Automated Test Generation for Java Generics

Gordon Fraser[1] and Andrea Arcuri[2]

[1] University of Sheffield
Dep. of Computer Science, Sheffield, UK
`gordon.fraser@sheffield.ac.uk`,
[2] Simula Research Laboratory
P.O. Box 134, 1325 Lysaker, Norway
`arcuri@simula.no`

Abstract. Software testing research has resulted in effective white-box test generation techniques that can produce unit test suites achieving high code coverage. However, research prototypes usually only cover subsets of the basic programming language features, thus inhibiting practical use and evaluation. One feature commonly omitted are Java's *generics*, which have been present in the language since 2004. In Java, a generic class has type parameters and can be instantiated for different types; for example, a collection can be parameterized with the type of values it contains. To enable test generation tools to cover generics, two simple changes are required to existing approaches: First, the test generator needs to use Java's extended reflection API to retrieve the little information that remains after *type erasure*. Second, a simple static analysis can identify candidate classes for type parameters of generic classes. The presented techniques are implemented in the EvoSuite test data generation tool and their feasibility is demonstrated with an example.

Key words: automated test generation, unit testing, random testing, search-based testing, EvoSuite

1 Introduction

To support developers in the tedious task of writing and updating unit test suites, white-box testing techniques analyze program source code and automatically derive test cases targeting different criteria. These unit tests either exercise automated test oracles, for example by revealing unexpected exceptions, or help in satisfying a coverage criterion. A prerequisite for an effective unit test generator is that as many as possible language features of the target programming language are supported, otherwise the quality of the generated tests and the usefulness of the test generation tools will be limited.

A particular feature common to many modern programming languages such as Java are *generics* [12]: Generics make it possible to parameterize classes and methods with types, such that the class can be *instantiated* for different types. A common example are container classes (e.g., list, map, etc.), where generics can be used to specify the type of the values in the container. For example, in Java a

`List<String>` denotes a list in which the individual elements are strings. Based on this type information, any code using such a list will know that parameters to the list and values returned from the list are strings. While convenient for programmers, this feature is a serious obstacle for automated test generation.

In this short paper, we present a simple automated approach to generating unit tests for code using generics by statically determining candidate types for generic type variables. This approach can be applied in *random testing*, or in *search-based testing* when exploring type assignments as part of a search for a test suite that maximizes code coverage. We have implemented the approach in the EvoSuite test generation tool and demonstrate that it handles cases not covered by other popular test data generation tools. Furthermore, to the best of our knowledge, we are aware of no technique in the literature that targets Java generics.

2 Background

In this paper, we address the problem of Java generics in automated test generation. To this purpose, this section presents the necessary background information on generics and test generation, and illustrates why generics are problematic for automated test generation.

2.1 Java Generics

In Java, generics parameterize classes, interfaces, and methods with type parameters, such that the same code can be instantiated with different types. This improves code reusability, and it helps finding type errors statically.

A generic class has one or more type parameters, similar to formal parameters of methods. When instantiating a generic class, one specifies concrete values for these type parameters. For example, consider the following simplistic implementation of a stack datastructure:

```
public class Stack<T> {
  private T[] data = new T[100];
  private int pos = 0;

  public T pop() {
    return data[pos--];
  }

  public void push(T value) {
    data[pos++] = value;
  }
}
```

The class `Stack` has one *type parameter*, `T`. Within the definition of class `Stack`, `T` can be used as if it were a concrete type. For example, `data` is defined as an array of type T, pop returns a value of type T, and push accepts a parameter of type T. When instantiating a Stack, a concrete value is assigned to T:

```
Stack<String> stringStack = new Stack<String>();
stringStack.push("Foo");
stringStack.push("Bar");
String value = stringStack.pop(); // value == "Bar"

Stack<Integer> intStack = new Stack<Integer>();
intStack.push(0);
Integer intValue = intStack.pop();
```

Thanks to generics, the Stack can be instantiated with any type, e.g., String and Integer in this example. The same generic class is thus reused, and the compiler can statically check whether the values that are passed into a Stack and returned from a Stack are of the correct type.

It is possible to put constraints on the type parameters. For example, consider the following generic interface:

```
public abstract class Foo<T extends Number> {
  private T value;

  public void set(T value) {
    this.value = value;
  }

  public double get() {
     return value.doubleValue();
  }
}
```

This class can only be instantiated with types that are subclasses of Number. Thus, Foo<Integer> is a valid instantiation, whereas Foo<String> is not. A further way to restrict types is the super operator. For example, Foo<T super Bar> would restrict type variable T to classes convertible to Bar or its superclasses.

