2014 IEEE 38th Annual International Computers, Software and Applications Conference

Quantifying the Characteristics of Java Programs that May Influence Symbolic Execution from a Test Data Generation Perspective

Marcelo M. Eler
EACH/USP
São Paulo - SP - Brazil
E-mail: marceloeler@usp.br

Andre T. Endo
UTFPR
Cornélio Procópio - PR - Brazil
E-mail: andreendo@utfpr.edu.br

Vinicius H. S. Durelli
ICMC/USP
São Carlos - SP - Brazil
E-mail: durelli@icmc.usp.br

***Abstract*—Testing plays a key role in assessing the quality of a software product. During testing, a program is run in hopes of finding faults. As exhaustive testing is seldom possible, specific testing criteria have been proposed to help testers to devise test cases that cover the most relevant faulty scenarios. Manually creating test cases that satisfy these criteria is time consuming, error prone, and unwieldy. Symbolic execution has been used as an effective way of automatically generating test data that meets those criteria. Although this technique has been used for over three decades, several challenges remain, such as path explosion, precision of floating-point data, constraints with complex expressions, and dependency of external libraries. In this paper, we explore a sample of 100 open source Java projects in order to analyze characteristics that are relevant to generate test data using symbolic execution. The results provide valuable insight into how researchers and practitioners can tailor symbolic execution techniques and tools to better suit the needs of different Java applications.**

***Keywords*—Software testing; symbolic execution; test data generation;**

I. Introduction

Testing plays a key role in assessing the quality of a software product. During this activity, a program under test is run against test cases with the goal of finding unrevealed faults [1]. However, a well-known theoretical limitation of testing is that it can only indicate the presence of faults, not their absence. Broadly speaking, the problem of finding all faults in a given program is undecidable. Therefore, testers have been resorting to testing criteria, which help testers decide what inputs are more likely to uncover different types of faults.

Structural testing is one of the most known testing technique. This technique proposes criteria to help testers judge the thoroughness of a test set by exercising the structure of the program under test. Two widely used structural testing strategies are control- and data-flow. Manually generating test data to satisfy these criteria, it turns out, is time consuming, error prone, and unwieldy. Thus, several approaches have been proposed to automate this process [2].

Symbolic execution and constraint solving have been used for more than three decades as effective techniques for generating test data that achieve a high coverage of control-flow criteria requirements [3], [4]. The general idea behind symbolic execution is to represent the program elements as functions of symbolic input values [4], [5]. Next, constraint solvers are used to generate concrete input values (test data) that satisfy a set of constraints related to execution paths.

Although symbolic execution has come a long way in the last decades, many challenges remain. *Path explosion* is one of the key challenges: symbolically executing a large number of paths entails high computational overhead [6]. Moreover, long paths tend to yield large constraint sequences and, though the performance of constraint solvers has been improved in recent years, large constraint sequences still hurt performance [4]. The *constraint complexity* is also an issue. The data types of the constraint elements and arithmetic expressions may affect the efficiency of the constraint solvers [7], [8]. Approaches also have to deal with *dependency*. Several constraints are related to method calls, whose values may not depend on symbolic input values or object configurations. Approaches have used different techniques for solving such type of constraints [6]. In addition, there are many *exception-depended paths*, that can only be executed when a given exception is thrown. This issue has not been covered in the literature yet, but it is a relevant matter since it cannot be solved by recent approaches.

This paper describes an investigation that sheds light into the nature and frequency of elements that affect symbolic execution from a test data generation perspective. Particularly, we investigate the distribution of loops and inner loops; data types; method calls; and constraints with exception declarations to quantify factors that may influence *path explosion*, *constraint complexity*, *dependency* and *exception-depended paths* issues, respectively. Notice that, in this paper, we have not investigated constraint complexity concerning mathematical expressions.

The overarching motivation for this research is to provide a greater understanding of the characteristics of these elements in real-world Java programs. We used a representative sample of 100 open source projects [9][1] in our investigation. The results of our investigation provide valuable insight into how researchers and practitioners can evaluate the adequacy of current approaches and tailor symbolic execution techniques and tools to better suit the needs of different Java applications.

Although there are previous work on analyzing the characteristics of programs that have an impact on symbolic

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

0730-3157/14 $31.00 © 2014 IEEE
DOI 10.1109/COMPSAC.2014.26

execution [10], this paper presents the results of a large scale empirical study in a different light. Moreover, we claim that several studies should be performed to understand how is the distribution of the characteristics that affect symbolic execution from a test data generation perspective.

The remainder of this paper is organized as follows. Section II introduces the background and discusses related work. Section III illustrates the data extraction procedure we carried out, along with an example. Section IV describes the study setting. Section V brings the results along with analyzes of the data. Section VI discusses some impacts in the area of symbolic execution for software testing. Finally, Section VII makes concluding remarks and sketches future work.

II. Background and Related Work

During symbolic execution, a program is analyzed and symbolic executed to determine what inputs cause the program to run each of its possible paths. Such an analysis is performed as follows. First, a constraint sequence is generated for each execution path. A constraint sequence is a logical expression connecting all constraints that should be satisfied to execute a particular path. Constraints usually present four types of elements: variables, method calls, constants, or exception declarations. We consider exception declarations as constraint elements because certain paths are run only when an exception is thrown. Next, specific techniques may be executed to handle constraint elements that are method calls and complex types. The constraint sequences are thus sent to a constraint solver that, if possible, finds solutions for each variable in the constraint sequence. If the program is run using the values produced by the solver, then the program will follow the same path represented by the constraint sequence.

