Generating Test Cases for Programs that Are Coded Against Interfaces and Annotations

MAINUL ISLAM and CHRISTOPH CSALLNER, University of Texas at Arlington

Automatic test case generation for software programs is very powerful but suffers from a key limitation. That is, most current test case generation techniques fail to cover testee code when covering that code requires additional pieces of code that are not yet part of the program under test. To address some of these cases, the Pex state-of-the-art test case generator can generate basic mock code. However, current test case generators cannot handle cases in which the code under test uses multiple interfaces, annotations, or reflection.

To cover such code in an object-oriented setting, we describe a novel technique for generating test cases and mock classes. The technique consists of collecting constraints on interfaces, annotations, and reflection, combining them with program constraints collected during dynamic symbolic execution, encoding them in a constraint system, solving them with an off-the-shelf constraint solver, and mapping constraint solutions to test cases and custom mock classes. We demonstrate the value of this technique on open source applications. Our approach covered such third-party code with generated mock classes, while competing approaches failed to cover the code and sometimes produced unintended side-effects such as filling the screen with dialog boxes and writing into the file system.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution*; *Testing tools*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*

General Terms: Algorithms, Reliability, Verification

Additional Key Words and Phrases: Dynamic symbolic execution, mock classes, stubs, test case generation

1. INTRODUCTION

Unit testing is an important technique for finding software bugs [Mockus et al. 2002; Do et al. 2005; Venolia et al. 2005; Williams et al. 2009; Williams et al. 2011; Zaidman et al. 2011; Mäntylä et al. 2012]. For example, according to a survey among software developers at Microsoft in 2005, the fraction of software developers who use unit tests was 79%. When a project uses unit tests, it often uses a lot of them. For example, it is reported that in many projects within Microsoft the production code is smaller than its unit tests [Tillmann and Schulte 2006b].

A unit test (or test case) consists of three steps. (1) First the test case prepares the test by creating input values that are suitable for the code unit under test. (2) Then the test case invokes the code under test, passing the input values determined in step one. These input values force testee execution along a particular execution path. (3) Finally the test case observes the execution results and passes them to a test oracle, which determines if the test exposed a bug. The first step of finding suitable input values is

This is a revised and extended version of a WODA workshop paper [Islam and Csallner 2010]. This material is based upon work supported by the National Science Foundation under Grants No. 1017305 and 1117369.
Authors' addresses: M. Islam, Computer Science and Engineering Department, University of Texas at Arlington, mainul.islam@mavs.uta.edu; C. Csallner, Computer Science and Engineering Department, University of Texas at Arlington, csallner@uta.edu.

DOI:http://dx.doi.org/10.1145/0000000.0000000

crucial, as the input values determine the execution behavior that can be observed and verified in the subsequent steps. That is, if we fail to find input values that can reach a certain part of the testee, we cannot test that part of the code.

Today many unit test cases are written manually, which makes testing expensive in terms of money and developer time. Given the importance of finding suitable input values and the cost of manually writing them down in test cases, it is not surprising that many tools and techniques have been developed for generating suitable input values automatically. Recent work includes random test case generators such as JCrasher [Csallner and Smaragdakis 2004] and Randoop by Pacheco et al. [2007], bounded exhaustive test case generators such as Korat by Boyapati et al. [2002], staged static and dynamic program analyses such as Check 'n' Crash [Csallner and Smaragdakis 2005], and systems that intertwine dynamic and symbolic analysis very tightly such as EXE by Cadar et al. [2006], KLEE by Cadar et al. [2008], and dynamic symbolic ("concolic") execution systems such as DART by Godefroid et al. [2005], SMART by Godefroid [2007], SAGE by Godefroid et al. [2008], and Pex by Tillmann and de Halleux [2008].

A key to the success of these test case generation tools is that, at their core, these tools are dynamic program analyses. That is, these tools are aware of the analyzed code's full execution semantics and only produce results based on actual execution paths through the code under test. This is in contrast with most static program analysis techniques, which approximate the analyzed code's execution behavior. Despite much progress in static analysis research, leading to tools such as ESC/Java by Flanagan et al. [2002], Spec# by Barnett et al. [2004], and Snugglebug by Chandra et al. [2009], static analysis tools are still prone to false warnings and spurious results, especially when dealing with programs that use native code, reflection, or aliasing [Hind 2001; Bessey et al. 2010; Brauer and Simon 2012].

1.1. Limitation of Current Dynamic Test Case Generators

While current (dynamic) test case generators such as Pex are very powerful, they have several problems. Ultimately these problems limit the code coverage achieved by the generated test cases. (By code coverage we mean standard structural code coverage such as branch coverage and statement coverage). While these problems are often cited in the literature to motivate the development of new tools and techniques, we are not aware of an exhaustive study of these problems.

On the positive side, in recent years first empirical studies have appeared that shed some light on the problems of automatic test case generation techniques. Most closely related is the study by Xiao et al. [2011], which carefully analyzed why the Pex dynamic symbolic execution engine failed to achive high coverage of 10 files of four .Net programs. In these files, the most common problem was the object-creation problem, i.e., Pex could not determine which methods to call in which order and with which parameters in order to bring objects needed for testee execution into a state needed for forcing execution along a certain path. In order of occurrence, the remaining problems found were the inability to analyze external code such as native code, exploring all paths through loops and recursion, and certain constraints not being supported by the underlying constraint solver.

Similar studies, by Anand et al. [2007] and Jaygarl et al. [2010], on related but different test case generation techniques had similar themes. That is, Anand et al. [2007] studied the problems encountered during symbolic execution with Java PathFinder. The two encountered kinds of problems were the tool's inability to reason about external code (i.e., native code) and certain constraints not being fully supported by the underlying constraint solver. Finally, Jaygarl et al. [2010] studied problems in random test case generation encountered by the Randoop tool. The main problem of Randoop,

similar to the Pex study by Xiao et al. [2011], was the inability of the tool to create the right kind of objects.

In this article we examine a problem that is related but orthogonal to the object-creation problem. That is, current test case generators do not work in several cases in which the testee can be compiled but not executed and therefore not tested. The reason for this problem is that the testee invokes some interfaces and there exists no code yet that implements these interfaces (see Section 1.2 for an example). Since no implementation class exists yet, current test case generators do not know how to create objects that implement these interfaces.

This limitation prevents current test case generators from advancing further on a grand goal of testing and software engineering, which is finding bugs earlier in the software development process [McConnell 2004, page 29]. People want to find bugs sooner because the longer a bug lingers in the code, the more costly it is to fix that bug. These increased costs have been explained and documented, for example in the 2002 NIST study on the economic impacts of shortcomings of current testing practices [National Institute of Standards and Technology (NIST) 2002, Section 4.1.2]:

> "An important reason why it is more costly to correct bugs the longer they are left undetected is because additional code is written around the code containing the bug. The task of unraveling mounting layers of code becomes increasingly costly the further downstream the error is detected."

When code required for testing is missing, a test case generator may generate code that can stand in for the missing code. Such stand-in code is often called a stub or a mock [Beizer 1990, page 115] [McConnell 2004, page 523f] [Pezzè and Young 2007, page 229f]. In the literature special cases of such code are sometimes also referred to as dummy, stand-in, fake, and double [Meszaros 2007, page 743]. For simplicity, we will refer to any code that is used to stand in for a current or future piece of code for testing as a *mock*. [1]

To contrast this problem with the object-creation problem, the object-creation problem deals with creating objects from existing classes, by calling existing methods. On the other hand, this article focuses on how to create objects from classes that do not exist yet.

While in recent years we have seen many practical mock frameworks for object-oriented programs from the open-source community (for example, jMock [Freeman et al. 2004a; 2004b; Freeman and Pryce 2006]), such frameworks have different goals and capabilities. At a high level, these mock frameworks aim at supporting manually written test cases. While these mock frameworks generate mock classes according to a user-provided specification, these frameworks do not attempt to explore and cover the testee automatically and do not generate test cases. In contrast, we aim for an automatic test case generator that can generate test cases with high code coverage, even if high code coverage requires generating mock classes.

In a modern object-oriented programming language such as Java or C#, generating mock classes is hard. Programs written in such languages make frequent use of interfaces and therefore multiple inheritance, dynamic type checks, annotations, native code, and reflection. For example, according to an empirical study of some 100,000 Java classes and interfaces, there is an **instanceof** dynamic type check expression in about every other class [Collberg et al. 2007].

---

[1] Another potential source of confusion is that some related work on testing object-oriented programs refers to mock classes as mock objects [Marri et al. 2009, Footnote 1]. Since in the terminology of object-oriented programming each such mock object is really a class, we prefer the term mock class and use it throughout this article.

For a test case to reach a certain part of the testee, we may therefore have to generate mock classes that implement certain interfaces, have certain annotations, satisfy complex type checks stemming from dynamic type checks and reflection, and satisfy constraints placed on the values returned from generated mock class methods. In our approach, we collect such complex constraints during dynamic symbolic execution [Godefroid et al. 2005; Godefroid 2007; Tillmann and de Halleux 2008; Elkarablieh et al. 2009] of the testee, encode them in the input language of an off-the-shelf SMT-solver [Moura and Bjørner 2008], interpret constraint solver models as test cases and mock classes, and generate concrete corresponding test cases and mock classes.

1.2. Motivating Example

As an example, in an object-oriented program the programmer may have just coded Listing 1, which is a first version of class C. In this freshly coded class the foo method casts its parameter p of static type interface I to a different interface J, calls on the parameter the bar method, and uses the result of the method call in its control-flow decisions. Now the programmer may want to start testing the foo method—before coding classes that implement I and J.

```
public interface I { public int doX(); /* .. */ }                    1
public interface J { public int bar(); /* .. */ }                    2
public class C {                                                     3
  public static void foo(I p, int x, Class c)                        4
  {                                                                  5
    if (p instanceof J)                                              6
    {                                                                7
      J j = (J) p;                                                   8
      if (j.bar() > x)                                               9
        int val = p.doX();                                           10
      // ..                                                          11
    }                                                                12
  }                                                                  13
}                                                                    14
```

Listing 1: First version of example class C under test.

A key observation about this example is that the control flow of the foo method depends on the outcome of at least one dynamic type check. That is, even if the programmer would have omitted the (p **instanceof** J) dynamic type check in line 6, the subsequent type cast in line 8 would have triggered the virtual machine to perform an (implicit) dynamic check of the type of parameter p. The virtual machine throws a runtime exception if the parameter cannot be cast to the given type. If thrown, such a runtime exception would divert the control flow of the foo method to a code block dedicated for catching and handling the exception. If there is no such code block, then the control flow terminates and the application crashes.

To avoid such runtime exceptions (and subsequent program crashes), the programmer used line 6 to guard the subsequent type cast with an explicit dynamic type check. In our example, the programmer guarded the cast to J with **if** (p **instanceof** J). A similar explicit type check can be achieved via reflection as in **if** (c.isAssignableFrom(p.**class**)). The latter checks if the dynamic type of p (the type represented by the Java class p.**class**) is a subtype of the type represented by the class variable c, which may refer to types other than J.

To cover such code, we extend automatic test case generation with reasoning about explicit and implicit dynamic type checks and solving them together with other con-

straints imposed by the program, such as constraints stemming from how values returned by generated code is used in subsequent control-flow decisions.

1.3. Research Questions and Contributions

In this article we extend automatic test case generation of Java-like programs to scenarios in which code that is required for testing is not yet implemented (or otherwise unavailable for execution). That is, the testee performs dynamic type checks and may call interface methods, but there is no class yet that implements these interfaces. Before developing our approach, we first confirm that such scenarios indeed occur in third-party applications. Specifically, we investigate the following five research questions (RQ).

RQ1: Do third-party object-oriented applications use complex dynamic type checks, for example, to check if a method parameter is an instance of two non-compatible supertypes (RQ2) or if a method parameter has certain annotations (RQ3)?
RQ2: Can automatically generated test cases cover code that (in addition to performing other computing tasks) type-checks references dynamically, even if this code cannot be covered with any of the existing classes?
RQ3: Can automatically generated test cases cover code that is coded against annotations, even if none of the existing classes has the required annotations?
RQ4: How does a test case generator that systematically covers the patterns of RQ2 and RQ3 compare with a random test case generator?
RQ5: How does a test case generator that covers the patterns of RQ2 and RQ3 compare with a state-of-the-art systematic test case generator such as Pex and Moles?

To summarize, this article makes the following contributions to automatic test case generation.

— We survey dozens of open-source applications with a total size of more than 2 MLOC and find hundreds of cases of RQ2 and RQ3 patterns, which answers RQ1 with yes.
— To answer RQ2 and RQ3 we design a novel test case generation technique based on dynamic symbolic execution. Our technique systematically covers code that is coded against interfaces and annotations. One part of the technique generates custom mock classes, which we also call dynamic symbolic mock classes.
— We describe an implementation of our test case generation technique for Java and distribute our implementation as an open source tool.
— We report on our experience of applying our technique on open source applications. To provide initial answers to RQ4 and RQ5 we compare our implementation to state-of-the-art test case generators, the dynamic symbolic execution system Pex and Moles and the random test case generator Randoop. We find that our approach can cover many pieces of third-party code that existing approaches cannot cover.

We describe our examples, design, implementation, and evaluation in terms of Java programs, but the ideas and techniques could also be adapted and applied to related object-oriented languages such as C++ and C#.