Note also that Foo is an abstract class. When creating a subclass or when instantiating a generic interface, the type parameter can be concretized, or can be assigned a new type parameter of the subclass. For example:

```
public class Bar extends Foo<Integer> {
  // ...
}

public class Zoo<U extends Number, V> extends Foo<U> {
  // ...
}
```

The class `Bar` sets the value of `T` in `Foo` to `Integer`, whereas `Zoo` delays the instantiation of `T` by creating a new type variable `U`. This means that inheritance can be used to strengthen constraints on type variables. Class `Zoo` also demonstrates that a class can have any number of type parameters, in this case two.

Sometimes it is not known statically what the concrete type for a type variable is, and sometimes it is irrelevant. In these cases, the *wildcard type* can be used. For example, if we have different methods returning instances of `Foo` and we do not care what the concrete type is as we are only going to use method `get` which works independently of the concrete type, then we can declare this in Java code as follows:

```
Foo<?> foo = ... // some method returning a Foo
double value = foo.get();
```

Generics are a feature that offers type information for static checks in the compiler. However, this type information is not preserved by the compiler. That is, at runtime, given a concrete instance of a `Foo` object, it is not possible to know the value of `T` (the best one can do is guess by checking `value`). This is known as *type erasure*, and is problematic for dynamic analysis tools.

The Java compiler also accepts generic classes without any instantiated type parameters. This is what most dynamic analysis and test generation tools use, and it essentially amounts to assuming that every type variable represents `Object`.

```
Stack stack = new Stack();
stack.push("test");
Object o = stack.pop();
```

As indicated with the waved underline, a compiler will issue a warning in such a case, as static type checking for generic types is impossible this way.

Besides classes, it is also possible to parameterize methods using generics. A generic method has a type parameter which is inferred from the values passed as parameters. For example, consider the following generic method:

```
public class Foo {
  public <T> List<T> getNList(T element, int length) {
    List<T> list = new ArrayList<T>();
    for(int i = 0; i < length; i++)
      list.add(element);
    return list;
  }
}
```

The method `getNList` creates a generic list of length `length` with all elements equal to `element`. `List` is a generic class of the Java standard library, and the generic parameter of the list is inferred from the parameter `element`. For example:

```
Foo foo = new Foo();
List<String> stringList = foo.getNList("test", 10);
List<Integer> intList = foo.getNList(0, 10);
```

In this example, the same generic method is used to generate a list of strings and a list of numbers.

2.2 Automated Test Generation

Testing is a common method applied to ensure that software behaves as desired. Developer testing involves writing test cases that exercise a program through its application programmer interface (API). This has been particularly popularized by the availability of convenient test automation frameworks for unit testing, such as JUnit. Test cases may be written before the actual implementation such that they serve as specification (test-driven development), they can be written while explicitly testing a program in order to find bugs, or they can be written in order to capture the software behavior in order to protect against future (regression) bugs. However, writing test cases can be a difficult and error-prone task, and manually written suites of tests are rarely complete in any sense. Therefore, researchers are investigating the task of automating test generation, such that developer-written tests can be supplemented by automatically generated tests.

Technically, the task of automatically generating unit tests consists of two parts: First, there is the task of generating suitable test inputs, i.e., individual sequences of calls that collectively exercise the class under test (CUT) in a comprehensive way. Second, there is the task of determining whether these tests find any bugs, i.e., the generated test cases require test oracles.

Automatically generated test cases are particularly useful when there are *automated* oracles, such that finding faults becomes a completely automated task. Fully automated oracles are typically encoded in code contracts or assertions, and in absence of such partial specifications, the most basic oracle consists of checking whether the program crashes [11] (or in the case of a unit, whether an undeclared exception occurs).

The alternative to such automated oracles consists of adding oracles in terms of *test assertions* to the generated unit tests. In a scenario of regression testing where the objective is simply to capture the current behaviour, this process is fully automated. Alternatively, the developer is expected to manually add assertions to the generated unit tests. This process can further be supported by suggesting possible assertions [6], such that the developer just needs to confirm whether the observed behaviour is as expected, or erroneous.

