Towards a Framework for Differential Unit Testing of Object-Oriented Programs

Tao Xie[1] Kunal Taneja[1] Shreyas Kale[1] Darko Marinov[2]
[1]Department of Computer Science, North Carolina State University, USA
[2]Department of Computer Science, University of Illinois at Urbana-Champaign, USA
xie@csc.ncsu.edu, {ktaneja,sakale}@ncsu.edu, marinov@cs.uiuc.edu

Abstract

Software developers often face the task of determining how the behaviors of one version of a program unit differ from (or are the same as) the behaviors of a (slightly) different version of the same program unit. In such situations, developers would like to generate tests that exhibit the behavioral differences between the two versions, if any differences exist. We call this type of testing differential unit testing. *Some examples of differential unit testing include regression testing, N-version testing, and mutation testing.*

We propose a framework, called Diffut, that enables differential unit testing of object-oriented programs. Diffut enables "simultaneous" execution of the pairs of corresponding methods from the two versions: methods can receive the same inputs (consisting of the object graph reachable from the receiver and method arguments), and Diffut compares their outputs (consisting of the object graph reachable from the receiver and method return values). Given two versions of a Java class, Diffut automatically synthesizes annotations (in the form of preconditions and postconditions) in the Java Modeling Language (JML) and inserts them into the unit under test to allow the simultaneous execution of the corresponding methods.

1 Introduction

Software developers often manipulate (slightly) different versions of the same software. The most common scenario is changing software systems by evolving them from one version to another. Another scenario is having multiple implementations of the same interface, feature, or functionality. For example, we may have multiple C compilers that handle ANSI C code [14]. Yet another scenario is in the context of mutation testing [4]: intentionally making slight changes to a program to create mutant versions. In all these scenarios with multiple versions of programs, the versions can have different functional behaviors. A typical task then is to determine how the behaviors of one version differ from (or are the same as) the behaviors of a different version. In such tasks, developers would like to generate test inputs that exhibit the behavioral differences between the two versions (producing different outputs for the same inputs), if any differences exist. This type of testing is called *differential testing* [14]. Some example uses of differential testing include regression testing, N-version testing, and mutation testing.

Researchers have developed approaches for differential testing of software at the system level, including testing of C compilers [14], flash file system software [7], and grammar-driven functionality [10]. We focus on *differential unit testing*, where differential testing is applied on a program unit. Specifically, we focus on object-oriented programs, where a unit can be a class or a set of classes. Object-oriented unit tests for a class consist of sequences of method invocations. Behavior of an invocation depends on the method's arguments and the state of the receiver at the beginning of the invocation. Behavior of an invocation can often be observed through the method's return and the state of the receiver at the end of the invocation. Differential unit testing of object-oriented programs thus requires (1) execution of pairs of corresponding methods from the two versions on the conceptually same inputs and (2) comparison of the outputs of the resulting method executions.

We propose a framework, called Diffut, that enables differential unit testing of object-oriented programs. Diffut enables "simultaneous" execution of the pairs of corresponding methods from the two program versions: methods can receive the same inputs (consisting of the object graph reachable from the receiver and method arguments), and Diffut compares their outputs (consisting of the object graph reachable from the receiver and method return values). If the outputs are different, Diffut reports to developers the different behaviors of the two versions.

Given two versions of a Java class, Diffut automatically synthesizes annotations (in the form of preconditions and postconditions) in the Java Modeling Language (JML) [11] and instruments them into the unit under test. These annotations, compiled with the JML compiler into the class bytecode for runtime checking, allow the simultaneous execu-

```
public class MyInput implements Comparable {
    private int o;
    public MyInput(int i) { o = i; }
    public boolean equals(Object that) {
       if (!(that instanceof MyInput)) return false;
       return (o == ((MyInput)that).o);
    }
}

class BST implements Set {
    Node root;
    int size;
    static class Node {
        MyInput value;
        Node left;
        Node right;
    }
    public BST() { ... }
    public void insert(MyInput m) { ... }
    public void remove(MyInput m) { ... }
    public boolean contains(MyInput m) { ... }
}
```

Figure 1. A set implemented as a binary search tree (`BST`)

tion of the corresponding methods. We have implemented Diffut in a tool that operates on Java classes. This paper introduces the Diffut framework, presents our implementation, discusses some alternative implementations of the framework, and proposes differential test generation based on the code instrumented by Diffut.