2. BACKGROUND

In this section we provide necessary background on our research questions, i.e., on Java reference types, their sub- and supertype relations, annotations, dynamic type checks, and reflection. We also provide background information on dynamic symbolic execution and our Dsc dynamic symbolic execution engine.

2.1. Sub-/Supertype Relation on Reference Types in Java

Java programs have two kinds of types, primitive types (boolean, int, float, etc.) and reference types. Reference types are arrays, interfaces, and classes. Like many object-oriented languages, Java defines on the reference types a binary subtype relation. For example, if a Husky type declares that it implements or extends the Dog type, then Dog is a direct supertype of Husky and Husky is a direct subtype of Dog. Reflexive and transitive closure of these direct relations yields the super- and subtype relations.

In Java, each class has one direct class supertype and arbitrarily many direct interface supertypes. The Object class is special, it has no direct supertype. Another special type is the null type; it has no direct subtype but is a subtype of every other reference type. An interface only has interface supertypes, except if it does not explicitly declare any supertype, then it has an implicit one—the Object class.

Java distinguishes between abstract and non-abstract reference types. Each interface is abstract and a class may declare to be abstract. A non-abstract type may declare to be final, in which case it has one subtype, the null type. Each reference value is either the null value or a pointer to an object. Each object is an array or an instance of a non-abstract class [Gosling et al. 2005].

2.2. Annotations in Java

A Java programmer can annotate various program elements with custom annotations, e.g., the public class C with annotation @A as follows: @A **public class** C {..}. Besides classes, a programmer can currently also annotate packages, interfaces, enums, annotations themselves, fields, constructors, methods, parameters, and local variables. In addition to that, Java specification request JSR 308 proposes to allow programmers to annotate several additional program elements[2].

In Java, annotations are implemented via interfaces. That is, each annotation definition such as **public** @interface A {..} is interpreted as the definition of reference type A that is essentially a special interface. Each annotation implicitly is a subtype of the interface java.lang.annotation.Annotation and cannot declare other super-types. However, an annotation can be the super-type of user-defined classes and interfaces.

Several pre-defined annotations have special semantics. That is, @Target is used in annotation definitions, for specifying the program elements the annotation can be applied to. @Retention specifies how long an annotation should be retained, i.e., until compile-time, in the class's bytecode, or until runtime. Retaining annotations until runtime allows accessing them via reflection.

2.3. Dynamic Type Checks and Reflection

Several Java program elements give rise to multiple-inheritance type constraints and constraints on annotations, including the following. The first two are the Java language dynamic type check and cast expressions. The remaining ones are reflection methods, defined by the java.lang.Class and java.lang.reflect.AnnotatedElement classes.

— e **instanceof** T
— (T) e
— **boolean** Class.isAssignableFrom(Class)
— **boolean** Class.isInstance(Object)
— **boolean** Class.isAnnotation()
— **boolean** AnnotatedElement.isAnnotationPresent(Class)
— Annotation AnnotatedElement.getAnnotation(Class)
— Annotation[] AnnotatedElement.getAnnotations()

[2]JSR 308 is available at http://jcp.org/en/jsr/detail?id=308

2.4. Dynamic Symbolic Execution and Dsc

Dynamic symbolic execution (DSE), first described by Godefroid et al. [2005], is a relatively new kind of program analysis. As its name suggests, dynamic symbolic execution executes a program both dynamically (concretely, on a standard interpreter or runtime system such as a standard Java virtual machine) and symbolically (on a symbolic interpreter). The key aspect is that these two executions are conducted side-by-side. That is, after a program instruction is executed concretely, the same instruction is evaluated symbolically. Thus the concrete execution guides the symbolic execution.

As in traditional symbolic execution [King 1976; Clarke 1976], the symbolic interpreter in DSE treats each program parameter as a symbolic variable and executes the program over those symbolic variables. All intermediate values are then represented as symbolic expressions over the symbolic input variables. For example, when analyzing a method that starts with **void** foo(**int** p) { **int** x = p + 1; **if** (x>0), first the standard interpreter of DSE takes the concrete input value passed to foo such as 5, executes the first statement **int** x = p + 1 and assigns the resulting value 6 to x. Then the symbolic interpreter of DSE assigns to parameter p the symbolic variable P and to x the resulting symbolic expression $P + 1$. This allows the symbolic interpreter to record later the symbolic conditions for which the standard interpreter took an execution branch. For example, the subsequent **if** (x>0) leads to a symbolical constraint of $P + 1 > 0$, which describes a constraint on the input variables that any future execution has to satisfy in order to execute the same execution path.

After executing a program path, the branching decisions collected by the symbolic interpreter are used to craft another concrete program input, by encoding a new path as a symbolic constraint system over the program parameters, solving the constraint system, and mapping the constraint solution to a new concrete program input value. Thereby the symbolic execution guides DSE to the next execution path.

Dsc is a dynamic symbolic execution engine for Java bytecode, in which we have implemented our approach for generating mock-classes[3]. Dsc uses the high-performance automated theorem prover Z3 from Microsoft Research [Moura and Bjørner 2008], to solve constraints generated during symbolic execution.

3. SURVEY OF OPEN-SOURCE SUBJECT APPLICATIONS (RQ1)

Research question 1 asks if applications actually use code that fits the patterns of RQ2 and RQ3. In this section we survey third-party applications, focusing on two example scenarios. In the first scenario, executing the code under test depends on having a class that satisfies multiple-inheritance constraints, which is a special case of RQ2. In the second scenario, executing the code under test depends on having a class that has certain annotations, which reflects RQ3. We focus on these two scenarios as they are examples of cases that state-of-the-art test case generation techniques such as Pex and Moles do not cover. In the surveyed applications these two example scenarios occur hundreds of times, which answers RQ1 with yes.

For this survey we selected from a wide range of application areas as subjects some 35 small to medium-sized Java applications. Many of our subjects are well-known open source representatives of their application area, including Apache ActiveMQ (message brokers), Apache Ant (build tools), Apache Derby (database engines), ArgoUML (UML editors), JUnit (test automation), Tomcat (web servers), and Xalan (XML processors). Table I lists the size of each subject in non-commenting source statements (LOC), the number of classes, interfaces, and methods, and the average cyclomatic complexity of each method (CC). LOC ranges from 2 to 338 kLOC with a total of over 2 MLOC.

[3]Dsc is available at http://ranger.uta.edu/~csallner/dsc/index.html

3.1. Multiple-Inheritance Patterns of Table I

We searched the subjects of Table I for several variants of *multiple-inheritance from incompatible super-types*. Specifically, patterns p1, p2, and p3 are branch conditions that use an **instanceof** dynamic type check to compare two incompatible types. Such a branch can only be covered with an instance of a class that satisfies multiple-inheritance constraints.

Table I lists some three hundred occurrences of these patterns. However, p1, p2, and p3 are just three special cases to show how different patterns can impose multiple inheritances and our subjects likely contain many more branches that existing techniques also cannot cover. For example, **instanceof** compares an arbitrary instance with a fixed type. Table I also lists hundreds of calls to the more flexible type checks in the isAssignableFrom(Class) and isInstance(Object) reflection methods, which can be invoked on arbitrary class instances and arbitrary types (RQ2).

The numbers in Table I should not be confused with numbers of test cases or numbers of bugs. That is, we did not study how many execution paths pass through each pattern occurrence. For example, a pattern 1 instance listed in Table I typically acts as a guard that controls the execution of a chunk of code. In order to fully cover the thereby guarded statements, a tester or test case generator may need to carefully craft dozens of distinct test cases. The key observation is that in order to cover any portion of such guarded application code, we need to either manually write custom mock classes or generate them.

When ordering the patterns, we placed p1 first, because we expect that it may be easier for a test case generator to generate test cases that cover p1 than to cover p2 or p3. That is, pattern p1 places type constraints on an object obj that could be set easily by a test case generator—the object obj is a parameter of the method under test. Patterns p2 and p3 are more involved, as they require a test case generator to perform additional solution steps. Specifically, pattern p2 requires such a solution object obj to be a field of another object. Setting such a field can be very complicated and is an instance of the well-known object-creation problem [Xiao et al. 2011]. Pattern p3, on the other hand, requires such a solution object obj to be returned by a method, which may require reasoning about an existing method or generating a new method that overrides an existing method to return the new object.

As a proof-of-concept, our implementation and evaluation focuses on pattern p1. Integrating our solution with solutions for other challenges such as the object-creation problem is future work. Looking at the numbers in Table I, it is clear that pattern p1, despite being potentially the simplest among the patterns, is still relatively common. That is, in the subjects examined, p1 was more common than the other two patterns.

3.1.1. Pattern p1: Method Parameter Check. Pattern 1, shown in Listing 2, matches branching statements that branch on an instanceof dynamic type check of a method parameter against a non-compatible type. Such code is frequently used to test if a method parameter instance is also a subtype of a type that is unrelated to the parameter's formal type. To cover code that matches this pattern, a test case generator must generate a mock class that is a subtype of both the formal parameter type and the type used in the dynamic typecheck.

```
M m(.., T t, ..) { // ..
  if (.. (t instanceof X))
  // ..
}
```

Listing 2: Pattern 1 matches branching statements that branch on an instanceof dynamic type check of a method parameter t against type X that is not compatible with the declared type T of t.

Table I: Multiple-inheritance patterns in Java applications; LOC = non-commenting source statements; Class = #classes and enums, Intf = #interfaces and annotations, M = #methods (each including public, protected, package-private, and private, abstract and non-abstract, top-level and nested); CC = average cyclomatic complexity per method; instof = #instanceof expressions; patterns p1, p2, p3 are not covered by other tools such as Pex and Moles. Calls to reflection methods iAF = #Class.isAssignableFrom(Class) and iI = #Class.IsInstance(Object) likely contain more such multiple-inheritance cases that no existing tool can cover (RQ2). ActiveMQ, Ant, Derby and Tomcat are Apache projects; JSR 308 = JSR 308 Checkers Framework; AndroMDA Core is v3.3, Unimode Core is v1.3.39.1; LOC and CC counted with JavaNCSS.

Subject	kLOC	Class	Intf	kM	CC	instof	p1r	p1	p2r	p2	p3r	p3	iAF	iI
ActiveMQ 5.5.0	156.6	2,047	316	20.5	1.8	389	11	11	0	0	0	0	8	8
AndroMDA Core	5.6	119	10	0.9	2.3	14	0	0	0	0	0	0	7	1
Ant 1.8.2	74.0	1,013	77	9.7	2.8	261	1	1	5	5	0	0	36	3
ArgoUML app 0.32	81.9	1,810	111	11.2	2.5	1,069	1	0	5	5	5	5	2	1
ASM 4.0 RC1	17.2	151	14	1.5	4.2	193	3	3	0	0	0	0	4	0
BCEL 5.2	17.2	348	35	2.9	2.2	337	4	3	0	0	0	0	0	6
Cobertura 1.9.4.1	44.0	112	10	3.4	6.9	7	0	0	0	0	0	0	0	0
Columba 1.2	16.4	1,165	130	7.0	1.9	188	0	0	3	1	0	0	1	1
Derby 10.8.1.2	149.6	1,240	265	18.6	3.0	1,013	1	1	41	37	6	4	10	25
DrJava r5425	75.0	4,908	292	40.7	2.0	1,744	8	5	3	1	2	2	11	11
Drools 5.1.1	61.0	1,289	229	11.2	1.9	518	7	7	7	7	0	0	20	7
EasyMock 3.0	2.9	74	9	0.6	2.0	50	0	0	0	0	0	0	5	1
FindBugs 1.3.9	74.8	1,564	171	11.2	3.4	697	0	0	0	0	0	0	14	0
Groovy 1.8.0	77.9	3,048	141	36.3	3.2	1,812	9	5	8	6	0	0	66	15
Guice Core 3.0	9.9	464	101	3.3	1.8	208	3	3	1	1	0	0	16	2
Hibernate Core 3.6.0	90.8	2,183	495	20.3	2.1	496	5	2	4	3	1	1	257	223
Jasmin 2.4.0	2.9	150	2	0.7	6.4	138	0	0	0	0	0	0	0	0
Javassist 3.11.0	25.0	330	17	3.3	3.0	135	0	0	0	0	0	0	0	1
JaxLib 0.6.4	68.8	536	68	11.0	3.0	794	137	35	29	20	0	0	8	3
Jdec 2.0	83.6	844	6	5.9	5.4	96	0	0	0	0	0	0	0	0
Jedit 4.3	40.8	892	44	6.7	3.3	518	0	0	2	2	0	0	4	2
JFreeChart 1.0.13	64.7	508	103	8.0	2.8	640	16	16	0	0	1	1	2	0
Jgap 3.5	18.9	349	92	3.1	2.4	30	1	1	2	2	0	0	28	3
JMonkeyEngine 3.0	338.4	859	103	8.5	2.5	346	2	2	0	0	0	0	8	1
JSR 308	13.6	999	321	9.5	3.6	448	2	2	1	1	4	3	106	5
JUnit 4.9b2	4.5	203	45	1.2	1.7	30	2	2	0	0	2	2	11	6
Jython 2.5.2	158.4	5,481	366	51.6	3.7	4,136	24	13	10	7	0	0	229	130
Polyglot 1.3.5	42.1	362	140	4.1	2.7	431	3	3	12	3	1	1	5	1
Soot 2.4.0	111.1	2,698	272	31.4	3.0	5,136	28	27	4	4	52	48	0	0
Spring Core 3.0.5	10.7	229	67	1.9	2.7	197	0	0	0	0	0	0	27	10
Tomcat 7.0.21	119.6	1,378	112	15.8	3.1	815	0	0	0	0	0	0	39	9
UmlGraph 5.4	2.2	30	2	0.3	2.9	10	0	0	0	0	0	0	0	0
Unimode Core	8.6	172	40	1.3	2.4	63	0	0	0	0	0	0	4	0
Weld Core 1.1.0	16.9	492	34	3.3	2.1	212	0	0	0	0	0	0	21	0
Xalan 2.7.1	93.9	703	36	6.4	3.1	342	7	5	24	7	0	0	15	0
Total	2,179.4	38.8k	4.3k	373.0	3.0	23.5k	275	147	161	112	74	67	964	475

Table II: Calls to annotation access reflection methods (RQ3), grouped by reflection object (class, method, field). Existing test case generators do not reason about annotations; iAP = isAnnotationPresent(Class), iA = isAnnotation(), gAs = getAnnotations(), gA = getAnnotation(Class), gDA = getDeclaredAnnotations().