Popular approaches to automated unit test generation include *random testing* [11], *dynamic symbolic execution* [7], and *search-based testing* [10]. In random testing, sequences of random calls are generated up to a given limit. A simple algorithm to generate a random test for an object oriented class is to randomly select methods of the class to add, and for each parameter to randomly select previously defined assignable objects in the test, or to instantiate new objects of the required type by calling one of its constructors or factory methods (chosen randomly). Given that random tests can be very difficult to comprehend, the main application of such approaches lies in exercising automated oracles. Dynamic symbolic execution systematically tries to generate inputs that cover all paths for a given entry function by applying constraint solvers on path conditions. Consequently, entry functions need to be provided, for example in terms of parameterized unit tests [13]. Finally, search-based testing has the flexibility of being suitable for almost any testing problem, and generating unit tests is an area where search-based testing has been particularly successful.

In search-based testing [1,10], the problem of test data generation is cast as a search problem, and search algorithms such as hillclimbing or genetic algorithms are used to derive test data. The search is driven by a fitness function, which is a heuristic that estimates how close a candidate solution is to the optimum. When the objective is to maximize code coverage, then the fitness function does not only quantify the coverage of a given test suite, but it also provides guidance towards improving this coverage. To calculate the fitness value, test cases are usually executed using instrumentation to collect data. Guided by the fitness function, the search algorithm iteratively produces better solutions until either an optimal solution is found, or a stopping condition (e.g., timeout) holds.

The use of search algorithms to automate software engineering tasks has been receiving a tremendous amount of attention in the literature [8], as they are well suited to address complex, non-linear problems.

2.3 The Woes of Generics in Automated Testing

So why are generics a problem for test generation? Java bytecode has no information at all about generic types — anything that was generic on the source code is converted to type `Object` in Java bytecode. However, many modern software analysis and testing tools work with Java bytecode rather than sourcecode. Some information can be salvaged using Java *reflection*: It is possible to get the exact signature of a method. For example, in the following snippet we can determine using Java reflection that the parameter of method `bar` is a list of strings:

```java
public class Foo {
  public void bar(List<String> stringList) {
    // ...
  }
}
```

However, consider the following variation of this example:

```java
public class Foo<T> {
  public void bar(List<T> stringList) {
    // ...
  }
}
```

If we now query the signature of method `bar` using Java reflection, we learn that the method expects a list of `T`. But what is `T`? Thanks to type erasure, for a given object, it is impossible to know for a given instance of `Foo` what the type of `T` is[1].

Our only hope is if we know how `Foo` was generated. For example, as part of the test generation we might instantiate `Foo` ourselves — yet, when doing so, what should we choose as concrete type for type variable `T`? We did not specify a type boundary for `T` in the example, and the implicit default boundary is `Object`. Consequently, we have to choose a concrete value for type `T` out of the set of all classes that are assignable to `Object`. In this example, this means *all* classes on the classpath are candidate values for `T`, and typically there are *many* classes on the classpath.

However, things get worse: It is all too common that methods are declared to return objects or take parameters of generic classes where the type parameters are given with a wildcard type. Even worse, legacy or sloppily written code may even completely omit type parameters in signatures. A wildcard type or an omitted type parameter looks like `Object` when we inspect it with Java reflection. So if we have a method that expects a `List<?>`, what type of list should we pass as parameter? If we have received a `List<?>`, all we know is that if we get a value out of it, it will be an `Object`. In such situations, if we do not guess the right type, then likely we end up producing useless tests that end up in `ClassCastException`s and cover no useful code. To illustrate this predicament, consider the following snippet of code:

```java
@Test
public void testTyping(){
  List<String> listString = new LinkedList<String>();
  listString.add("This is a list for String objects");
  listString.add("Following commented line would not
      compile");
  //List<Integer> listStringButIntegerType = listString;
  List erasedType = listString;
  List<Integer> listStringButIntegerType = erasedType;
  listStringButIntegerType.get(0).intValue();
}
```

If we define a parametrized list as to contain only String objects, then it is not possible to assign it to a list for integers; the compiler will not allow it. But, if we first assign it to a generic list, then this generic list can be assigned to an integer list without any compilation error. In other words, a reference to an integer list does not give any guarantee that it will contain only integers. This is a particular serious problem for automated test data generation because, if generics are not properly handled, we would just end up in test cases that are full of uninteresting `ClassCastExceptions`. For example, if a method of the CUT takes as input a list of Strings, then it would be perfectly legit (i.e., it will compile) to give as input a generic list that rather contains integers.