Since the seminal work of King in 1976 [5], symbolic execution has been used to generate test data for control-flow criteria [3], [4]. Although this research area has advanced over the past years due to better constraint solvers and the introduction of hybrid approaches (e.g., concolic testing [11]), symbolic execution still poses several challenges when applied to software testing [4], [6], [7], [12], such as *(a)* path explosion, *(b)* complexity of constraints, *(c)* dependency, and *(d)* paths triggered by exceptions.

a) Path explosion: Path explosion is one of the key challenges in this field [6]. A workaround to this issue is to use algorithms that generate paths that cover all branches by going through loops only once. Yet, the number of constraints to be solved in the resulting paths may be huge. This can overwhelm the constraint solver and reduce the performance [4]. Furthermore, constraint sequences might be unsolvable because they demand more than one loop iteration. In such case, methods with several inner loops tend to generate a huge number of paths until reaching a solvable constraint sequence.

b) Constraint complexity: the complexity of a constraint may be related to the data types of their elements or to the complexity of the arithmetic expressions. Constraints with primitive types (e.g., fixed-point data types) are better handled by symbolic execution approaches than complex types (e.g., arrays and objects) [7]. While the initial approaches only dealt with primitive types (specially integers), we can now find approaches for complex structures like arrays and objects [10], [13]–[16]. Usually, complex types are reduced and expressed as primitive types or even recursive structures. As for primitive data types, constraint solvers handle integers more properly than floating-point numbers depending on the required precision [17]. Regarding the arithmetic expressions, constraint solvers may not deal with nonlinear mathematical constraints that are undecidable or very hard to solve as, for instance, a seventh degree polynomial [8].

c) Dependency: constraint solvers are unable to tackle constraints that depend on a method call's response because they cannot symbolically reason about method invocations. A solution is to integrate the caller and the callee. This is possible for calls to methods within the same class or for methods in different classes of the same project, whose code is available for integration. However, calls for external libraries and/or native functions are harder to be handled in this fashion [6]. For Bytecode-based languages this may not be an issue since it may be possible to process libraries and external projects at Bytecode-level [10]. But it would not be possible, for example, for remote calls such as web services invocations. A drawback is that the methods integration may bring other challenges due to high degrees of coupling or recursion.

d) Exception-depended paths: some paths can be executed only when exceptions are thrown. Methods with exception handling code tend to have a large number of paths since extra branches related to exception handling operations are included. The branches can increase if the exception handling code declares a Java `finally` statement, for example. As far as we are concerned, this issue has not been covered in the literature of symbolic execution for testing.

So far, few studies with real-world software have been conducted to evaluate symbolic execution approaches to generate test data. Cadar and Sen [8] provide a preliminary assessment of the symbolic execution for software testing in academia, research laboratories, and industry. They present characteristics of different techniques and discuss how mature symbolic execution tools can handle some of the aforementioned issues. However, they did not investigate how frequently these issues are in real-world programs. Pasareanu and Visser [7] also present a survey on symbolic execution approaches without evaluating the characteristics of real settings.

To our best knowledge, only Qu and Robinson [10] had studied the frequency of symbolic execution limitations applied in large scale software. They investigated the frequency and the density of floating-point numbers, pointers (objects), native calls, symbolic offsets and bitwise operations over the functions that present such limitations to symbolic execution. There are three main differences between the study of Qu and Robinson [10] and ours. First, Qu and Robinson [10] aimed to study concolic testing tools and their limitations, counting the elements statically. In this study, we only considered factors (e.g., data types and method calls) when they impact on the constraint sequence (as a consequence, in the test data generation process). Second, we used a third-party benchmark (SF100) with 100 Java projects with different sizes, while Qu and Robinson [10] selected six large projects written in C and Java (industrial and open-source). Third, we also analyze factors that may influence *path explosion* and *exception-depended paths*. As for *dependency* issue, we split method calls into three types: inner, inter and external, which may be different

depending on the target implementation. Moreover, concerning *constraint complexity*, we investigated the distribution of other data types besides floating-point numbers.

III. Data Extraction

This paper presents an analysis performed to quantify characteristics that may influence the issues mentioned in Section II. We defined a thorough set of metrics to be extracted from the programs selected. The procedure adopted to perform symbolic execution and extract metrics are illustrated using the code snippet in Listing 1. The method `siteForPoint` belongs to the open source project `corina` [18], which is in the SF100 corpus. The data extraction can be divided into five steps: A) Build control-flow graph; B) Generate paths; C) Obtain constraint sequences; D) Perform symbolic execution; and E) Calculate metrics.

Listing 1. Method `siteForPoint` (extracted from [18]).

```
1 public Site siteForPoint(Projection r, Point p, int dist)
2                                 throws SiteNotFoundException {
3   if (siteHash == null) ❶
4     throw new SiteNotFoundException(); ❷
5
6   BufferedImage buf =
7   new BufferedImage(1,1,BufferedImage.TYPE_INT_ARGB_PRE); ❸
8   Graphics2D g2 = buf.createGraphics(); ❸
9   setFontForLabel(g2, view); ❸
10  int textHeight = g2.getFontMetrics().getHeight(); ❸
11  Point t = new Point(); ❸
12  Iterator iter = labels.getLocations(); ❸
13  Site returnValue = null; ❸
14
15  while (iter.hasNext()) { ❹
16    Location loc = (Location) iter.next(); ❺
17    r.project(loc, p2); ❺
18
19    if (p2.getZ() < 0) ❺
20      continue; ❻
21
22    Site s = SiteDB.getSiteDB().getSite(loc); ❼
23    String txt = s.getCode(); ❼
24    int textWidth = g2.getFontMetrics().stringWidth(txt);❼
25    Offset o = getOffset(loc); ❼
26    t.x = p2.getX() +
27          (o.dist * view.getZoom() * Math.sin(o.angle)); ❼
28    t.y = p2.getY() −
29          (o.dist * view.getZoom() * Math.cos(o.angle)); ❼
30    int left = t.x − (textWidth / 2 + EPS); ❼
31    int width = textWidth + 2 * EPS; ❼
32    int top = t.y − (textHeight / 2 + EPS / 4); ❼
33    int height = textHeight + EPS / 2; ❼
34
35    if (p.x >= left ❼ && p.x <= (left + width) ❽ &&
36          p.y >= top ❾ && p.y <= (top + height)) ❿
37      returnValue = s; ⓫
38  } ⓬
39
40  if (returnValue != null) ⓭
41    return returnValue; ⓮
42  else throw new SiteNotFoundException(); ⓯
43 }
```

A. Build Control-Flow Graph (CFG)

This step analyzes the method structure and builds a control-flow graph (CFG) [19], [20]. CFGs are directed graphs in which each node represents a block of instructions without flow deviation (i.e., a basic block). Directed edges represent transitions (unconditional branch or jump) in the control flow. In this study, CFGs are generated based on Java Bytecode.