Subject	Class					Method				Field				
	iAP	iA	gAs	gA	gDA	iAP	gAs	gA	gDA	iAP	gAs	gA	gDA	Total
ActiveMQ 5.5.0	0	0	0	0	0	0	0	2	0	0	0	0	0	2
AndroMDA Core	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Ant 1.8.2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ArgoUML app 0.32	3	0	0	0	0	0	0	0	0	0	0	0	0	3
ASM 4.0 RC1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
BCEL 5.2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Cobertura 1.9.4.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Columba 1.2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Derby 10.8.1.2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DrJava r5425	0	2	4	3	0	0	2	6	0	0	1	3	0	21
Drools 5.1.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
EasyMock 3.0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
FindBugs 1.3.9	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Groovy 1.8.0	0	0	2	6	0	0	1	4	0	0	0	0	0	13
Guice Core 3.0	0	0	2	4	0	2	2	0	0	0	1	0	0	11
Hibernate Core 3.6.0	0	0	0	4	0	1	0	3	0	0	0	2	0	10
Jasmin 2.4.0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Javassist 3.11.0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
JaxLib 0.6.4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Jdec 2.0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Jedit 4.3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
JFreeChart 1.0.13	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Jgap 3.5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
JMonkeyEngine 3.0	2	0	0	2	0	0	0	0	0	0	0	1	0	5
JSR 308	2	2	0	14	0	0	0	0	0	0	0	0	0	18
JUnit 4.9b2	0	0	4	5	0	0	2	6	0	0	1	3	0	21
Jython 2.5.2	0	0	0	4	0	1	1	4	0	0	0	0	0	10
Polyglot 1.3.5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Soot 2.4.0	0	0	0	2	0	0	0	0	0	0	0	0	0	2
Spring Core 3.0.5	2	0	19	4	1	0	10	2	0	0	2	0	0	40
Tomcat 7.0.21	4	2	0	7	0	7	0	5	0	5	0	5	0	35
UmlGraph 5.4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Unimode Core	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Weld Core 1.1.0	6	2	8	3	3	3	2	0	1	1	1	1	1	32
Xalan 2.7.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Total	19	8	39	58	4	14	20	32	1	6	6	15	1	223

ACM Transactions on Software Engineering and Methodology, Vol. V, No. N, Article A, Pub. date: January YYYY.

Following are the pattern 1 details. Unless noted otherwise, each pattern matches as broadly as possible. For example, if there is a method definition "M m(..)" as in pattern 1 we mean that M can be any return type including primitive types and void, m can be static or non-static, m can have any visibility modifier, etc. In the patterns we use an if-statement as a short-hand for Java's branch and loop constructs. Due to a limitation of our pattern matching, Table I does not contain all such instanceof expressions that occur in branch conditions.

(1) T and X are non-compatible.
(2) m is user-defined.
(3) At least one of {T, X} is an interface.
(4) None of {T, X} is final.
(5) At least one of {T, X} is a user-type.

Condition (1) excludes simple inheritance cases that can be resolved by plainly using or subclassing T or X. Such simple cases can be covered with some of the existing tools and techniques. Condition (2) limits our search to the code that is currently under test. Conditions (3) and (4) exclude cases that are never satisfiable. That is, in Java no class can be a subtype of two non-compatible classes or be a subtype of a final type. Condition (5) focuses the search to cases in which there is likely no existing class readily available in the JDK that is a subtype of both T and X.

3.1.2. Patterns p2 and p3: Method Parameter Field and Method Return Value Check. Pattern 2, shown in Listing 3, is similar to pattern 1, but checks a method parameter's field value against an incompatible type.

```
class P { T t; .. }
class Q {
  M m(.., P p, ..) { // ..
    if (.. (p.t instanceof X))
    // ..
  }
}
```

Listing 3: Pattern 2 matches branching statements that branch on an instanceof dynamic type check of a method parameter's field with a type X, such that X and the field's declared type T are not compatible.

Pattern 3, shown in Listing 4, is similar to the previous patterns, except that here we test against a non-compatible method return-type T.

```
M m(..) { // ..
  if (.. (..).foo() instanceof X)
  // ..
}
```

Listing 4: Pattern 3 matches branching statements that branch on an instanceof dynamic type check of a method return value with a type X, such that X and the method's declared return type are not compatible.

3.1.3. "Relaxed" Patterns p1r, p2r, p3r. Patterns p1r, p2r, and p3r relax their p1, p2, and p3 counterparts, by not requiring that the instanceof-expression is part of a branch condition. We search for these patterns to gauge how often programmers check with non-compatible types.

p1r: M m(.., T t, ..) {.. (t **instanceof** X) ..}
p2r: M m(.., P p, ..) {.. (p.t **instanceof** X) ..}
p3r: M m(..) {.. ((..).foo() **instanceof** X) ..}

3.2. Annotation Patterns of Table II

Table II lists the subjects of Table I and their calls to reflection methods that access annotations. Similar to the patterns of Table I, applications typically access annotations as part of a branching decision. That is, the code under test checks if some class has a certain annotation. The entries of Table II have a similar flavor as the more general cases in the last two columns of Table I, as each annotation access method can be applied on two arbitrary parameter classes. A representative example is discussed in Listing 13 of Section 5.1.3, in which the code under test calls the Class.isAnnotationPresent(Class) method in an if-condition that guards the bulk of the method body. Within our sample of 35 open-source Java applications we found 223 instances.

4. SOLUTION (RQ2, RQ3)

In this section we describe our solution, first on an example from the Apache Active MQ subject, then by providing key details of the algorithm.

4.1. Overview and Example

Apache Active MQ implements the Java Message Service (JMS) standard message-broker interface [Hapner et al. 2002]. Method transform(Destination) of class ActiveMQDestination shown in Listing 5 (cut and formatted for space reasons, with adapted line numbers) takes a parameter of the JMS interface Destination and performs on the parameter several dynamic type checks. Figure 1 shows in UML class diagram notation [Booch et al. 2005] the types that are relevant for these type checks, i.e., the Destination subtypes. Execution branches in line 7 to lines 8–15 if the dynamic type of the method parameter is not null and a subtype of both Topic and Queue. Searching the core component's 1,731 production types, there is no such class.

```
public static ActiveMQDestination                                                    1
transform(Destination dest) throws JMSException {                                    2
  if (dest==null)                                                                    3
    return null;                                                                     4
  if (dest instanceof ActiveMQDestination)                                           5
    return (ActiveMQDestination) dest;                                               6
  if (dest instanceof Queue && dest instanceof Topic) {                              7
    String queueName = ((Queue) dest).getQueueName();                                8
    String topicName = ((Topic) dest).getTopicName();                                9
    if (queueName!=null && topicName==null)                                          10
      return new ActiveMQQueue(queueName);                                           11
    else if (queueName==null && topicName!=null)                                     12
      return new ActiveMQTopic(topicName);                                           13
    throw new JMSException(/* [cut] */);                                             14
  }                                                                                  15
  if (dest instanceof TemporaryQueue)                                                16
    return new ActiveMQTempQueue(((TemporaryQueue)dest).getQueueName());             17
  // [cut] three more if-(dest instanceof ..)-stmts                                  18
  throw new JMSException(/* [cut] */);                                               19
}                                                                                    20
```

Listing 5: The transform(Destination) method of the ActiveMQDestination class.

The test cases accompanying Apache Active MQ core contain a class that implements Destination, Queue, and Topic and can therefore reach lines 8–15. It appears as if this class, CombyDest, has been written in response to the bug report[4] it refers to, AMQ-

[4]The bug report AMQ-2630 is available at https://issues.apache.org/activemq/browse/AMQ-2630

2630. AMQ-2630 is categorized as bug, major, fixed. AMQ-2630 refers to a third-party class, AQjmsDestination of the Oracle Streams Advanced Queuing Java API, which similarly implements Destination, Queue, and Topic and caused a runtime exception in the transform method. It is not surprising that this bug went undetected, as reaching the various blocks of the transform method requires very specific kinds of classes and writing all of them manually may be viewed as tedious and may have therefore been skipped during manual unit testing.

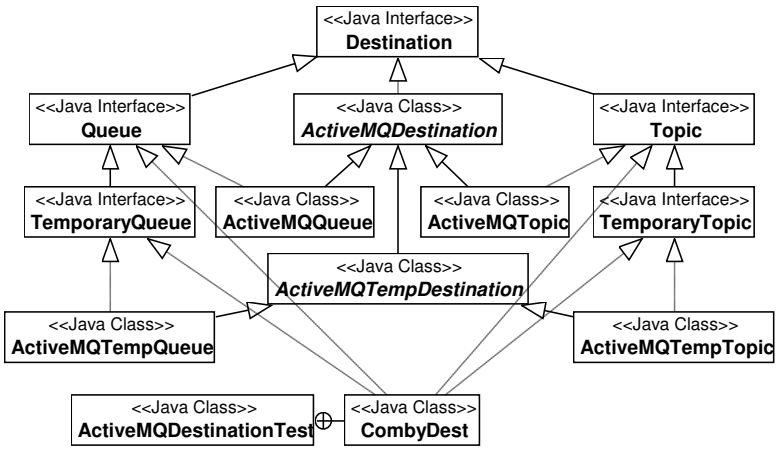

Fig. 1: Type hierarchy of the Destination interface used in the transform(Destination) method in Listing 5: Destination and those subtypes that are visible in the Apache Active MQ core component and relevant for the transform method; CombyDest is nested inside a test case.

Our key insight is that we can infer the necessary constraints while executing the testee. In the transform method example, we start execution with a dest parameter value of null. This will lead execution to take the (dest==null) branch in line 3 and return. Using code instrumentation, we can track such branch condition outcomes, represent them symbolically in terms of the method parameters, and systematically manipulate resulting path conditions. Similarly, we also encode formal method parameter types, the subtype hierarchy of the types involved, and general language rules, including the meaning of type modifiers such as abstract, final, etc.

In this example, we solve the constraint (dest!=null), construct a corresponding non-null Destination value, and use it for the next iteration. Continuing in this fashion, we eventually build a path condition of (dest!=null) && (dest instanceof Queue) && (dest instanceof Topic). As no such type exists yet, this constraint system is unsatisfiable. At this point we introduce mock types. Whereas existing types have fixed super types, we encode the super types of a mock type as a sequence of variables and let the constraint solver determine if there exists a solution that satisfies the entire constraint system. This is an elegant solution, as it integrates well with the other constraints collected during dynamic symbolic execution. That is, we can create a satisfying mock class and continue exploration with a mock class instance. Now we can continue collecting constraints on the mock class, such as constraints on the values returned by its methods, for example, (queueName!=null) in line 10.

In summary, while on a high level our approach follows the work on Pex, EXE, and DART, our approach differs in two important ways, which correspond to our research

questions RQ2 and RQ3. First, we model precisely the type constraints of an object-oriented program and can therefore reason about complex type constraints including ones that stem from dynamic type checks and reflection (RQ2). Second, not part of this example, we model the constraints that arise from annotations (RQ3).

4.2. Algorithm

At a high level, our approach follows the idea of dynamic symbolic execution (DSE) [Godefroid et al. 2005; Godefroid 2007; Tillmann and de Halleux 2008; Elkarablieh et al. 2009], as illustrated by our main function, Function DSE. Initially, the program counter pc points to the testee's first bytecode instruction. The instruction is executed by a normal Java virtual machine represented by evalDynamic. The instruction is then evaluated symbolically by the symbolic evaluator evalSymbolic, which mirrors the behavior of evalDynamic in a symbolic representation of the program state.

Execution proceeds with the next bytecode (pc') as determined by the concrete execution. This means that the symbolic execution follows the control flow decisions of the concrete execution. Each subsequent bytecode instruction is also executed both concretely and symbolically. This symbolic shadowing of the concrete execution allows us to build a complete symbolic representation of the program's runtime state, including the state of the method invocation stack, the state of each operand stack and all local variables, and the full state of all heap objects.

Function DSE is the main routine of our dynamic symbolic executor. The symbolic evaluator evalSymbolic follows the control flow decisions of the regular Java execution evalDynamic. Symbolic entities are capitalized; pc is the program counter.

```
pc ← first_bytecode(testee);
thread ← thread executing testee code;
Thread ← fresh symbolic Thread;
heap ← jvm heap;
Heap ← fresh symbolic Heap;
while true do
    pc' ← evalDynamic(pc, thread, heap);
    evalSymbolic(pc, thread, Thread, heap, Heap);
    pc ← pc';
```

The evalSymbolic function may read the concrete program state to get access to the current, fully evaluated, concrete state. This is important when Java bytecode instructions are mixed with native code, which occurs frequently in reflection code. That is, after the execution of a native code fragment, evalSymbolic accesses the concrete program state to retrieve the values computed by the native code, represents them as symbolic literals, and thereby brings the symbolic shadow state back in sync with the shadowed concrete program state.