[1] Except if we know the implementation of `Foo` and `List` such that we can use reflection to dig into the low level details of the list member of `Foo` to find out what the type of the internal array is.

Considering all this, it is not surprising that research tools on automated test generation have steered clear of handling generics so far.

3 Generating Tests for Generic Classes

The problem of generics becomes relevant whenever a test generation algorithm attempts to instantiate a new object, or to satisfy a parameter for a newly inserted method call. For example, this can be the case of a random test generation algorithm exploring sequences of calls, but it can just as well be part of a genetic algorithm evolving test suites. Assume that our test generation algorithm decides to add a call to the method `bar` defined as follows:

```
public class Foo<T> {
  public void bar(T baz) {
    // ...
  }
}
```

The signature of `bar` does not reveal what the exact type of the parameter should be. In fact, given an object of type `Foo`, when we query the method parameters of method `bar` using Java's standard reflection API reveals that `baz` is of type `Object`! Fortunately, the reflection API was extended starting in Java version 1.5, and the extended API adds for each original method a variant that returns generic type information. For example, whereas the standard way to access the parameters of a `java.lang.reflect.Method` object is via method `getParameters`, there is also a generic variant `getGenericParameters`. However, this method only informs us that the type of `baz` is `T`.

Consequently, the test generator needs to consider the concrete instance of `Foo` on which this method is called, in order to find out what `T` is. Assume the test generator decides to instantiate a new object of type `Foo`. At this point, it is necessary to decide on the precise type of the object, i.e., to instantiate the type parameter `T`. As discussed earlier, any class on the classpath is assignable to `Object`, so any class qualifies as candidate for `T`. As randomly choosing a type out of the entire set of available classes is not a good option (i.e., the probability of choosing an appropriate type would be extremely low), we need to restrict the set of candidate classes.

To find a good set of candidate classes for generic type parameters, we can exploit the behaviour of the Java compiler. The compiler removes the type information as part of type erasure, but if an object that is an instance of a type described by a type variable is used in the code, then before the use the compiler inserts a *cast* to the correct type. For example, if type variable `T` is expected to be a string, then there will be a method call on the object representing the string or it is passed as a string parameter to some other method. Consequently, by looking for casts in the bytecode we can identify which classes are relevant. Besides explicit casts, a related construct giving evidence of the concrete type is the `instanceof` operator in Java, which takes a type parameter that is preserved in the bytecode.

The candidate set is initialized with the default value `Object`. To collect the information about candidate types, we start with the dedicated class under test (CUT), and inspect the bytecode of each of its methods for `castclass` or `instanceof` operators. In addition to direct calls in the CUT, the parameters may be used in subsequent calls, therefore this analysis needs to be interprocedural. Along the analysis, we can also consider the precise method signatures, which may contain concretizations of generic type variables. However, the further away from the CUT the analysis goes, the less related the cast may be to covering the code in the CUT. Therefore, we also keep track of the depth of the call tree for each type added to the set of candidate types.

Now when instantiating a generic class `Foo`, we randomly choose values for its type parameters out of the set of candidate classes. The probability of a type being selected is dependent on the depth in the call tree, and in addition we need to determine the subset of the candidate classes that are compatible with the bounds of the type variable that is instantiated. Finally, it may happen that a generic class itself ends up in the candidate set. Thus, the process of instantiating generic type parameters is a recursive process, until all type parameters have received concrete values. To avoid unreasonably large recursions, we put an upper boundary on the number of recursive calls, and use a wildcard type if the boundary has been reached.

4 EvoSuite: A Unit Test Generator Supporting Generics

We have extended the EvoSuite unit test generation tool [3] with support for Java generics according to the discussed approach. EvoSuite uses a genetic algorithm (GA) to evolve test suites. The objective of the search inEvoSuite is to maximize code coverage, so the fitness function does not only quantify the coverage of a given test suite, but it also provides guidance towards improving this coverage. For example, in the case of branch coverage, the fitness function considers for each individual branching statement how close it was to evaluating to true and to false (i.e., its branch distance), and thus can guide the search towards covering both outcomes.