Figure 1 shows the CFG generated for `siteForPoint`, which has 15 nodes and 20 edges. The numbers after each instruction in Listing 1 indicate the node in CFG. Notice that the CFG represents the `if` statement in line 35 using Nodes 7, 8, 9 and 10 because compound conditions like `a>0 && b<5` are split into several `if` statements in Bytecode. The CFG also stores pieces of information that are useful for symbolic execution. For instance, nodes can be associated to variable assignments and edges can be associated to constraints.

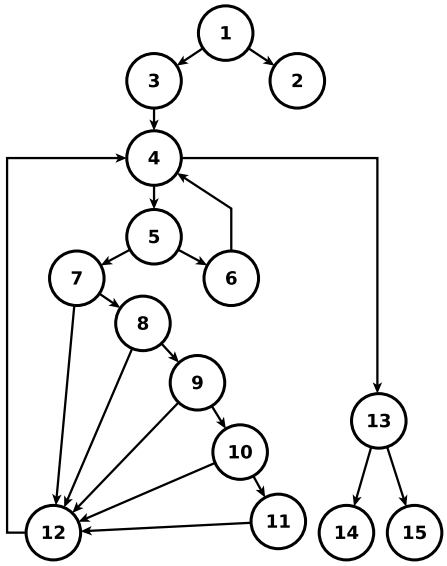

Fig. 1. CFG for method `siteForPoint`.

B. Generate Paths

The goal of this step is to identify a set of paths that covers all edges in th method's CFG. We reproduce a common goal in symbolic execution for software testing, i.e., to generate test data that cover all branches in the program under test (the all-edges criterion). To do so, the paths are generated using a spanning tree built through a breadth-first search. Figure 2 illustrates the tree constructed for the CFG in Figure 1. For each leaf L of the spanning tree, a path P representing the path from Node 1 (root) to L is generated.

Using this algorithm, method `siteForPoint` has nine paths that need to be symbolically executed in order to cover all-edges (the paths are labelled in the spanning tree in Figure 2). This strategy to path generation aims to produce shorter paths and, as a consequence, shorter constraint sequences. Notice that paths with loops take into account only one iteration. A drawback of putting a limit on the number of loop iterations is that some edges might end up uncovered because some constraints might not be satisfied.

C. Obtain Constraint Sequences

In this step, a constraint sequence is generated for each path identified in the previous step. Constraint sequences are built by connecting all constraints along a given path with operator `AND`. The constraint sequence generated for Path#9 is presented in Figure 3. Notice that the constraints in Figure 3 are related to Edges 1-3, 4-13, and 13-15.

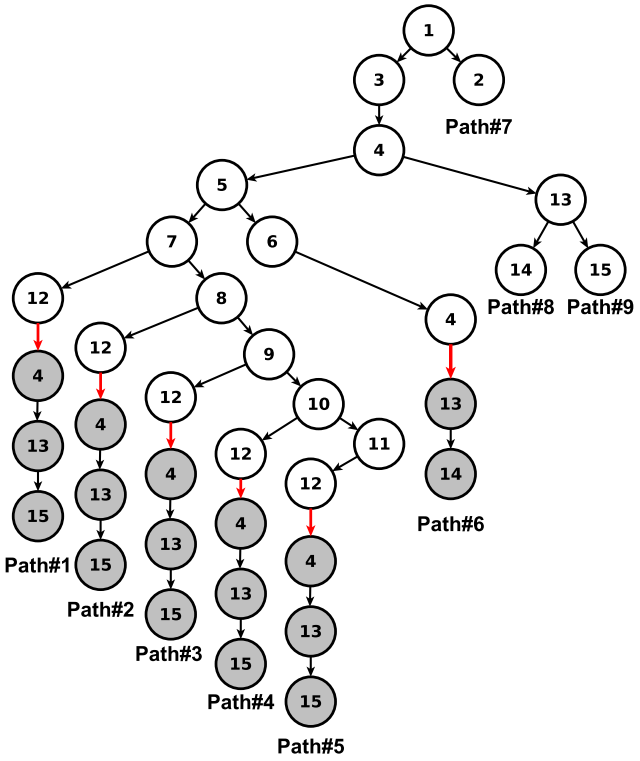

Fig. 2. Spanning tree to cover all-edges of `siteForPoint`.

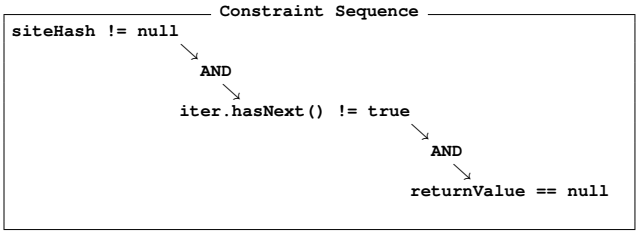

Fig. 3. Constraint sequence to cover Path#9.

D. Perform Symbolic Execution

We illustrate the symbolic execution using the constraints shown in Figure 3. First, we select the variables to be expressed as a function of other elements. Then, the target variable is replaced by values to which it was assigned during the execution of a specific path. For Path#9, for example, there is no assignment to `siteHash`, then the first constraint remains the same.

The variable `iter` is used in the constraint related to Edge 4-13. So, the procedure checks whether assignments to `iter` took place in the nodes preceding Node 4. In this example, there is one assignment to `iter` only at Node 3 (see Line 12 of Listing 1). Hence, the variable is replaced by its assigned value, i.e., `labels.getLocations()`. After the substitution of the variable for its previously assigned value, the constraint can be expressed as follows: `labels.getLocations().hasNext() != true`. The same procedure is adopted to this new constraint, starting from assignment of Line 11 of Node 3 back to the previous assignments in the path. In this example, there are no assignments to the variable `labels`.

The variable `returnValue` is also replaced by its assigned value: `null` (see Line 13 of Listing 1). The result of symbolically executing Path#9 is shown in Figure 4.

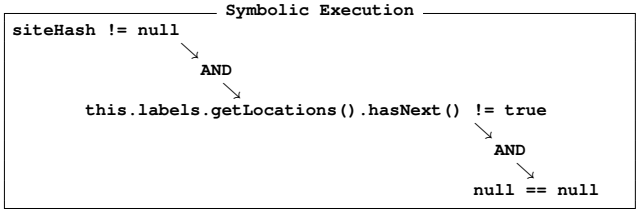

Fig. 4. Symbolically executing the constraints for Path#9.