Figure 2 illustrates the key steps of our approach. Initially we invoke DSE on the code under test, providing as input a given or default value such as zero or null, to collect an initial path condition. Then, in each iteration, we try to cover a new execution path through the code under test. Following are the key steps taken in each iteration.

(1) Invoke DSE on the given input values, collecting branch outcomes in a path condition and collecting type and annotation constraints (using evalSymbolic).
(2) Invert one conjunct of one of the collected path conditions to obtain a new path condition.
(3) Map each encountered reference type to a constraint literal (e.g., using Table IV).

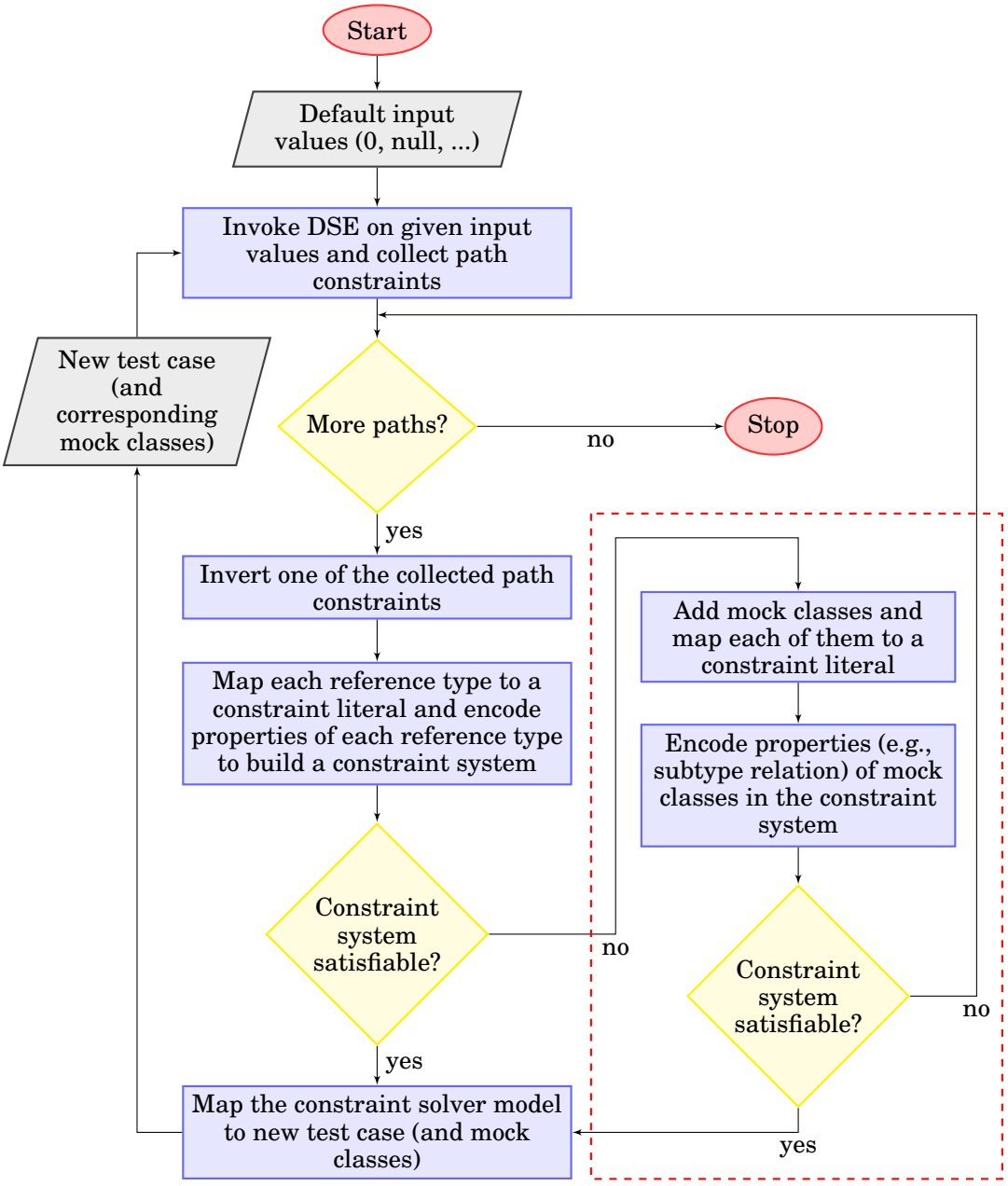

Fig. 2: High level flow-chart view of the algorithm for generating test cases and mock classes. Mock classes extend the standard flow-chart of dynamic symbolic execution. Elements that are not in the standard flow-chart are shown in the red dashed box.

(4) Encode properties of the encountered reference types (e.g., using Table III), including their subtype relation.
(5) If the new path condition (of step 2) plus the type encoding of steps 3 and 4 are not satisfiable, update the constraint system as follows.
    (a) Add mock classes and encode each of them as a constraint literal.
    (b) Represent the mock class subtype relation and other properties with constraint variables.
(6) If the constraint system is satisfiable, map the constraint variables, including the properties of the mock classes, from the constraint solver model to a concrete new test case (input values and concrete mock classes).

4.3. Constraint Encoding

In order to solve constraints, we map the symbolic expressions built by our symbolic bytecode interpreter evalSymbolic to the input language of an off-the-shelf SMT-solver. At a high level, we map each reference type encountered during execution to a constraint literal, encode their subtype hierarchy as a sparse boolean-valued two-dimensional array called Supertypes, and encode the super-types of a mock-type as boolean variables within that array.

Relative to these constructs we can then express the constraints that arise from executing program elements, including dynamic type checks and reflection, i.e., in function evalSymbolic. For example, when encountering an **instanceof** bytecode instruction, evalSymbolic pops the top symbolic expression from the symbolic operand stack of the current method. This symbolic expression represents the reference that **instanceof** checks for being an instance of a given type. Then evalSymbolic retrieves this type from the bytecode. We then build a symbolic Subtype expression that represents that the dynamic type of the retrieved symbolic expression is indeed a subtype of the given type. To match the semantics of the Java bytecode **instanceof** instruction, we finally build a symbolic if-then-else expression (Ite) that returns 1 if the symbolic reference is non-null and of the given type and else returns 0.

Besides encoding constraints arising during program execution, we use the symbolic functions of Table III to encode the properties of the encountered reference types. That is, we assert type properties such as a type being abstract, an interface, an array, and final. Having these basic facts and the subtype relation asserted, we can encode the desired properties of variables such as the method parameters. For example, we encode the rule that the dynamic type of a method parameter has to be an array or non-abstract subtype of the parameter's declared type.

4.4. Dealing with Annotations

An annotation definition **public** @interface A {..} in Java essentially defines a special interface named A. Programs can use this annotation type A in ways similar to using any other interface. For example, programs can use A in instanceof expressions such as **if** (p **instanceof** A) {..} or in .class expressions and reflective method invocations such as **if** (p.getClass().isAnnotationPresent(A.**class**)) {..}. Given that an annotation can be used like any other interface, we treat annotations like interfaces. That is, we treat each annotation as a type and encode it in the subtype relation together with all other class and interface types.

In addition to behaving like an interface, there are a few additional operations a program can perform that are exclusive to annotations. For example, the isAnnotationPresent(Class a) reflection method determines if the receiver instance is annotated with annotation a.

Function evalSymbolic(pc, thread, Thread, heap, Heap) is our symbolic bytecode interpreter. Symbolic functions are capitalized and summarized in Table III. By default we push, pop, and read operands from the top method invocation frame of thread; dyntype and subtype return the dynamic type of a reference and determine if two types are in a subtype relation; imd = immediate operand of a bytecode instruction; Ite(E,A,B) = **if**(E) A **else** B; cp = runtime constant pool of the current class; switch cases do not fall through.

```
bytecode, imd ← bytecode(pc);
switch bytecode do
    case aload                                              // read reference from local var.
        Push(Locals[imd]);                                  // and push on operand stack
    case checkcast                                          // type cast
        Ref ← Topoperand();
        Type ← Reftype(type(cp[imd]));                      // literal type
        TypeConstraint ← Subtype(Dyntype(Ref),Type);
        if subtype(dyntype(ref),type) then
            Path ← Path+TypeConstraint;
        else
            Path ← Path+Not(TypeConstraint);
    case instanceof                                         // reference instanceof type
        Ref ← Pop();
        Type ← Reftype(type(imd));                          // literal type
        TypeCnstr ← Subtype(Dyntype(Ref),Type);
        Push(Ite (And(Not(Null(Ref)), TypeCnstr), Bv32(1), Bv32(0)));   // JVM bool is int
    case ldc                                                // ".class" expression such as X.class
        switch cp[imd] do                                   // type of constant to be loaded
            case "reference to a class"                     // Class constant
                Type ← Reftype(type(cp[imd]));
                Push(Classref(Type));
            case ..                                         // .. int, String, etc. constants
    case invokevirtual                                      // method call
        switch cp[imd]) do                                  // signature of called method
            case "Object.getClass()"
                Obj ← Pop();
                Push(Classref(Dyntype(Obj));
            case "Class.isAssignableFrom(Class)"
                Param ← Type(Pop());
                Receiver ← Type(Pop());
                Push(Ite(Subtype(Param,Receiver), Bv32(1), Bv32(0)));   // literal ints
            case "Class.isAnnotationPresent(Class)"
                Param ← Type(Pop());
                Receiver ← Type(Pop());
                Push(Ite(Annotated(Receiver,Param), Bv32(1), Bv32(0)))
            case ..                                         // .. other methods
    case ..                                                 // .. other bytecode instructions
```

Table III: Functions we use in evalSymbolic to encode program and type constraints in the input language of an SMT-solver. Supertypes is a two-dimensional symbolic array, which has RefType index types and Bool values. Subtype is a convenience access with Subtype(A,B) = (Supertypes[A])[B]. The unary functions are uninterpreted functions, i.e., the SMT-solver can define them freely except that a solution has to satisfy the constraints collected during dynamic symbolic execution. Although both DynType and Type have the same signature, i.e., they both map a symbolic reference to a symbolic type, they capture different concepts. DynType maps a reference to its dynamic type, whereas Type maps a reference to a java.lang.Class Java reflection class object to the symbolic type it represents.

Function	Type signature
Null	Ref
Null_Type	Reftype
Dyntype	Ref → Reftype
Classref	Reftype → Ref
Type	Ref → Reftype
Supertypes	Array: Reftype → (Reftype → Bool)
Subtype	(Reftype, Reftype) → Bool
Annotated	(Reftype, Reftype) → Bool
Abstract	Reftype → Bool
Array	Reftype → Bool
Final	Reftype → Bool
Interface	Reftype → Bool

The concept of an entity E having an arbitrary number of annotations has much in common with the concept of a type E having an arbitrary number of super-interfaces. That is, both concepts can be modeled as a sequence of simple E-has-X constraints that can be captured in a boolean-valued matrix. The straight-forward approach is therefore to create another large boolean-valued matrix in the constraint solver to encode the annotations that individual code elements have. However, we found that in practice subtype constraints rarely conflict with annotation constraints. That is, programs rarely check if a type is a subtype of A and is not annotated with A (or vice versa). That is, as part of our survey of the 35 open-source applications of Section 3, we did not encounter such code. To keep constraints compact, we therefore encode subtype and annotation constraints in the same matrix.

4.5. Generating Mock Class Method Bodies

In several cases the mock class generator has to generate meaningful method bodies. For example, the code under test may have an abstract type such as an interface as the formal type of a parameter, may call instance methods on the parameter value, and may use values returned from such calls in subsequent branching decisions. If no class exists that implements such an interface, we have to generate mock classes that contain meaningful method bodies that can satisfy such subsequent branching decisions. To generate such method bodies, at a high-level, we use the following algorithm.

(1) Initially, for each abstract method of each super-type, generate a method body that just returns a default value, i.e., null or zero.
(2) During execution, in the i-th invocation of a mock class method, in the symbolic state, replace the literal return value with a fresh symbolic variable that is labeled with the current invocation count i. If a mock method is called N times the mock method will have N associated symbolic variables labeled 1 to N.
(3) Build new constraints relative to the new variables and issue them to the constraint solver.

(4) If the constraint solver finds a solution (model), map the model back to corresponding test input values and generate corresponding new mock classes.
   (a) For each mock method, create a list of the model's solutions of the N symbolic variables associated with the mock method, ordered by their invocation label.
   (b) Generate a mock method body that returns in the i-th method invocation the i-th list value.
   (c) After N invocations, the method body returns the default zero or null value.

4.6. Implementation in Dsc

Dsc is a dynamic symbolic execution engine for Java bytecode. Dsc encodes the constructs of the Java programming language as constructs of an underlying SMT-solver. Table IV summarizes how we encode the types of the Java programming language and the boolean constraint type in the input language of an SMT-solver. Currently missing is support for floating point numbers.

Table IV: Encoding of the core elements of a Java program, used in Table III as well as in the evalSymbolic function. In braces are the concrete encoding we use for the underlying SMT-solver; integral = one of {int, boolean, short, byte, char}; the array notation "Array: A $\rightarrow$ B" is for a symbolic array of index type A and value type B.