The GA has a population of candidate solutions, which are test suites, i.e., sets of test cases. Each test case in turn is a sequence of calls (like a JUnit test case). EvoSuite generates random test suites as initial population, and these test suites are evolved using search operators that mimic processes of natural evolution. The better the fitness value of an individual, the more likely it is considered for reproduction. Reproduction applies mutation and crossover, which modify test suites according to predefined operators. For example, crossover between two test suites creates two offspring test suites, each containing subsets from both parents. Mutation of test suites leads to insertion of new test cases, or change of existing test cases. When changing a test case, we can remove, change, or insert new statements into the sequence of statements. To create a new test case, we simply apply this statement insertion on an initially empty sequence until the test has a desired length. Generic classes need to be handled both, when generating the initial random population, and during the search, when test cases are mutated. For example, mutation involves adding and changing statements, both of which require that generic classes are properly handled. For details on these search operators we refer to [5].

To demonstrate the capabilities of the improved tool, we now show several simple examples on which test generation tools that do not support generics fail (in particular, we verified this on Randoop [11], Dsc [9], Symbolic PathFinder [2], and Pex [13]).

The first example shows the simple case where a method parameter specifies the exact signature. As the method accesses the strings in the list (by writing them to the standard output) outwitting the compiler by providing a list without type information is not sufficient to cover all the code – anything but an actual list of strings would lead to a `ClassCastException`. Thus, the task of the test generation tool is to produce an instance that exactly matches this signature:

```
import java.util.List;

public class GenericParameter {

  public boolean stringListInput(List<String> list){
    for(String s : list)
      System.out.println(s.toLowerCase());
    if(list.size() < 3)
      return false;
    else
      return true;
  }
}
```

For this class, EvoSuite produces the following test suite to cover all branches:

```
public class TestGenericParameter {
  @Test
  public void test0() throws Throwable {
      GenericParameter genericParameter0 = new GenericParameter();
      LinkedList<String> linkedList0 = new LinkedList<String>();
      linkedList0.add("");
      linkedList0.add("");
      linkedList0.add("");
      boolean boolean0 = genericParameter0.stringListInput(linkedList0);
      assertEquals(true, boolean0);
  }

  @Test
  public void test1() throws Throwable {
      GenericParameter genericParameter0 = new GenericParameter();
      LinkedList<String> linkedList0 = new LinkedList<String>();
      boolean boolean0 = genericParameter0.stringListInput(linkedList0);
      assertEquals(false, boolean0);
  }
}
```

As second example, consider a generic class that has different behavior based on what type it is initialized to, such that a test generator needs to find appropriate values for type parameter `T` in order to cover all branches:

```java
import java.util.List;

public class GenericsExample<T,K> {

  public int typedInput(T in){
    if(in instanceof String)
      return 0;
    else if(in instanceof Integer)
      return 1;
    else if(in instanceof java.net.ServerSocket)
      return 2;
    else
      return 3;
  }
}
```

Again EvoSuite is able to create a branch coverage test suite easily:

```java
public class TestGenericsExample {
  @Test
  public void test0() throws Throwable {
      GenericsExample<ServerSocket, ServerSocket> genericsExample0 = new
            GenericsExample<ServerSocket, ServerSocket>();
      ServerSocket serverSocket0 = new ServerSocket();
      int int0 = genericsExample0.typedInput(serverSocket0);
      assertEquals(2, int0);
  }

  @Test
  public void test1() throws Throwable {
      GenericsExample<Integer, Object> genericsExample0 = new
            GenericsExample<Integer, Object>();
      int int0 = genericsExample0.typedInput((Integer) 1031);
      assertEquals(1, int0);
  }

  @Test
  public void test2() throws Throwable {
      GenericsExample<String, ServerSocket> genericsExample0 = new
            GenericsExample<String, ServerSocket>();
      int int0 = genericsExample0.typedInput("");
      assertEquals(0, int0);
  }

  @Test
  public void test3() throws Throwable {
      GenericsExample<Object, Object> genericsExample0 = new
            GenericsExample<Object, Object>();
      Object object0 = new Object();
      int int0 = genericsExample0.typedInput(object0);
      assertEquals(3, int0);
  }
}
```

To evaluate the examples, we compare with other test generation tools for Java described in the literature. Research prototypes are not always freely available, hence we selected tools that are not only available online, but also popular (e.g., highly cited and used in different empirical studies). In the end, we selected Randoop [11], Dsc [9], Symbolic PathFinder [2], and Pex [13]).