Notice that there are two elements to which a solution should be found: variable `siteHash` and the return value of the method `hasNext()`. In this case, the return value of `hasNext()` depends on the return value of `getLocations()` that, in turn, depends on the state of variable `labels`. Constraint solvers can find solutions to variables, but they cannot find direct solutions for method calls. Constraint solvers are hampered by method calls because the result of some method calls depends on the internal state of one or many objects. In the example, solving the constraint sequence in Figure 4 heavily relies on finding the internal state of object `labels`.

E. Calculate metrics

Method `siteForPoint` is not too complex since its CFG has only 15 nodes and 20 edges. It also has only one loop. For each path of `siteForPoint`, we generated a constraint sequence similar to the one for Path#9 (shown in Figure 4). Then, we quantify the characteristics of each constraint sequence and extracted metrics related to its elements.

Table I shows the metrics we collected from the constraint sequences generated for the nine paths of `siteForPoint`. For each path, we show the constraint sequence size (CSS) [2]; the number of elements of the constraint sequence (CSE) [3]; the number of constants (CT); the number of variables (VAR); the number of exception declarations (EX); and the number of methods (MTH). Notice that we provide extra information regarding method calls instead of only providing the total number of calls. The total number of method calls are split into inner calls (InnCs), inter calls (IntCs) and external calls (ExtCs). It also brings the number of elements of each of these types: integers (INT)[4], floating-point numbers (FPN)[5], nulls (NL), strings (ST), objects (OT), and arrays (AT).

We illustrate the metric extraction using the constraint sequence generated for Path#9 (Figure 4). That constraint sequence has three constraints with eight elements. Four of these elements are literals (three `null` and a `true` value), two are variables, and two are method calls. Variables `siteHash` and `labels` are both objects. Method `getLocations()` is

[2]the number of constraints logically connected with `AND`

[3]the sum of the number of variables, constants, method calls and exception declarations

[4]it comprises the types: byte, char, short, int, long, and Boolean.

[5]it comprises the types: float and double.

TABLE I. METRICS EXTRACTED FROM PATHS OF SITEFORPOINT.

						Method calls				Types					
Path	**CSS**	**CSE**	**CT**	**VAR**	**EX**	**MTH**	**InnC**	**IntC**	**ExtC**	**INT**	**FPN**	**NL**	**ST**	**OT**	**AT**
1	1	2	1	1	0	0	0	0	0	0	0	1	0	1	0
2	3	8	4	2	0	2	0	1	1	2	0	3	0	3	0
3	3	8	4	2	0	2	0	1	1	2	0	3	0	3	0
4	5	15	6	4	0	5	0	3	2	4	2	3	0	6	0
5	6	47	10	14	0	23	2	11	10	11	7	3	0	26	0
6	7	94	18	26	0	50	4	23	23	24	12	3	0	55	0
7	8	121	23	35	0	63	6	27	30	32	17	3	0	69	0
8	9	157	32	45	0	80	8	31	41	46	22	3	0	86	0
9	9	161	31	46	0	84	8	34	42	46	22	2	0	91	0
Mean	5.7	68.1	14.3	19.4	0	34.3	3.1	14.6	16.7	18.6	9.1	2.7	0.0	37.8	0.0

an inter call and it returns an object type. Method `hasNext()` is a library call, which returns a `Boolean` value.

IV. STUDY SETTING

This section describes the experimental setup we used to analyze a sample of 100 open source Java projects. As mentioned, we analyzed these Java projects in order to quantify their characteristics that may lead to path explosion, constraint complexity, dependency, and exception-depended paths. Our research questions (RQs) reflect four recurrent issues in applying symbolic execution to software testing:

RQ$_1$: What is the distribution of the factors that may cause *path explosion*?
RQ$_2$: What is the distribution of the factors that may have a influence on *constraints complexity*?
RQ$_3$: What is the distribution of the distinct types of *dependency*?
RQ$_4$: What is the distribution of *exception-depended paths*?

A. Sample Selection

Aiming at selecting an unbiased sample of Java software, we based our study on the corpus of classes extracted from many projects (SF100 benchmark) [9]. The SF100 benchmark 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 8,784 classes and 136,156 methods. The largest project has 2,189 classes and the smallest has only one class.[6]

Our investigation considered only methods that have at least one constraint to be solved. Thus, although the SF100 benchmark comprises 136,156 methods, only 34,493 methods were analyzed. Figure 5 shows the distribution of the SF100 methods and it indicates that around 25% of the methods were the focus of our investigation. We removed from the initial sample 6,524 abstract methods and 95,139 methods with no branches.

We performed a further investigation to understand why approximately 70% of the methods have no constraint. We found that 43% of these methods are getters and setters and approximately 20% are constructors. This information would seem to suggest that it is common to find methods with no constraint in most projects since classes tend to have at least one get/set pair for each attribute and at least one constructor.

[6]All the experimental data is available at https://sites.google.com/site/andreendo/home/compsac2014.

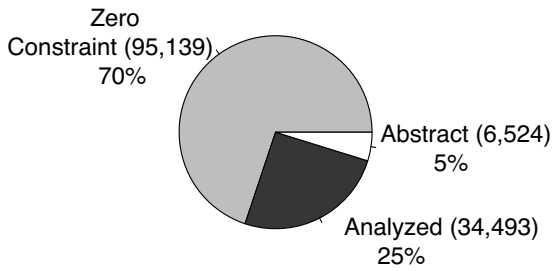

Fig. 5. Distribution of methods in the SF100 benchmark.

B. Supporting Tool

We developed a tool called `CP4SE` (Constraint Profiling For Symbolic Execution) to perform the static analysis and collect the metrics. `CP4SE` was implemented in Java and performs all analysis based on Java Bytecode. It receives a JAR (Java ARchive) file containing Java classes as input and executes the data extraction procedure presented in Section III for each method of each class. The static analysis performed by our tool is based on either each path of each method of a given class (as shown in rows 1 to 9 of Table I) or all paths of each method are taken into account and the mean value is reported (as shown in the last row of Table I). In this paper, we used the mean value of considering all paths of a method in our investigation.

C. Threats to Validity

We identified some threats to the validity of our study. The metric collected for each method is the mean of the metrics of all paths generated. Our procedure to generate paths for a method follows a breadth-first algorithm that aims to cover all branches of a CFG. The algorithm considers only one loop iteration and many paths may generate unsolvable constraints leaving branches uncovered. Some branches can be covered only if two or more loop iterations were considered. As statically defining which constraint sequences are unsolvable (they require longer paths) is an open issue, we decided to perform our analysis based on the paths generated by this algorithm.