Java program element	Encoding
integral	Bv32 (32-bit bit-vector)
long	Bv64 (64-bit bit-vector)
reference	Ref (30-bit bit-vector)
array of integral	Array: Bv32 $\rightarrow$ Bv32
array of long	Array: Bv32 $\rightarrow$ Bv64
array of reference	Array: Bv32 $\rightarrow$ Ref
instance field: integral	Array: Ref $\rightarrow$ Bv32
instance field: long	Array: Ref $\rightarrow$ Bv64
instance field: reference	Array: Ref $\rightarrow$ Ref
reference type	Reftype (nat. number)
constraint, e.g.: on control	Bool (boolean)

As most dynamic symbolic execution systems, Dsc supports loops. For example, the Xalan subject method dispatchNodeData of Listing 6 contains a loop. With mock classes Dsc achieves a higher code coverage of this code than Pex or the random test case generator Randoop (see Tables VI and VII).

Compared to Pex, the strategy for handling loops is currently weak in Dsc. Specifically, for some loops, Dsc may spend a lot of time on exploring many different paths through these loops. The reason of this sub-optimal performance on loops is that Dsc currently implements a fixed strategy (i.e., depth-first search) for deciding which execution path to explore next. This decision takes place in the "Invert one of the collected path constraints" node of Figure 2 and determines the order in which DSE explores the reachable execution paths through the code under test.

Xie et al. [2009] pioneered a flexible, fitness-function guided strategy for picking the next execution path and implemented it in Pex. This strategy can minimize the time spent on a single loop, to quickly also explore other parts of the code under test. Reimplementing this fitness-function guided strategy in Dsc is part of future work. We view adding this strategy as orthogonal to mock classes, as it does affect the way in which we build complex type constraints, solve them with a constraint solver, or map constraint solver solutions back to test case and mock classes.

Our dynamic symbolic execution system implements a few standard optimizations [Cadar et al. 2008], to minimize the size of the symbolic expressions before sending them to the SMT-solver. For example, we hash-cons, normalize, and rewrite sym-

bolic expressions, and treat the path constraint as an ordered set, to add each symbolic constraint only once.

```
public static void dispatchNodeData( Node node, ContentHandler ch, int depth )
  throws org.xml.sax.SAXException
{
  switch (node.getNodeType()) {
    case Node.DOCUMENT_FRAGMENT_NODE :
    case Node.DOCUMENT_NODE :
    case Node.ELEMENT_NODE :
      for (Node child = node.getFirstChild(); null != child; child = child.getNextSibling()) {
        dispatchNodeData(child, ch, depth+1);
      }
    break;
    /* [cut 4 cases] */
    case Node.ATTRIBUTE_NODE :
      String str = node.getNodeValue();
      if (ch instanceof CharacterNodeHandler)
        ((CharacterNodeHandler)ch).characters(node);
      else
        ch.characters(str.toCharArray(), 0, str.length());
      break;
    default : // ignore
      break;
  }
}
```

Listing 6: The dispatchNodeData method under test from the Xalan project (cut and formatted for space reasons). The method contains a for loop and requires a mock class with a certain type to achieve high coverage.

4.7. Implementation Details by Example

To illustrate the mechanics of our approach, we use the example foo method of Listing 7.

```
public @interface A { /* .. */ }
public interface I { public int m1(); /* .. */ }
public interface J { public int m2(); /* .. */ }
public class C {
  public static void foo(I i) {
    i.m1();
    if (i instanceof J) {
      J j = (J) i;
      int x = j.m2();
      if (i.getClass().isAnnotationPresent(A.class)) {
        m3();
        if (x==10)
          m4();
        // ..
      }
    }
  }
}
```

Listing 7: To reach or cover the m3 method call in line 11 we need an instance of a class that implements interfaces I and J and has annotation A. To cover line 13, such a class also has to have a method m2 that returns value 10 when called in foo line 9.

Besides the basic mock class algorithm, this example allows us to highlight the additional solution components of (a) dealing with annotations (Section 4.4) and (b) generating mock class method bodies (Section 4.5). The Listing 7 foo code is a variation of our motivating example of Section 1.2. New is the annotation processing in line 10. The second change is the if-statement of line 12, which branches based on the value returned by the m2 method that is called on foo parameter i. Covering the former requires reasoning about annotations. Covering the latter requires generating a mock class that contains a custom method body.

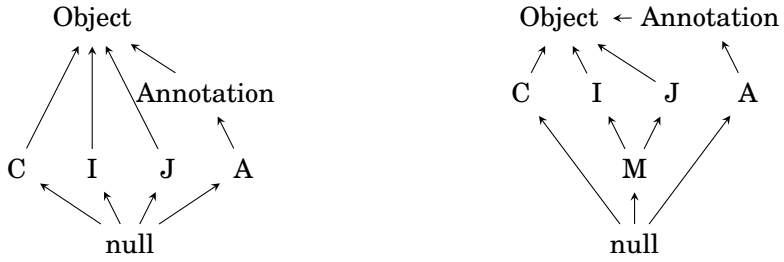

Fig. 3: Direct subtype relation for the Listing 7 method foo under test (left) and a desired solution for covering the method call in line 9 (right).

4.7.1. Generating a Mock Class. The method under test in Listing 7, foo, uses two interfaces, I and J, as well as annotation A. The foo method declares parameter i of type I, calls the m1 method on it in line 6 and later possibly the m2 method in line 9. To call m2, as m2 is defined by J, foo casts i to J. To reach the m2 method call in line 9, we therefore need an instance of I that is also an instance of J.

Let's assume we are executing our main algorithm of Section 4.2. At the end of the previous iteration we have created a simple mock class that only implements I. In the first step of the current iteration we instantiate this class. Recall that in this step our algorithm executes the code under test both dynamically and symbolically (using evalSymbolic). Executing the foo method on this instance leads to taking the false branch of the (i **instanceof** J) branch condition. This foo execution yields constraints including the following.

— Dyntype(i) != Null_Type
— Subtype(Dyntype(i),I))
— Not(Subtype(Dyntype(i),J)))

After inverting the last constraint in step 2 we encode the type constraints with steps 3 and 4. We number Java reference types (classes, interfaces, arrays, enums) in the order we discover them, from zero to N. We discover types during dynamic symbolic execution of a program path in one of several events, for example, when symbolically executing a static field access of a previously undiscovered class. The first two numbers, 0 and 1, represent the null type and java.lang.Object, respectively.

We encode the subtype hierarchy in a two-dimensional Bool-valued symbolic array, Supertypes[0..N][0..N]. We use arrays, because the Z3 constraint solver we use [Moura and Bjørner 2008] allows us to create arrays that have a default element, such as false. This is convenient, as subtype relations are typically sparse, which means a default value of false allows us to encode the subtype relation as a compact constraint.

Table V: Subtype relation matrix of Listing 7: Encoding of the reference types (left column); Obj = Object; An = Annotation; Mx are boolean variables; t in (row A, column B) = A is a subtype of B.

		null	Obj	C	M	I	J	An	A
0	null	t	t	t	t	t	t	t	t
1	Obj		t						
2	C		t	t					
3	M		t	MC	t	MI	MJ	MAn	MA
4	I		t			t			
5	J		t				t		
6	An		t					t	
7	A		t					t	t

Step 5 attempts to solve the constraints with the existing types. The left part of Figure 3 shows the direct subtype relation of the types involved in Listing 7 as a graph. However no existing type satisfies the constraints we collected in this iteration. We thus introduce mock class M, which yields the subtype relation matrix of Table V.

```
public void test1() throws Exception {
  MockClass local1 = new MockClass();
  C.foo(local1);
}
```

Listing 8: Unit test case generated by Dsc for the foo method of Listing 7. Together with the mock class of Listing 9, this test case can take the true branch of the if-statement of foo line 7.

By assigning the boolean variables MC, MI, MJ, MAn, and MA, the constraint solver chooses where in the subtype lattice to place the mock class. As the new constraint system is satisfiable, step 6 generates a new test case (Listing 8) and a concrete mock class (Listing 9) that implements interfaces I and J. Together, generated test case and mock class will allow our algorithm in its next iteration to cover the m2 method call in line 9, which is guarded by the (i **instanceof** J) expression.

```
public class MockClass implements I, J {
  public MockClass() {}
  public int m1() { return 0; }
  public int m2() { return 0; }
  // ..
}
```

Listing 9: Mock class generated for the Listing 8 test case.

Our prototype tool implementation generates JUnit test cases [Beck and Gamma 1998]. JUnit is one of the main test execution frameworks for Java [Beust and Suleiman 2007]. Listing 8 shows the generated JUnit test case for the example of Listing 7.

4.7.2. Dealing With Annotations. The if-condition in line 10 checks if the dynamic type of the method parameter i is annotated with annotation A. To reach the block that starts in line 11, we therefore need an instance of I and J that is annotated with A.

Let's assume we are executing our main algorithm of Section 4.2. At the end of the previous iteration we have created a mock class that implements I and J. In the first step of the current iteration we instantiate this class and use it while executing the foo method both dynamically and symbolically. Executing foo on this instance leads to taking the false branch of the branch condition in line 10. This foo execution yields constraints including the following.

— Dyntype(i) != Null_Type
— Subtype(Dyntype(i),I))
— Not(Subtype(Dyntype(i),J)))
— Not(Annotated(Type(Classref(Dyntype(i))),A))

The remaining steps are similar to Section 4.7.1, except for the treatment of the last constraint. The last constraint captures that the class of i was not annotated with A. Since in our implementation we encode annotation constraints in the same matrix as subtype constraints, Annotated is essentially an alias of the Subtype constraint.

As in Section 4.7.1, the constraint system is not satisfiable with the existing types. Adding mock classes, the system becomes satisfiable. In the constraint solver solution, MA is true, since we share the matrix between sub-typing and annotations. Since A is an annotation type we interpret the solution to mean that M must have annotation A. Dsc generates a test case and mock class that are very similar to Listings 8 and 9, except that the generated mock class also has annotation A.

4.7.3. Generating a Mock Class Method Body. The deepest nested if-statement shown in Listing 7, line 12, branches based on the value returned by the m2 method called on i in line 9. To cover line 13, we therefore have the additional requirement that i also has to be an instance of a class whose method m2 returns value 10 in line 9.

Let's assume we are executing our main algorithm of Section 4.2. At the end of the previous iteration we have created a mock class that implements I and J and has annotation A. In the first step of the current iteration we instantiate this class and use it while executing the foo method both dynamically and symbolically.

In this execution of the foo method, foo calls m2 on the generated mock class in line 9. Method m2 returns immediately, returning 0 (as shown in Listing 9). The symbolic evaluator replaces the corresponding symbolic zero literal with the fresh symbolic variable M_m2_1. The name M_m2_1 encodes that this value was returned by the first invocation of mock class M method m2.

This execution of foo ultimately triggers the false branch of the branch condition in line 12 and yields constraints including the following.

— Dyntype(i) != Null_Type
— Subtype(Dyntype(i),I))
— Not(Subtype(Dyntype(i),J)))
— Not(Annotated(Type(Classref(Dyntype(i))),A))
— M_m2_1 != 10

The remaining steps are similar to Section 4.7.2. The last constraint captures the fact that the value returned from the first invocation of mock class M method m2 was not 10. However inverting this constraint results in a constraint system that can be solved similarly to the one of Section 4.7.2. Hence Dsc generates a test case such as Listings 8 with the mock class of Listing 10.

```
@A public class MockClass implements I, J {
  public MockClass() {}
  public int m1() { return 0; }
  public int m2() { return 10; }
  // ..
}
```

Listing 10: The generated mock class to cover the method call m4 in Listing 7.

5. EXPERIENCE (RQ4, RQ5)

In this section we explore research questions RQ4 and RQ5 and compare a prototype implementation of our mock class generation technique with related approaches. First we evaluate the impact of adding mock class generation to an existing dynamic symbolic execution system (RQ5). Then we also compare our implementation with a baseline technique, random test case generation (RQ4).

For random test case generation (RQ4) we use the Randoop [Pacheco et al. 2007] random test case generator for Java. Compared with our Dsc tool, Randoop is more mature and has been used widely in the research literature.

For dynamic symbolic execution we use our own Dsc tool, as we have full access to the tool's source code and can therefore implement mock class generation in a straightforward fashion. The closest related dynamic symbolic execution system is Pex with its recent Moles extension [Tillmann and de Halleux 2008; de Halleux and Tillmann 2010]. Pex is more mature than Dsc and supports a wider range of programming language features. As we do not have source code access to Pex we compare our tool with and without mock class generation with a recent Pex release.

As subjects we used a subset of the Table I subject methods that both contain patterns we are interested in and could be analyzed with the current version of Dsc. Since efficient handling of loops was not a focus of the study and the current version of Dsc is not optimized for handling loops, we excluded code from the experiment if for the piece of code Dsc did not terminate normally within 60 seconds. This timeout lead us to exclude several pieces of code that contain loops.

We thereby selected sample code from 10 subject applications and also included the motivating example from Listing 7. The number of subject methods together with their lines of code and number of branches are summarized in Table VII. In these subjects we only had to do small modifications, i.e., we only changed some of the subject method modifiers to allow analysis with the current version of Dsc.

The prototype status of Dsc also dictated the way we counted coverage. That is, we only counted the coverage in the methods under test and not the coverage of the methods called by the methods under test. However this is not a fundamental limitation of the approach. Dynamic symbolic approaches such as Dsc and Pex by design capable of inter-procedural analysis and can cover deep call chains, given a certain level of engineering maturity.

