmann. Characteristic studies of loop problems for structural test generation via symbolic execution. In *Proc. 28th IEEE/ACM Int. Conf. on Automated Software Engineering*, November 2013.

[17] H. Zhu, P. A. V. Hall, and J. H. R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.