In this study, we decided to perform symbolic execution by replacing constraint elements based only on direct assignments. We do not consider, for example, situations in which constraint elements are changed when used as parameters or even when the element is an object and invokes a method that changes its state.

We tried to remove any bias related to the selection of samples by adopting a third-party benchmark. As a drawback, we brought some threats reported by [9]. For instance, SF100 was built based on SourceForge projects and the results might not be equal if a different repository is considered. Moreover, the results are based on open source projects and cannot be generalized to industrial settings.

`CP4SE` performs all analyses based on the Java Bytecode. The Java Bytecode usually has more instructions because it has to handle implicit declarations and structures, such as casting and string handling. Moreover, the Java Bytecode works as a stack machine. Consequently, several instructions have to be

split into others, e.g., an `if` instruction with several constraints connected by logical operators (`AND`, `OR`).

V. Analysis of Results

As mentioned, the results presented in this section are based on methods that have at least one constraint. We discuss the results obtained for each of the RQs as follows.

A. RQ_1: *What is the distribution of the factors that may cause* path explosion*?*

We looked at two factors that may cause path explosion: *(i)* the number loops and *(ii)* the depth of inner loops. Table II shows the distribution of the number of loops. As highlighted in Table II, 75.15% (25,915) of the investigated methods have no loops, which means that they have no potential for path explosion, whereas 24.85% (8,578) have at least one loop structure. As shown in Table II, methods containing one loop are far more common (18.10%) than methods with two or more loops (6.75%). The method containing more loops had 48 loop occurrences and belongs to the `APBSmem` project.[7] It is worth mentioning that one loop is enough to lead to path explosion, but the presence of more than one loop can aggravate the problem.

TABLE II. Distribution of methods with loops.

Methods	Loops	% of analyzed methods
25915	0	75.15%
8578	>=1	24.85%
6238	1	18.10%
1431	2	4.15%
433	3	1.25%
180	4	0.50%
296	5 to 48	0.85%

Although the number of loops may indicate methods with potential for path explosion, the number of inner loops is also an important matter. Methods with several inner loops tend to present more complex logic and several numbers of execution paths, even considering only one loop iteration. Therefore, we also looked at the distribution of inner loops as shown in Table III. Of the 8,578 methods that have at least one loop, 6,369 (74%) have no inner loop, whereas 2,209 have at least one inner loop structure (26%).

TABLE III. Distribution of methods with inner loops.

Methods	Inner loops	% of methods with loops	% of analyzed methods
6369	0	74.00%	18.50%
2209	>=1	26.00%	6.5%
1450	1	17.00%	4.25%
408	2	4.80%	1.20%
152	3	1.80%	0.45%
75	4	0.90%	0.25%
124	5 to 35	1.50%	0.35%

B. RQ_2: *What is the distribution of the factors that may have an influence on* constraints complexity*?*

We analyzed the distribution of two factors that may influence the complexity of constraints: *(i)* the size and *(ii)* the data types in constraint sequences. A constraint sequence with many constraints typically takes longer to solve. From a test generation perspective, this may lead to bottlenecks when applying symbolic execution to generate test suites [4]. Therefore, we examined the size of constraint sequences.

Table IV gives an overview of the distribution of constraint sequence sizes. The mean value in Table IV indicates that most constraint sequences have around two constraints. However, since the data has outliers, the trimmed mean and median values in Table IV are more useful measures of central tendency than the mean. Moreover, because of the outliers, the median absolute deviation (MAD) is a more robust measure of statistical dispersion than the standard deviation (SD).

Another factor that influences the complexity of constraint sequences is the type of elements. Figure 6 provides an overview of the number of methods that have at least one element of the following types: integers (i.e., fixed-point data types), floating-point numbers, strings, arrays, and objects. From analyzing Figure 6, it is clear that integers and objects are the most frequent types. Most constraint sequences (66%) have constraints that include at least one object. Moreover, 61% of the analyzed methods have integers in their constraint sequences. The frequency in which floating-point numbers appear in constraint sequences is surprisingly low: 1.4%. As expected, the number of arrays is also low given that most programmers prefer the Java Collections Framework to arrays.

TABLE IV. Constraint size information.

Constraint Size Overview	
Mean	2.19
Trimmed	1.68
SD	2.29
Median	1.50
MAD	0.74
Min	1.00
Max	62.04

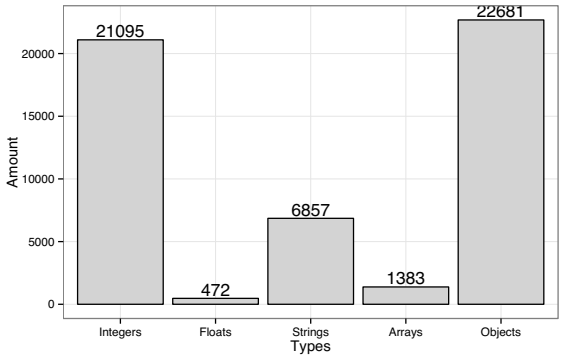

Fig. 6. Overview of all analyzed types.

Finding solutions to integer elements is easier than to floating-point elements due to precision requirements. Dealing with complex types (arrays and objects) is even harder for constraint solvers. To deal with complex types, many constraint-solving optimizations reduce complex types into simple types. Also, to speed up constraint solving, strings are often regarded as primitive[8] types. There are also constraint solvers that handle strings as a type of its own. In this paper, we consider strings an independent type apart from primitive and object types.

[7]For more details regarding each Java project see http://www.evosuite.org/sf100/ .

[8]The Java programming language has eight primitive data types: `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`

Based on these classifications, we carried out different analysis for each of the types presented in Figure 6. We calculated the relative frequency of each type in Figure 6. In this context, the relative frequency is defined as the ratio of the number of appearances of a given type to the total number of observations. That is, in the following histograms, the height of each rectangle equals the relative frequency of the corresponding type divided by the total number of elements. To improve the understanding of the histograms, we removed the methods that do not have any element of the analyzed type.