Pex targets .Net languages such as C#. To compare Dsc with Pex, we manually translated the subject code from Java to C# and explored it with a recent version of Pex[5]. Such a translation is difficult for two reasons. First, the semantics of Java and C# language constructs are similar but different. This requires us to carefully consider these semantics and compensate for semantic differences in the translated code.

The second difficulty is that our subjects are part of large third-party applications. Such code typically has many dependencies. Since it is not feasible to translate entire applications for our experiments, we had to simplify the subject code significantly, to remove these dependencies. Specifically, we removed many method calls and removed much surrounding code.

Our simplified subjects are summarized in Table VI. Compared to the original subject code of Table VII, for example, we removed about every third line of code. The simplified versions of both the Java and C# code samples are available on our project web site[6]. To count lines and branches in Java and C# code we used the eCobertura[7]

[5] v0.94.51006.1, available at http://research.microsoft.com/projects/pex/
[6] http://cseweb.uta.edu/~mainul/MockClass/
[7] v0.9.8, available at http://ecobertura.johoop.de/

and NCover[8] (along with TestDriven.net[9]) coverage tools; eCobertura is an Eclipse plugin for Cobertura[10].

To collect measurements we mainly used a 2.26GHz Core2 Duo processor machine with 3GB RAM on a 32bit OS. Randoop measurements are an exception. As Randoop frequently ran out of memory, we took all Randoop measurements on a machine that has more memory, a 2.33GHz Xeon processor machine with 32GB RAM on a 64bit OS. Beyond enabling different amounts of heap memory, these two machines perform similarly. That is, for small subjects, Randoop performed similarly on both machines.

5.1. Higher Coverage Than Basic Dynamic Symbolic Execution (RQ5)

Table VI shows the run times of Dsc+Mock, Dsc, Pex, and Randoop. The dynamic symbolic approaches (Dsc+Mock, Dsc, and Pex) ran until they either reached their default exploration limits (such as the number of branches explored) or exhausted all execution paths they could cover. Dynamic symbolic execution approaches rely on third-party constraint solvers which may find more or better solutions if run longer. To give the dynamic symbolic approaches ample time to find solutions, we used for each of them the same time-out setting of 100 seconds. This was a conservative choice, as none of these approaches needed as much time for any of its constraint systems.

A random test case generator such as Randoop does not keep track of all possible execution paths but builds a set of those program values it can feed to the methods under test as input parameters. A random generator then runs until it exhausts all possible combinations of those method input values or until a timeout. In these experiments we aimed to be conservative and gave Randoop enough time to maximize its coverage. That is, we picked one subject randomly and first ran Randoop on the subject with a high time-out value, which resulted in a large set of randomly generated test cases. Then we reduced the timeout value, yielding a smaller set of test cases. We kept reducing the timeout value as long as the (shrinking) set of test cases achieved the same line and branch coverage of the subject methods. For the experiments in Table VI this let us to a timeout of 60 seconds, which we then adopted for the remaining subjects.

We have not performed further efforts on minimizing the timeout value (while keeping the coverage constant). So it is possible that for several subjects, Randoop may produce the same coverage with lower timeout values. However, we feel that our conservative approach is a good model of how people would use a random test case generator in practice. We assume a real-world test engineer would also pick a timeout value that appears reasonable based on the size of the subject and avoid lengthy timeout value minimizations.

Table VI shows that the runtime of Dsc+Mock was about three times higher than the runtime of Pex+Moles and some nine times higher than (plain) Dsc. One key insight here is that in most of the subject methods a large portion of the code is guarded by branch conditions similar to the patterns described in Section 3. As Dsc+Mock usually covers such branch conditions, whereas the other techniques do not, Dsc+Mock explores a significantly higher number of lines and branches and therefore takes longer. However given the difference in engineering maturity between Dsc and Pex, we expect these runtime differences to change significantly with future versions of Dsc and Pex.

Table VI also shows that among the dynamic symbolic approaches, Dsc achieved the lowest code coverage. This is not surprising, as basic Dsc lacks the mock classes of Dsc+Mock and the higher engineering maturity and Moles extension of Pex+Moles. Pex both spent more time, generated more test cases, and achieved a higher code cov-

[8]v1.5.8_beta, available at http://downloads.ncover.com/NCover-1.5.8.zip
[9]v3.4.2803_personal_beta3, available at http://www.testdriven.net/download.aspx
[10]Available at http://cobertura.sourceforge.net/

Table VI: Dsc with mock class generation (Dsc+Mock) vs. plain Dsc, Pex+Moles, and Randoop on simplified code samples. Dsc+Mock covered the highest percentage of code lines (%L) and code branches (%B). M, L, B are the number of subject methods, their lines of code, and their number of branches; t[s] is the runtime in seconds and T is the number of test cases generated by the respective tool. Overall, the dynamic symbolic approaches (Dsc, Dsc+Mock, and Pex+Moles) spent less time and fewer test cases but achieved higher code coverage than the random approach (Randoop).

Subject				Randoop				Pex+Moles				Dsc				Dsc+Mock		
	M	L	B	T	t[s]	%L	%B	T	t[s]	%L	%B	T	t[s]	%L	%B	T	t[s]	%L	%B
Listing 7	1	7	4	1	60	14	25	2	1.1	28	25	1	0.4	14	25	4	1.4	100	100
ActiveMQ	4	28	10	15	60	14	40	8	4.5	29	60	4	1.4	14	40	9	10.0	71	80
ASM	2	18	6	2	60	33	50	6	2.0	33	50	2	0.8	33	50	6	4.0	100	100
BCEL	2	15	6	2	60	20	50	9	2.0	20	50	2	1.2	20	50	15	9.4	100	100
Drools	3	51	16	530	60	8	31	14	4.8	47	50	3	1.5	8	25	20	11.0	90	88
Guice	2	20	7	6	60	40	43	6	3.0	50	57	2	1.2	30	57	10	14.8	90	86
JaxLib	5	30	12	6	60	23	42	15	6.0	33	50	5	1.5	27	42	21	18.0	100	100
JFreeChart	4	40	12	520	60	20	33	15	4.0	65	50	4	1.5	20	33	12	10.5	100	100
JMonkeyEngine	2	17	8	460	60	47	38	4	1.5	47	38	2	0.8	47	38	4	2.7	82	75
Soot	6	46	18	440	60	41	33	14	6.6	54	44	6	2.0	48	44	20	14.2	100	100
Xalan	3	48	23	3	60	17	26	18	4.0	25	35	3	1.2	17	26	40	23.5	88	91
Total	34	320	122	1,985	660	24	35	111	39.5	42	46	34	13.5	24	37	161	119.5	92	93

erage than Dsc. Pex benefits from its Moles extension, which provides a limited form of class generation and can therefore cover additional branches and code lines. Dsc+Mock in turn spent more time and test cases than Pex and achieved the highest code coverage of all approaches. In summary, Dsc+Mock achieved some 50% higher coverage than Pex+Moles and some 60% higher coverage than basic Dsc and Randoop. These results show that Randoop, Dsc, and Pex+Moles cannot generate the mock classes that are needed to cover code that uses multiple-inheritance constraints, reflection, and annotations.

In the following we describe some of the subjects of Table VI in more detail. The subject source code is abbreviated and reformatted for space.

5.1.1. Listing 7 Example Method. Listing 11 shows our manual Java to C# translation for the Listing 7 motivating example. The resulting C# code differs from the original Java version in lines 1, 8, and 12, due to the different naming of operators in Java and C# (for example, **instanceof** in Java versus **is** in C#) and built-in classes and reflection methods. However the semantics are very similar, as we picked C# constructs that are very similar to (and often behave exactly the same as) their Java counterparts.

Without mock classes, (basic) Dsc can only cover the first statement, which corresponds to line 6 in Listing 11. That is, Dsc only generates a test case that invokes the testee with the default parameter value of null. Line 6 invokes a method on this parameter, which triggers a null pointer exception. As the method does not contain an exception handler, this exception abruptly terminates the execution of this method.

Pex+Moles covers the first two statements in lines 7 and 8. That is, Pex first also just generates an initial value of null. Beyond that, Pex also generates a class that is a subtype of interface I, which is the formal parameter type of the method foo under test. Using an instance of this custom subtype of I in the second generated test case, the test execution reaches the branch decision in line 8. Covering additional parts of this method requires a mock class that is a subtype of both I and J. To cover the entire method, this mock class also has to have an annotation of type A.

```
public class A: System.Attribute { /* .. */ }                                  1
public interface I { public int m1(); /* .. */ }                               2
public interface J { public int m2(); /* .. */ }                               3
public class C {                                                               4
  public static void foo(I i)                                                  5
  {                                                                            6
    i.m1();                                                                    7
    if (i is J)                                                                8
    {                                                                          9
      J j = (J) i;                                                             10
      j.m2();                                                                  11
      if (i.GetType().GetCustomAttributes(true)[0] is A)                       12
      {                                                                        13
        m3();                                                                  14
        // ..                                                                  15
      }                                                                        16
    }                                                                          17
  }                                                                            18
}                                                                              19
```

Listing 11: C# version of the Listing 7 motivating example.

As described in Section 4.7, Dsc+Mock can reason about such multiple-inheritance constraints, as well as constraints imposed by reflection and annotations. Dsc+Mock thereby creates 4 test cases and 3 mock classes, and covers all lines of the foo method.

5.1.2. Multiple-Inheritance Mock Classes for JaXLib. The JaXLib core library provides efficient data structures, I/O streams, etc. Listing 12 shows the readfully(..) method of the JaXLib class jaxlib.io.stream.IOStreams. The method uses the abstract class java.io.InputStream as a formal parameter type. Line 4 checks if the parameter is also an instance of the DataInput interface. This check may return false, as InputStream is not a subtype of DataInput. To cover this method, if we do not want to use an existing class because of the potential for unintended I/O side-effects, we need to generate an InputStream that is also a DataInput. Indeed, our generator covers this code with mock classes, but cannot cover it without. Pex+Moles does not generate a class that is a subtype of both InputStream and DataInput, and therefore cannot cover line 5.

```
public static void readFully(InputStream in, byte[] dest, int off, int len)   1
  throws IOException                                                           2
{                                                                              3
  if (in instanceof DataInput)                                                 4
    ((DataInput) in).readFully(dest, off, len);                                5
  // ..                                                                        6
}                                                                              7
```

Listing 12: Covering the readFully method requires a parameter that is a subtype of two incompatible types, InputStream and DataInput.

5.1.3. Annotated Mock Classes for a 3D Game Engine. jMonkeyEngine[11] is a 3D game engine that has been used by game studios and university classes. This game engine has an abstract class named Serializer, which serializes objects such that they can be transferred over a network. One of its methods is the registerClass method under test shown in Listing 13.

One of the inputs of the registerClass method is the cls parameter of type Class. Class is a Java reflection type that can only be instantiated by the Java virtual machine. Each instance of the Class class represents a corresponding Java type. For example, at program runtime there will be one instance of Class that represents the Serializer class.

```
public static SerializerRegistration registerClass(Class cls, boolean failOnMiss) {   1
  if (cls.isAnnotationPresent(Serializable.class))                                    2
  {                                                                                   3
    Serializable serializable = (Serializable) cls.getAnnotation(Serializable.class);  4
    short classId = serializable.id();                                                5
    /* [cut] − bulk of the method */                                                  6
  }                                                                                   7
  if (failOnMiss)                                                                     8
    throw new IllegalArgumentException("Class is not marked @Serializable:" + cls );  9
  return null;                                                                        10
}                                                                                     11
```

Listing 13: A jMonkeyEngine method under test that requires a mock class with a certain annotation.

To cover the bulk of the registerClass method, we need to pass as input an argument Class instance such that the underlying Java type has a certain annotation. That is, registerClass checks in line 2 if the underlying type has an annotation of type @Serializable. Listing 14 shows this annotation type. If the underlying type does not have such an annotation, the method call in line 2 returns null, which is dereferenced

[11] v3.0, build 2011-08-29, available at http://jmonkeyengine.com/nightly/

in the subsequent statement, yielding a null pointer exception, which terminates the method execution.

Without mock classes, Dsc can only cover the first statement, i.e., line 2. With mock classes enabled, Dsc generates 5 test cases and 4 mock classes with the required annotation, along with values for the second parameter failOnMiss, and covers all lines. Pex+Moles does not reason about annotations (which are called attributes in C#) and therefore cannot cover the bulk of this method.

```
public @interface Serializable {
  Class serializer() default FieldSerializer.class;
  short id() default 0;
}
```

Listing 14: The @Serializable annotation type used in Listing 13.

5.2. Higher Coverage and Fewer Dangerous Code Executions Than Randoop (RQ4)

Table VII summarizes our comparison with the random test case generator Randoop by Pacheco et al. [2007]. For this experiment we used the same subjects as in our Pex comparison in Table VI, with the important difference that the Table VI subjects were strongly simplified versions of the Table VII subjects.

The results of Randoop are affected significantly by the way a test engineer uses Randoop for test case generation. The first option is running Randoop only on those classes that contain the methods under test. However, since a project typically contains many more classes that may call the methods under test, another option is to run Randoop on all classes of the project. As our goal was to be conservative and give Randoop a higher chance of covering the methods under test, we included these additional classes, to allow such calls from the additional classes to cover the methods under test. We also increased the runtime, starting with a long runtime and then decreasing the runtime as long as the test coverage remained constant, yielding a runtime of 10 minutes each. Table VII contains the results of these experiments.