Like for EvoSuite, setting up Randoop to generate test cases for `GenericsExample` is pretty straightforward. Already after a few seconds, it has generated JUnit classes with hundreds of test cases. However, Randoop generated no tests that used a list as input for the `stringListInput` method (0% coverage) or a ServerSocket for `typedInput`.

A popular alternative to search-based techniques in academia is dynamic symbolic execution (DSE). For Java, available DSE tools are Dsc [9] and Symbolic PathFinder [2]. However, these tools assume static entry functions (Dsc), or appropriate test drivers that take care of setting up object instances and selecting methods to test symbolically[2]. As choosing the “right” type for a `GenericsExample` object instantiation is actually part of the testing problem (e.g., consider `typedInput` method), then JPF does not seem to help in this case.

To overcome this issue, we also investigated Pex, probably the most popular DSE tool, but a tool that assumes C#. However, generics also exist in C#, so it is a good opportunity to demonstrate that the problem addressed in this paper is not a problem specific to Java.

We translated the `GenericsExample` class to make a comparison, resulting in the following C# code.

```
using System;
using System.Collections.Generic;

public class GenericsExample<T> {
  public int typedInput(T input){
    if (input is String)
      return 0;
    else if (input is Int32)
      return 1;
    else if (input is System.Collections.Stack)
      return 2;
    else
      return 3;
   }
}
```

Note that we replaced the TCP socket class with a `Stack` class such that the example can also be used with the web interface for Pex, PexForFun[3]. Like the other DSE tools, Pex assumes an entry function, which is typically given

[2] http://javapathfinder.sourceforge.net/, accessed June 2013.
[3] http://www.pexforfun.com/, accessed June 2013.

by a parameterized unit test. For example, the following code shows an entry function that can be used to explore the example code using DSE:

```
using Microsoft.Pex.Framework;

[PexClass]
public class TestClass {

 [PexMethod]
 public void Test<T>(T typedInput) {
   var ge = new GenericsExample<T>();
   ge.typedInput(typedInput);
 }
}
```

This test driver and the `GenericsExample` class can be used with Pex on Pex4Fun, and doing so reveals that Pex does not manage to instantiate `T` at all (it just attempts `null`).

5 Conclusions

Generics are an important feature in object-oriented programming languages like Java. However, they pose serious challenges for automated test case generation tools. In this short paper, we have presented a simple techniques to handle Java generics in the context of test data generation. Although we implemented those techniques as part of the EvoSuite tool, they could be used in any Java test generation tool.

We showed the feasibility of our approach on artificial examples. While EvoSuite was able to achieve 100% coverage in all these examples quickly, other test generation tools fail — even if they would be able to handle equivalent code without generics (e.g., in the case of Randoop).

Beyond the simple examples of feasibility, as future work we will perform large scale experiments to determine how significant the effect of generics support is in practice, for example using the SF100 corpus of open source projects [4].

EvoSuite is a freely available tool. To learn more about EvoSuite, visit our Web site:

`http://www.evosuite.org`

Acknowledgments. This project has been funded by a Google Focused Research Award on "Test Amplification" and the Norwegian Research Council.

References

1. S. Ali, L. Briand, H. Hemmati, and R. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test-case generation. *IEEE Transactions on Software Engineering (TSE)*, 36(6):742–762, 2010.

2. S. Anand, C. S. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extension to java pathfinder. In *Proceedings of tools and algorithms for the construction and analysis of systems (TACAS)*, 2007.
3. G. Fraser and A. Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, pages 416–419, 2011.
4. G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 178–188, 2012.
5. G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.
6. G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering (TSE)*, 28(2):278–292, 2012.
7. P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *ACM Conference on Programming language design and implementation (PLDI)*, pages 213–223, 2005.
8. M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45(1):11, 2012.
9. M. Islam and C. Csallner. Dsc+mock: A test case + mock class generator in support of coding against interfaces. In *International Workshop on Dynamic Analysis (WODA)*, pages 26–31, 2010.
10. P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.
11. C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 75–84, 2007.
12. C. Parnin, C. Bird, and E. Murphy-Hill. Adoption and use of Java generics. *Empirical Software Engineering*, pages 1–43, 2012.
13. N. Tillmann and N. J. de Halleux. Pex — white box test generation for .NET. In *International Conference on Tests And Proofs (TAP)*, pages 134–253, 2008.