Figure 7 shows the histogram of the distribution of integer types. In our analysis, the following primitive types are considered integers: `boolean`, `byte`, `short`, `char`, `int`, and `long`. 13,398 (which corresponds to 39%) methods were removed from this analysis since they do not have any constraint containing integers types. As shown in Figure 7, 25% to 75% of the constraint elements in the methods considered are integers. Moreover, around 2,300 methods have constraint sequences that are made up of only integer types.

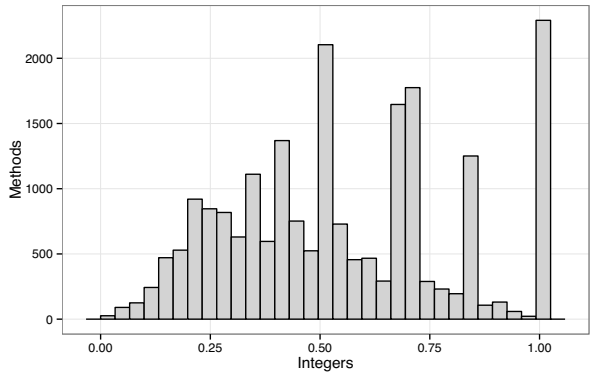

Fig. 7. Ratio of integers in constraint sequences.

Figure 8 shows the histogram of the distribution of floating point data types (i.e., `float` and `double`). We removed 98.6% (34,021) of all analyzed methods of the histogram since they do not have any constraint with these types. Methods with floating-point numbers represent only 1.4%. As indicated in Figure 8, for most methods considered, less than 30% of their constraint elements are floating-point.

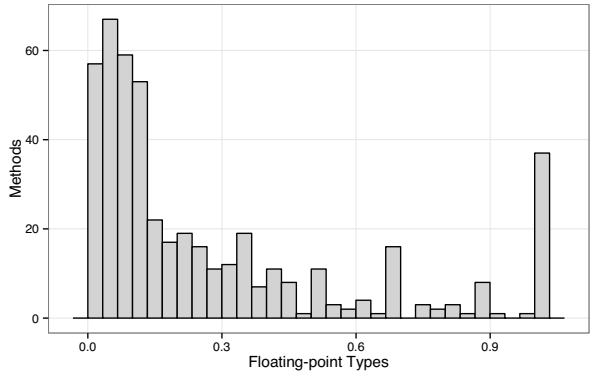

Fig. 8. Ratio of floating-point types in constraint sequences.

Figure 9 shows the histogram for the distribution of the string type. Our analysis shows that 6,857 methods have strings in their constraint sequences, which corresponds to 20% of the investigated methods. As shown in Figure 9, strings represent less than 40% of the constraint elements. According to our results, there is no method in which all constraint elements are strings.

Regarding complex types, we performed two different analyses: one for arrays and one for objects. Figure 10 shows the histogram for the distribution of array data types. Only 4% (1,383) of the analyzed methods appear in this histogram because 96% (33,110) do not have any array. Considering these methods, arrays represent less than 40% of their constraint elements most of the times. Methods with most array elements have up to 50% of their constraint sequence by array data types.

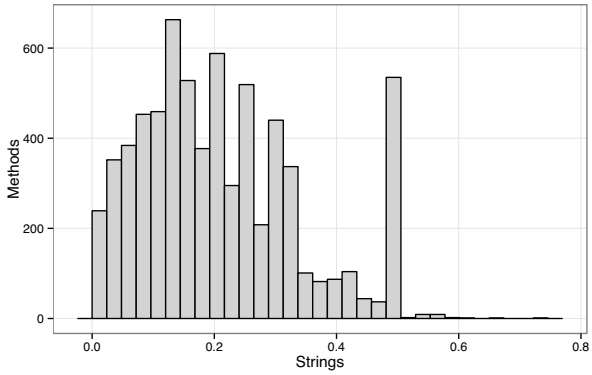

Fig. 9. Ratio of strings in constraint sequences.

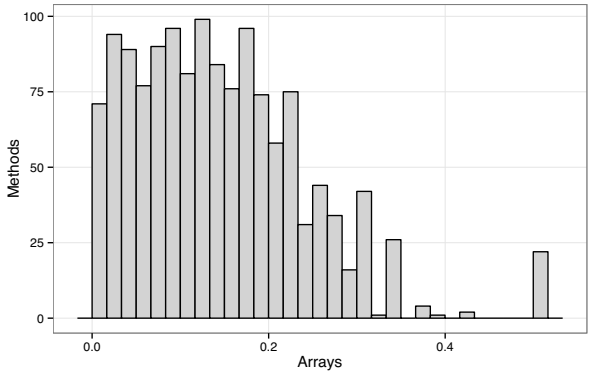

Fig. 10. Ratio of array data types in constraint sequences.

As expected, objects are the most predominant data type in constraints: 66% of the analyzed methods have objects in their constraint sequences. This is a direct effect of the programming language in which the chosen projects were written (i.e., Java). Figure 11 shows the histogram of the distribution of objects. 34% (11,812) of the analyzed methods were removed because they do not have constraints with objects. According to our results, 45% of the elements in constraint sequences are objects, on average.

C. RQ_3: What is the distribution of the distinct types of dependency (inner, inter, and external)?

There are 19,273 methods whose constraint sequences depend on a method call, which represents about 56% of the 34,493 methods in the SF100 benchmark. This sort of dependency may be expressed as inner, inter, or external calls. Methods may depend on all of these types of calls at the same time. For each type of method call analyzed, we computed the amount of method calls and divided it by the number of constraint elements. Then, we use histogram to highlight the distribution of these methods, removing methods that have no method calls in their bodies.

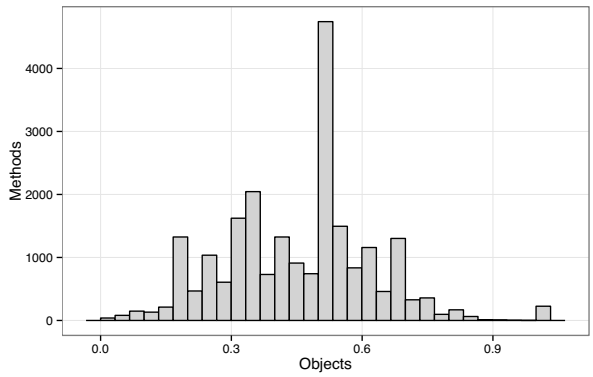

Fig. 11. Ratio of objects in constraint sequences.