Even though Randoop ran only for up to 10 minutes, in the search for suitable instances Randoop randomly called many methods that changed the state of our system. That is, Randoop created several UI elements and placed them on our screen. Randoop also called methods that wrote into our file system, which, for example, created 259 new top-level directories.

We also ran this experiment using the other Randoop option and ran Randoop only on those classes that contained the methods under test, using the same 60 second timeout as in Table VI. This experiment yielded a total of 7,410 randomly generated test cases. We did not include the results of this experiment in Table VII, as Randoop achieved the same coverage as shown in Table VII, but without producing any side-effects. Our interpretation of these results is that for some subjects, running Randoop on the subject classes only is advantageous over running Randoop on all project classes, as running on additional project classes does not necessarily increase the coverage of the subject methods.

In both options Randoop covered significantly smaller portions of code than our approach, even though our approach finished in around 10 seconds, allowing close to interactive test case generation. Our approach also did not have to resort to calling random methods and thereby avoided the unwanted side-effects of these methods.

6. RELATED WORK

In this section we briefly review representative pieces of related work from both research and development practice.

Table VII: Dsc with mock class generation (Dsc+Mock) vs. basic Dsc and Randoop on real-world code samples. These code samples are not strongly simplified as their counterparts in Table VI and we observe several side-effects (E) when analyzing them with a random test case generator such as Randoop. The dynamic symbolic approaches Dsc and Dsc+Mock are more directed, do not call random methods, and did thereby not exhibit any side-effects. For Randoop we observed side-effects on both the UI (dialog boxes appearing on the screen) as well as writing new files and directories into the file system (FS). Dir is the number of new directories created, which we omitted for Dsc and Dsc+Mock as the values were all zero. As in Table VI, Dsc+Mock covered the highest percentage of code lines (%L) and code branches (%B). M, L, B are the number of subject methods, their lines of code, and their number of branches; t[s]is the runtime in seconds and T is the number of test cases generated by the respective tool. Randoop achieved a slightly higher code coverage than basic Dsc but a significantly lower code coverage than Dsc+Mock.

Subject				Randoop						Dsc					Dsc+Mock			
	M	L	B	T	t[s]	%L	%B	E	Dir	T	t[s]	%L	%B	E	T	t[s]	%L	%B	E
Listing 7	1	7	4	1	604	14	25	0	0	1	0.4	14	25	0	4	1.4	100	100	0
ActiveMQ	4	60	26	13,850	481	13	15	0	0	4	2.2	13	15	0	14	29.2	60	42	0
ASM	2	18	6	14,359	609	33	50	0	0	2	1.0	33	50	0	6	4.0	100	100	0
BCEL	2	40	19	14,568	610	28	42	0	0	2	1.6	20	32	0	24	12.4	82	84	0
Drools	3	62	20	14,774	488	16	25	0	0	3	2.0	16	25	0	20	11.0	74	70	0
Guice	2	20	7	9,615	607	45	43	0	0	2	1.5	30	29	0	10	14.8	90	86	0
JaxLib	5	68	24	1,707	119	38	42	FS	35	5	3.0	32	42	0	30	20.0	88	92	0
JFreeChart	4	56	16	15,882	611	21	25	0	0	4	2.0	21	25	0	16	12.2	86	75	0
JMonkeyEngine	2	17	8	8,801	607	47	50	UI+FS	42	2	1.0	41	38	0	4	2.7	82	75	0
Soot	6	110	30	6,306	455	25	40	FS	118	6	2.4	27	33	0	56	31.3	72	67	0
Xalan	3	48	23	13,673	600	17	26	FS	64	3	1.8	17	26	0	40	23.5	88	91	0
Total	34	506	183	113,536	5,791	25	33	UI+FS	259	34	18.9	23	30	0	224	162.5	79	75	0

As discussed throughout this article, closest to our work is the Pex system with its Moles extension [Tillmann and de Halleux 2008; de Halleux and Tillmann 2010]. Tillmann and Schulte [2006a] described earlier how their system can generate a limited form of mock classes. When a method under test has a parameter of abstract type then their system generates a mock class subtype. However, their system cannot handle complex type constraints such as multiple-inheritance constraints, annotations, or reflection. See Section 5.1 for an empirical comparison. Since the dynamic symbolic execution system (Dsc) on which our techniques are built is conceptually similar to Pex, it should be relatively easy to re-implement our techniques for Pex and Moles.

Our technique of generating mock class method bodies in Section 4.5 replaces some symbolic literals with symbolic variables. Recall that we want to track how values returned by generated methods are being used during execution. To achieve this goal, we replace the values returned by such methods in the symbolic evaluator with symbolic variables. This concept of replacing symbolic literals returned by certain methods with symbolic variables has been used in a different context, i.e., by Xiao et al. [2011], to track if values returned by native methods cause the Pex dynamic symbolic execution engine to not cover parts of the code. The paper by Xiao et al. [2011] uses this information to direct developer attention to such less covered parts of the code, to facilitate interactive solving of such cases.

An alternative to dynamic symbolic execution is bounded exhaustive test case generation as pioneered by the Korat tool [Boyapati et al. 2002]. Similar to dynamic symbolic execution, Korat monitors how the code executes generated input values. Korat systematically generates new variants of the input value based on which value fields are accessed by the code during execution. It would be interesting to extend bounded exhaustive test case generation with techniques for generating mock classes, to allow tools such as Korat to cover code that requires classes that do not yet exist.

In the following we discuss the use of mock classes in other related work. We group the approaches by the following three key data sources from which the respective techniques infer mock classes. (1) The first group generates mock classes from a specification, such as a programmer-provided design-by-contract specification. (2) The second group infers mock classes from static analysis or static symbolic program execution. (3) The final group of approaches infers mock classes from existing test executions.

6.1. Mock Classes Inferred from Programmer-Written Specifications

SynthiaMock by Galler et al. [2010] generates mock classes that satisfy a given design-by-contract specification. SynthiaMock is a black-box technique as it derives mock classes not from code internals but from external design-by-contract specifications. If the method under test has a parameter of reference type, SynthiaMock generates for the parameter type a mock subclass, instantiates it, and sets its fields to values that satisfy the design-by-contract preconditions of the method under test. Each call to a mock class method triggers a constraint solver call to synthesize a value that satisfies the called method's design-by-contract postconditions. In contrast to SynthiaMock, dynamic symbolic mock classes do not require design-by-contract specifications. SynthiaMock is orthogonal to dynamic symbolic mock classes as dynamic symbolic mock classes could use SynthiaMock to reason about and satisfy existing design-by-contract specifications.

In recent years the software development community has created several practical mock frameworks including EasyMock[12], jMock[13], and Mockito[14] for Java pro-

[12]Available at http://www.easymock.org
[13]Available at http://www.jmock.org
[14]Available at http://www.mockito.org

grams, NMock[15] for .Net programs, and the Google C++ Mocking Framework[16] Google Mock [Freeman et al. 2004b; 2004a; Freeman and Pryce 2006]. The main focus of these frameworks is to generate mock classes according to a user-provided specification. These frameworks are similar in the features they offer and have in common that they aim to be user-friendly by allowing the programmer to write down specifications in the form of method calls. That means the programmer does not have to learn a formal specification language (beyond learning the API of the respective mock framework).

We now illustrate how such a mock framework is typically used, by describing how a programmer may test our C.foo example method of Listing 7 with the popular EasyMock mock framework. To explore the true branch originating from line 7 in Listing 7 using EasyMock we have (manually) written the test case shown in Listing 15. In line 1 we introduce interface K, which extends both of the interfaces I and J. Line 5 creates a mock object for interface K. We then set the expected method calls on the mock object and the return values expected from these method calls in lines 6 and 7. (The verify method called in line 10 checks if the testee has met the expectations we set in lines 6 and 7.)

```
public interface K extends I, J {}                                    1
public class Test {                                                   2
  public void testFooIsInstanceOfJ()                                  3
  {                                                                   4
    K mock = createMock(K.class);                                     5
    expect(mock.m1()).andReturn(0);                                   6
    expect(mock.m2()).andReturn(0);                                   7
    replay(mock);                                                     8
    C.foo(mock);                                                      9
    verify(mock);                                                     10
  }                                                                   11
}                                                                     12
```

Listing 15: A unit test case we wrote manually using the EasyMock framework. This test case triggers the same program execution path as the automatically inferred test case shown in Listing 8.

While this test case triggers the program path we had in mind, we had to devise and write down this new interface and test case manually. More user intervention will be required to cover other parts of the C.foo method under test. For example, to cover the true branch originating at Listing 7 line 10, we need to devise and write down a new type, one that implements both interfaces I and J and has an annotation of type A. In contrast, with our mock class generation technique, Dsc automatically generates appropriate mock classes and test cases that cover these execution paths.

6.2. Mock Classes Inferred by Static Analysis

At a high level, static analyses provide a different trade-off for reasoning about testee code. Given the undecidability of the halting problem, any program analysis—be it static, dynamic, or a hybrid—by definition suffers from at least one of the following fundamental program analysis problems.

(1) Incomplete coverage: An analysis may miss some program behaviors and may therefore not fully cover the analyzed code.

[15]Available at http://www.nmock.org
[16]Available at http://code.google.com/p/googlemock

(2) Over-approximating the code's behavior: An analysis may attribute to the code behaviors that in fact cannot occur. The latter leads to the analysis generating false warnings or spurious test cases.
(3) Non-termination: For some inputs an analysis may not terminate.

Practical program analysis tools typically avoid non-termination (problem 3). If for a given input an analysis tool cannot produce a perfect answer within a reasonable amount of time, the tool typically times out such a run, returns whatever partial results may be available, and terminates. Such timeouts are common in symbolic analysis (e.g., during static analysis or during dynamic symbolic execution), for example in the form of constraint-solver timeouts or limits placed on the number of execution paths explored.

Incomplete coverage (problem 1) is typically associated with dynamic analyses such as ours, as a dynamic analysis only covers a subset of all execution behaviors. On the positive side, dynamic analyses such as ours do not suffer from over-approximation (problem 2).

Over-approximating the code's execution behavior (problem 2) is typically associated with static analyses, as static analyses approximate execution semantics to remain feasible. These approximations are often most pronounced for advanced language features such as native code, reflection, and aliasing, which leads to static analyses having many imprecisions and producing many false warnings [Hind 2001; Zitser et al. 2004; Wagner et al. 2005; Bessey et al. 2010; Brauer and Simon 2012]).

Besides typically suffering from over-approximation (problem 2), in practice static analyses sometimes also suffer from incomplete coverage (problem 1). That is, static analyses are sometimes neither sound nor complete [Flanagan et al. 2002]. We illustrate this phenomenon on a well-known static analysis, ESC/Java by Flanagan et al. [2002]. When searching for a certain kind of problem in the code, ESC/Java may (1) fail to find some of the problems that are in the code and (2) may produce false warnings, e.g., warn about a runtime exception that can never occur in the code under any circumstances. There are several potential explanations of this phenomenon, including ESC/Java's limited reasoning about complex constraint types, its inability to reason about external code such as native code, and its limited handling of loops.

ESC/Java performs a relatively deep static analysis, by calculating the weakest precondition of each testee method. By default, without user-provided specifications, ESC/Java searches for potential runtime exceptions such as division-by-zero exceptions. This feature is valuable as such runtime exceptions often reveal programming errors or bugs. ESC/Java2, an extension of ESC/Java, can reason about the type constraints that are imposed by the dynamic type checks that are triggered by the Java instanceof operator and by type casts [Cok and Kiniry 2004; Kiniry 2004].

For example, when analyzing the Testee.foo method in Listing 16, ESC/Java2 suffers from both incompleteness and unsoundness and both misses a legitimate warning and produces a false warning. Both problematic locations are guarded by the if-condition of line 12. This condition can be satisfied with an instance of a sub-class of C that implements interface I.

ESC/Java2 produces a false warning for line 16 about a potential division-by-zero exception. This warning is a false warning because the native method Testee.bar, implemented in the C code of Listing 17, never returns zero. The reason behind this false warning is that ESC/Java cannot reason about external code and thus assumes that the code may return any value allowed by the method's return type (int), which includes the value zero. For exposition we kept the native example code in Listing 17 extremely simple. However, in practice, such native code is typically only available in binary form, for example, as a dynamically linked library (dll) and thus it is very hard

to reason about such code statically. A dynamic technique such as ours does not suffer from such false warnings, as it only produces a warning about an execution path after executing it.

```
public class C { /* .. */ }                                                    1
public interface I { /* .. */ }                                                2
public class Testee {                                                          3
  public static native int bar(int x);                                        4
                                                                               5
  static {                                                                     6
    System.loadLibrary ("testee");                                             7
  }                                                                            8
                                                                               9
  public static int foo(final C c, int n, int d) {                             10
    int count = -1;                                                            11
    if (c instanceof I) {                                                      12
      count = 0;                                                               13
      for (int i=1; i<=n && i<=100; i++) {                                     14
        if (i < 10)                                                            15
          count += i/Testee.bar(d);                                            16
        else                                                                   17
          count += i/d;                                                        18
      }                                                                        19
    }                                                                          20
    return count;                                                              21
  }                                                                            22
}                                                                              23
```

Listing 16: An example in which ESC/Java2 fails to produce valid warnings.

As an example of a missed warning, ESC/Java2 does not generate a warning for line 18. But this line may produce a division-by-zero exception when the method is invoked with a parameter value of d==0. ESC/Java2 misses this problem because it only considers scenarios in which the loop body is executed less than a configurable but fixed number of times. That is, by default ESC/Java2 only considers cases in which the loop body is either not executed at all or only once. However this problem is only found when executing the loop body at least 10 times.

```
int bar (int x) {                                                              1
  return (x==0) ? 1 : x;                                                       2
}                                                                              3
```

Listing 17: C-language version of the native method declared at line 4 in Listing 16.

Our (dynamic) technique of test case generation with mock-classes finds the division-by-zero exception at line 18 of Listing 16 and does not produce any (false) warning for line 16. On a side note, existing dynamic test case generators such as Pex also do not cover and therefore cannot detect the exception at line 18 because, as we discussed before, these earlier dynamic tools cannot reason about the type constraint at line 12.

In earlier work on Check 'n' Crash and DSD-Crasher [Csallner and Smaragdakis 2005; Csallner et al. 2008] we combined the static checking of ESC/Java with a dynamic test case generator. I.e., the dynamic analysis acts as a filter of the output of the static analysis. On the positive side, in such a static-dynamic analysis hybrid, the dynamic analysis acts as a filter of the results produced by the static analysis and can thereby remove the false warnings produced by ESC/Java. On the other hand, the dynamic analysis is not a precise filter and, besides filtering out many false warnings it also filters out true warnings. That means that such a static-dynamic hybrid suffers from problem (1) even more than the underlying static analysis ESC/Java.

As an example, Check 'n' Crash and DSD-Crasher cannot detect the problem in line 18 of Listing 16, as the underlying static analysis does not report it either. In addition, for code not shown in the example, Check 'n' Crash may filter out many true warnings that ESC/Java may produce.

Tamiflex by Bodden et al. [2010; Bodden et al. [2011] provides a way for static analysis to reason about reflection. That is, Tamiflex collects execution traces and performs static analysis on those collected traces. However, this limits the scope of the static analysis to the observed executions, which essentially turns the static analysis into a form of dynamic analysis. That means that an analysis in the style of Tamiflex is no longer complete in the sense that it no longer reasons about all possible execution behaviors.

6.3. Mock Classes Inferred from Test Executions: Test Carving, Capture & Replay

Saff and Ernst [2004] and Saff et al. [2005] generate mock classes during test factoring. The goal of test factoring is to simplify an existing long-running system test, by replacing a call to a long-running component with a call to an automatically generated mock class. The generated mock class then simulates the earlier recorded behavior of the long-running component. Whereas test factoring takes as input an existing test suite, dynamic symbolic mock classes are used in test case generation and do not require an existing test suite.

The paper by Thummalapenta et al. [2010] applies the trace capturing technique of Saff et al. [2005] to infer from existing traces parameterized unit test cases (PUTs). That is, for a method call in the execution trace, the technique can generate a parameterized unit test that invokes the method on parameters supplied by a dynamic symbolic execution engine. While related, this technique is orthogonal to our work and could be integrated into our approach in future work.

GenUTest by Pasternak et al. [2009] generates mock aspects to capture and replay how the code under test interacts with external components. Interactions that can be captured and replayed include the code under test calling a method of an external component or the code under test accessing a field of such a component. To replay such captured interactions, GenUTest generates mock aspects, which use aspect-oriented programming techniques [Kiczales et al. 1997] to intercept and redirect new interaction attempts between the code under test and these external components.

While GenUTest uses mock aspects for capture and replay, we could also use mock aspects as an alternative implementation technique for a test class and mock code generator such as our dynamic symbolic mock classes. That is, once we have collected and manipulated a path condition and found a satisfying solution, we could map that satisfying solution to a test case plus mock aspect combination (instead of a test case plus mock class combination). This may be useful in an environment that prefers generated aspects over generated classes.

7. CONCLUSIONS

We surveyed 35 third-party open-source applications and found hundreds of branches that branch on multiple incompatible super-types, reflection, and annotations, which current test case generation techniques do not cover. We described a technique that can cover such code, implemented it for Java, and showed that it can cover real code that existing techniques cannot cover.

Our implementation and instructions on how to replicate our experiments are freely available via our web site (http://cseweb.uta.edu/~mainul/MockClass/).

REFERENCES

Saswat Anand, Alessandro Orso, and Mary Jean Harrold. 2007. Type-Dependence Analysis and Program Transformation for Symbolic Execution. In *Proc. 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 117–133.

Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. 2004. The Spec# programming system: An overview. In *Proc. International Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*. Springer, 49–69.

Kent Beck and Erich Gamma. 1998. Test infected: Programmers love writing tests. *Java Report* 3, 7 (July 1998), 37–50.

Boris Beizer. 1990. *Software testing techniques* (second ed.). Van Nostrand Reinhold.

Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM (CACM)* 53, 2 (Feb. 2010), 66–75.

Cédric Beust and Hani Suleiman. 2007. *Next Generation Java Testing: TestNG and Advanced Concepts*. Addison-Wesley.

Eric Bodden, Andreas Sewe, Jan Sinschek, and Mira Mezini. 2010. *Taming Reflection (Extended version)*. Technical Report TUD-CS-2010-0066. TU Darmstadt. http://www.bodden.de/pubs/tr-tamiflex.pdf

Eric Bodden, Andreas Sewe, Jan Sinschek, and Mira Mezini. 2011. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *Proc. 33rd ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, 241–250.

Grady Booch, James Rumbaugh, and Ivar Jacobson. 2005. *The Unified Modeling Language User Guide* (second ed.). Addison-Wesley.

Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated testing based on Java predicates. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 123–133. DOI:http://dx.doi.org/10.1145/566172.566191

Jörg Brauer and Axel Simon. 2012. Inferring Definite Counterexamples through Under-Approximation. In *Proc. 4th International NASA Formal Methods Symposium (NFM)*. Springer, 54–69.

Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 209–224.

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically generating inputs of death. In *Proc. 13th ACM Conference on Computer and Communications Security (CCS)*. ACM, 322–335. DOI:http://dx.doi.org/10.1145/1180405.1180445

Satish Chandra, Stephen J. Fink, and Manu Sridharan. 2009. Snugglebug: A powerful approach to weakest preconditions. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 363–374.

Lori A. Clarke. 1976. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering (TSE)* 2, 3 (1976), 215–222.

David R. Cok and Joseph R. Kiniry. 2004. *ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2*. Technical Report NIII-R0413. Nijmegen Institute for Computing and Information Science.

Christian Collberg, Ginger Myles, and Michael Stepp. 2007. An empirical study of Java bytecode programs. *Software—Practice & Experience (SPE)* 37, 6 (May 2007), 581–641.

Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: An automatic robustness tester for Java. *Software—Practice & Experience (SPE)* 34, 11 (Sept. 2004), 1025–1050.

Christoph Csallner and Yannis Smaragdakis. 2005. Check 'n' Crash: Combining static checking and testing. In *Proc. 27th ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, 422–431.

Christoph Csallner, Yannis Smaragdakis, and Tao Xie. 2008. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 17, 2 (April 2008), 1–37.

Jonathan de Halleux and Nikolai Tillmann. 2010. Moles: Tool-Assisted Environment Isolation with Closures. In *Proc. 48th International Conference on Objects, Models, Components, Patterns (TOOLS)*. Springer, 253–270.

Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering* 10, 4 (Oct. 2005), 405–435.

Bassem Elkarablieh, Patrice Godefroid, and Michael Y. Levin. 2009. Precise pointer reasoning for dynamic test generation. In *Proc. 18th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 129–140.

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended static checking for Java. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 234–245.

Steve Freeman, Tim Mackinnon, Nat Pryce, and Joe Walnes. 2004a. jMock: Supporting responsibility-based design with mock objects. In *Companion to the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 4–5.

Steve Freeman, Tim Mackinnon, Nat Pryce, and Joe Walnes. 2004b. Mock roles, not objects. In *Companion to the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 236–246.

Steve Freeman and Nat Pryce. 2006. Evolving an embedded domain-specific language in Java. In *Companion to the 21th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 855–865.

Stefan J. Galler, Martin Weiglhofer, and Franz Wotawa. 2010. Synthesize It: From Design by Contract to Meaningful Test Input Data. In *Proc. 8th IEEE International Conference on Software Engineering and Formal Methods (SEFM)*. IEEE, 286–295.

Patrice Godefroid. 2007. Compositional dynamic test generation. In *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 47–54.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 213–223. DOI:http://dx.doi.org/10.1145/1065010.1065036

Patrice Godefroid, Michael Y Levin, and David Molnar. 2008. Automated whitebox fuzz testing. In *Proc. Network and Distributed System Security Symposium (NDSS)*. The Internet Society.

James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. 2005. *The Java Language Specification* (third ed.). Prentice Hall.

Mark Hapner, Rich Burridge, Rahul Sharma, and Joseph Fialli. 2002. *Java Message Service: Version 1.1*. Sun Microsystems.

Michael Hind. 2001. Pointer analysis: Haven't we solved this problem yet?. In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. ACM, 54–61.

Mainul Islam and Christoph Csallner. 2010. Dsc+Mock: A test case + mock class generator in support of coding against interfaces. In *Proc. 8th International Workshop on Dynamic Analysis (WODA)*. ACM, 26–31.

Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K. Chang. 2010. OCAT: Object capture-based automated testing. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 159–170.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-Oriented Programming. In *Proc. 11th European Conference on Object-Oriented Programming (ECOOP)*. 220–242.

James C. King. 1976. Symbolic execution and program testing. *Communications of the ACM (CACM)* 19, 7 (1976), 385–394.

Joseph R. Kiniry. 2004. The Logics and Calculi of ESC/Java2.

Mika Mäntylä, Juha Itkonen, and Joonas Iivonen. 2012. Who tested my software? Testing as an organizationally cross-cutting activity. *Software Quality Journal* 20, 1 (March 2012), 145–172.

Madhuri R. Marri, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. An Empirical Study of Testing File-System-Dependent Software with Mock Objects. In *Proc. 4th International Workshop on Automation of Software Test (AST)*. IEEE, 149–153.

Steve McConnell. 2004. *Code Complete* (second ed.). Microsoft Press.

Gerard Meszaros. 2007. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley.

Audris Mockus, Roy T. Fielding, and James D. Herbsleb. 2002. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11, 3 (July 2002), 309–346. DOI:http://dx.doi.org/10.1145/567793.567795

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proc. 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 337–340. DOI:http://dx.doi.org/content/60hx121083823548

National Institute of Standards and Technology (NIST). 2002. *The economic impacts of inadequate infrastructure for software testing: Planning report 02-3*.

Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *Proc. 29th ACM/IEEE International Conference on Software Engineering (ICSE)*. IEEE, 75–84.

Benny Pasternak, Shmuel S. Tyszberowicz, and Amiram Yehudai. 2009. GenUTest: A unit test and mock aspect generation tool. *International Journal on Software Tools for Technology Transfer (STTT)* 11, 4 (Oct. 2009), 273–290.

Mauro Pezzè and Michal Young. 2007. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley.

David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. 2005. Automatic test factoring for Java. In *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 114–123.

David Saff and Michael D. Ernst. 2004. Mock object creation for test factoring. In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. ACM, 49–51.

Suresh Thummalapenta, Jonathan De Halleux, Nikolai Tillmann, and Scott Wadsworth. 2010. DyGen: Automatic generation of high-coverage tests via mining Gigabytes of dynamic traces. In *Proc. 4th International Conference on Tests and Proofs (TAP)*. Springer, 77–93.

Nikolai Tillmann and Jonathan de Halleux. 2008. Pex - White Box Test Generation for .Net. In *Proc. 2nd International Conference on Tests And Proofs (TAP)*. Springer, 134–153.

Nikolai Tillmann and Wolfram Schulte. 2006a. Mock-object generation with behavior. In *Proc. 21st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 365–368.

Nikolai Tillmann and Wolfram Schulte. 2006b. Unit Tests Reloaded: Parameterized Unit Testing with Symbolic Execution. *IEEE Software* 23, 4 (July 2006), 38–47.

Gina D. Venolia, Robert DeLine, and Thomas LaToza. 2005. *Software development at Microsoft observed*. Technical Report MSR-TR-2005-140. Microsoft Research.

Stefan Wagner, Jan Jürjens, Claudia Koller, and Peter Trischberger. 2005. Comparing Bug Finding Tools with Reviews and Tests. In *Proc. 17th IFIP TC6/WG 6.1 International Conference on Testing of Communicating Systems (TestCom)*. Springer, 40–55.

Laurie Williams, Gabe Brown, Adam Meltzer, and Nachiappan Nagappan. 2011. Scrum + Engineering Practices: Experiences of Three Microsoft Teams. In *Proc. 5th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 463–471.

Laurie Williams, Gunnar Kudrjavets, and Nachiappan Nagappan. 2009. On the Effectiveness of Unit Test Automation at Microsoft. In *Proc. 20th IEEE International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 81–89.

Xusheng Xiao, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. 2011. Precise identification of problems for structural test generation. In *Proc. 33rd ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, 611–620.

Tao Xie, Nikolai Tillmann, Peli de Halleux, and Wolfram Schulte. 2009. Fitness-Guided Path Exploration in Dynamic Symbolic Execution. In *Proc. International Conference on Dependable Systems and Networks (DSN)*. IEEE, 359–368.

Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. 2011. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering* 16, 3 (June 2011), 325–364.

Misha Zitser, Richard Lippmann, and Tim Leek. 2004. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proc. 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 97–106. DOI:http://dx.doi.org/10.1145/1029894.1029911