Figure 12 shows the distribution of methods that have inner calls in their bodies. 12% (4,191) of the analyzed methods have inner calls, while 88% (30,302) do not. Considering these methods, in most cases, inner calls represent less than 40% of their constraint elements. Considering constraint sequences that have inner calls, on average (mean), 19% of the elements in these constraint sequences are inner calls.

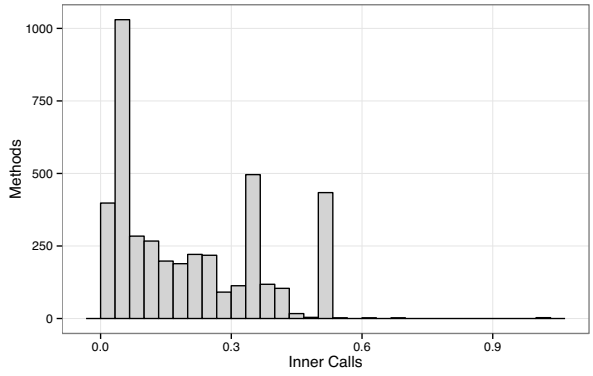

Fig. 12. Ratio of inner calls in constraint sequences.

Figure 13 shows the distribution of inter calls. 8,597 (25%) of all analyzed methods have inter calls. Considering these methods, inter calls usually represent less than 40% of their constraint elements.

Figure 14 shows the distribution of external dependencies. 41% (14,126) of the analyzed methods have constraints with external dependency, while 59% (20,367) do not. Regarding these methods, external calls generally represent up to 50% of their constraint elements.

D. RQ_4: What is the distribution of exception-depended paths?

There are 12,545 methods with exceptions-depended paths, i.e., paths that are executed only when specific exceptions are thrown. It represents 36% of the analyzed methods. Figure 15 shows the distribution of exception-depended paths. To improve the understanding of the histogram, we removed methods that do not have exception-depended paths (it represents about 64% of the analyzed methods). Notice that, out of the 12,545 methods with exception-depended paths, about 8,000 methods have exclusively exception declarations elements. This means that 23% of all analyzed methods have `try-catch-finally` instructions but with no conditional statements (e.g., `if`, `switch`, and `while`).

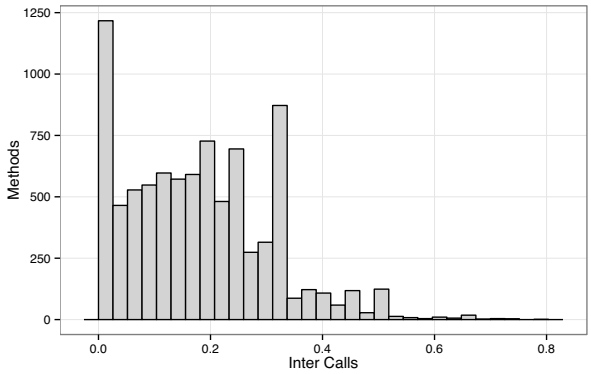

Fig. 13. Ratio of inter calls in constraint sequences.

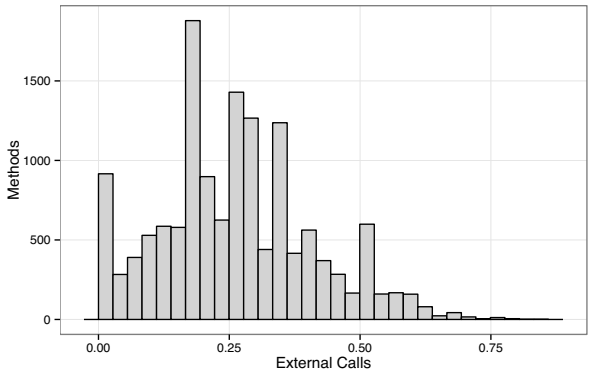

Fig. 14. Ratio of external calls in constraint sequences.

It is important to highlight that symbolic execution considering exception-depended paths has not been further explored before, then we based our study in some assumptions. Suppose that node *C* represents instructions to handle an exception *E* that can be thrown on node *N*. If node *N* is executed without an exception thrown, it goes to node *R* in the CFG. The constraint required to go from node *N* to *C* is an implicit `if (E is thrown)` statement. In this study, we did not consider that there is a constraint `if (E is not thrown)` required to go from node *N* to *R*, since this case does not impact on exception-depended paths.

VI. Discussion

In this section, we provide some discussion related to path explosion, constraint complexity, dependency, and exception-depended paths based on the analysis we performed for the SF100 projects.

Path explosion: from analyzing Tables II and III, we can see that methods with loops are not very common, and methods containing more than one loop are even less common. So, as far as $\mathbf{RQ}_1$ is concerned, we can argue that loops and inner loops do not add much to the complexity of symbolic execution, at least regarding the SF100 benchmark analysis.

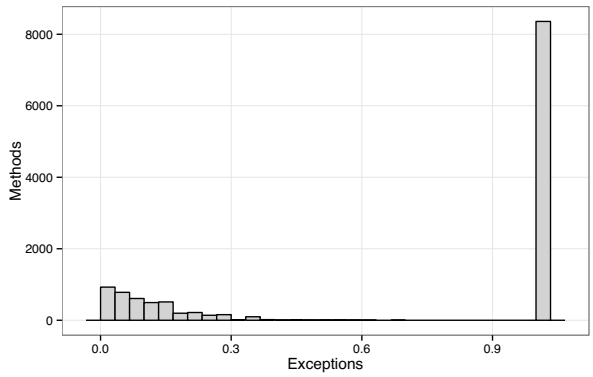

Fig. 15. Ratio of exception-depended paths.

Methods that have at least one inner loop represent only 6.4% of all analyzed methods (34,493) and 1.6% of all methods (135,156) of SF100. These according to these results, only 25% of the analyzed methods have potential for path explosion, but, in fact, only 6.4% have at least one inner loop and have higher potential to cause path explosion. As mentioned, the number of methods with more than one inner loop is even smaller: 2.25% of all analyzed methods.

Path explosion is one of the key challenges of symbolic execution. It can dominate runtime and hamper the test data generation process. For instance, we found methods with up to 35 inner loops. During test data generation using symbolic execution, this method would hog system resources, jeopardizing the entire process. However, we can conclude that generally symbolic execution lends itself to generating test data (from the path explosion point of view). An interesting strategy is to generate test data for 93.6% of the analyzed methods using symbolic execution, and employ different techniques for methods that may lead to path explosion.

It is worth emphasizing that our findings are based on a symbolic execution process that does not consider the integration among methods of the same class or project. This can potentially hide inner loops that arise from these integrations.

Constraint Complexity: in the light of the results presented in Section V-B, the answer to $\mathbf{RQ}_2$ is that integers and objects are the two data types that have the most influence on the complexity of constraint sequences, while the other data types are not too frequent.

Due to advances in the constraint solving technology, constraint solvers can easily handle most issues related to fixed-point numbers. Nevertheless, there are only around 2,300 methods whose constraint sequences are 100% composed of integer type elements. This information provides evidence that symbolic execution approaches that deal only with integers can automatically generate test data only for 6.6% of the analyzed methods.

According to our analysis, floating-point data types are not a key issue to symbolic execution approaches since the number of methods whose constraints have floating-point elements is too low. This may not be hold for domains in which floating-point numbers are predominant.

The low number of arrays may be explained by the widespread use of Java collections, which are considered object types in this investigation. Array types should not limit symbolic execution approaches since most of them have handled this type at both symbolic execution and constraint solver level. Apart from those reasons, there are few methods using arrays at least as constraint elements.

Object types still pose several challenges to constraint solvers and constraint solving is still one of the major bottlenecks in symbolic execution. The predominance of objects over the constraint sequences of the SF100 benchmark shows that researchers should focus on optimizing constraint solvers to deal with complex types (e.g., objects), since handling objects will increase the applicability of symbolic execution approaches.

Dependency: as far as $\mathbf{RQ}_3$ is concerned, results presented in Section V-C show that 73% (14,126) out of the 19,273 methods with dependency are external calls. Therefore, techniques to integrate the caller with the called methods to eliminate dependency may work only for 27% of the methods with some dependency. This suggests that approaches to handle calls to other libraries are strongly recommended if intended to be used in real-world software projects. External calls may not represent a bigger issue when the external library or project is available for integration at bytecode level, for example [10].

Exception-Depended Paths: the answer to $\mathbf{RQ}_4$ (Section V-D) indicates that exception-depended paths are an important issue to symbolic execution since one third of the methods has to deal with exceptions. 25% of the analyzed methods presented only exception elements. We claim that approaches that want to generate test data to real-world software using symbolic execution should be able to handle exception-related constraints. Unfortunately, there is a lack of approaches that can cope with this issue.

Comparison with related work: the results of this investigation are similar to the results obtained by Qu and Robinson [10]. As for dependency, they found that 20% of the analyzed methods have native calls, but they did not explore inner, inter, and other types of external calls. Concerning constraint complexity, they found that floating-point numbers are concentrated in a small number of methods and objects are far more predominant.

VII. Concluding Remarks and Future Directions

In this paper, we described a study with real-world projects aimed at quantifying the factors that affect symbolic execution for testing. We used a corpus of 100 Java projects

from which several metrics were extracted. Several results have been obtained for known issues, namely, path explosion, complexity of constraint sequences, dependency of method calls, and exception-depended paths. These results provide valuable insight into how current approaches can be evaluated and new techniques can be tailored to better suit the needs of different real-world applications. The provided analysis is also extremely useful in driving future efforts to automate test data generation using not only symbolic execution, but other techniques (e.g. concolic testing and search-based strategies).

Nevertheless, further investigation is required to obtain more knowledge regarding the limitations of symbolic execution applied to test data generation. As future work, we intend to analyze other 100 different Java projects to compare the results with the SF100 analysis. Also, we intend to extend our tool to perform the integration of inner and inter calls in many degrees. Integration between methods may affect constraint size and loop or inner loop counting. Moreover, we intend to perform the same analysis described in this paper, but by grouping projects according to their domain.

ACKNOWLEDGMENTS

The authors would like to thank the financial support provided by CAPES (grant number BEX 1714/14-7), FAPESP (process 2014/07969-0), and CNPq.

REFERENCES

[1] G. J. Myers, C. Sandler, T. Badgett, and T. M. Thomas, *The Art of Software Testing*. John Wiley & Sons, Inc., Hoboken, New Jersey, 2004.

[2] S. Galler and B. Aichernig, "Survey on Test Data Generation Tools," *International Journal on Software Tools for Technology Transfer*, pp. 1–25, 2013.

[3] C. Ramamoorthy, S.-B. F. Ho, and W. Chen, "On the automated generation of program test data," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 293–300, 1976.

[4] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[5] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[6] 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*, vol. 86, no. 8, pp. 1978–2001, 2013.

[7] C. S. Pasareanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *International Journal of Software Tools and Technology Transfer*, vol. 11, no. 4, pp. 339–353, 2009.

[8] 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*. ACM, 2011, pp. 1066–1071.

[9] G. Fraser and A. Arcuri, "Sound Empirical Evidence in Software Testing," in *Proceedings of the 2012 International Conference on Software Engineering*, 2012, pp. 178–188.

[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*, 2011, pp. 117–126.

[11] K. Sen, "Concolic testing," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, New York, NY, USA, 2007, pp. 571–572.

[12] P. Godefroid, "Test Generation Using Symbolic Execution," in *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, ser. Leibniz International Proceedings in Informatics, vol. 18, 2012, pp. 24–33.

[13] S. Khurshid, C. S. Păsăreanu, and W. Visser, "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*, 2003, pp. 553–568.

[14] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test input generation with java pathfinder," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2004, pp. 97–107.

[15] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," *SIGPLAN Notes*, vol. 40, no. 6, pp. 213–223, 2005.

[16] B. Elkarablieh, P. Godefroid, and M. Y. Levin, "Precise pointer reasoning for dynamic test generation," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, 2009, pp. 129–140.

[17] P. Godefroid and J. Kinder, "Proving memory safety of floating-point computations by combining static and dynamic program analysis," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, 2010, pp. 1–12.

[18] SourceForge, "Corina: The cornell tree-ring analysis system," [February 2014]. [Online]. Available: http://sourceforge.net/projects/corina/

[19] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Transactions on Software Engineering*, vol. 11, no. 4, pp. 367–375, 1985.

[20] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software Unit Test Coverage and Adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366–427, 1997.