2 Example

To illustrate our Diffut framework, we use a binary search tree class `BST` that implements a set of comparable elements, shown in Figure 1. The class `MyInput` in the figure is the comparable type of elements (e.g., integers in this example) stored in the stack. Each tree has a pointer to the root node and a field `size` that denotes the number of elements in the tree. Each node has an element and pointers to the left and right children. The class also implements the standard set operations: `insert`, `remove`, and `contains`. The class also has a constructor that creates an empty tree. In this example, we consider `BST` to be the class under test, and its earlier version to be the *reference class*, renamed to `ReferenceBST`. Our current implementation of Diffut adopts class renaming to distinguish the class under test and its old version (reference class). Note that there are other mechanisms, such as renaming packages or using different class loaders, to distinguish classes with the same name.

Given the class under test (`BST`) and its reference class (`ReferenceBST`), Diffut automatically synthesizes for `ReferenceBST` a wrapper class (`WrBST`) shown in Figure 2. This wrapper class inherits `ReferenceBST`, and declares a set of wrapper methods for the public methods (called *reference methods*) declared in `ReferenceBST`. Each wrapper method (1) invokes the wrapped reference method in `ReferenceBST` with the same arguments being passed to the corresponding method under test in `BST`, (2) compares the receiver-object state (of the

```
public class WrBST extends ReferenceBST {
  public WrBST() { super(); }

  public boolean equals(Object t) {
     if (!(t instanceof BST))
       return false;
     BST b = (BST)t;
     if (size != b.size) return false;
     if (!root.equals(b.root)) return false;
     return true;
  }

   public boolean insert(BST c, MyInput m) {
     this.insert(m);
     return this.equals(c);
   }

   public boolean remove(BST c, MyInput m) {
     this.remove(m);
     return this.equals(c);
   }

   public boolean contains(BST c, MyInput m, boolean r) {
     return (r == this.contains(m)) && this.equals(c);
   }
 }
```

Figure 2. Synthesized wrapper class for `ReferenceBST`

`WrBST`/`ReferenceBST` class) after the reference-method execution with the object state (of the `BST` class) after the method-under-test execution, (3) compares the return values of the reference method and method under test if the methods have non-void returns.

Diffut then automatically annotates `BST` with synthesized JML [11] preconditions and postconditions, as well as one extra `WrBST`-type field `_oldThis` and one extra method `_createShadowReferenceObj`. Figure 3 shows the annotated class. The `_createShadowReferenceObj` method constructs an object of the wrapper class `WrBST` and assigns it to `_oldThis`. Diffut annotates the constructor with a JML precondition (marked with "`@`**`requires`**", as shown in Figure 3) that simply invokes `_createShadowReferenceObj` to create a reference-class object for comparisons during later method executions. Diffut also annotates each public method with a JML postcondition (prefixed with "`@`**`ensures`**", as shown in Figure 3). The postcondition of a method `f` invokes on `_oldThis` the corresponding wrapper method with these arguments: first, the current receiver object after the method execution (if `f` is not static); then, `f`'s arguments (if any, taken in the pre-state, denoted with "`\`**`old`**" in JML); and finally `f`'s return (if any, denoted with "`\`**`result`**" in JML). Note that the JML compiler (jmlc) translates the given annotations into Java code. For postconditions that involve "`\`**`old`**`(m)`", jmlc instruments extra code at the beginning of the method body to cache the value of the argument `m`, because `m` may be modified at the end of the method execution. The wrapper method requires the preceding arguments to accomplish its three tasks, described earlier. The synthesized wrapper class for the reference class and the synthesized JML annotations for the class under test form the core mechanism for coordinating

```
class BST implements Set {
   ...
   transient WrBST _oldThis;
   boolean _createShadowReferenceObj() {
      _oldThis = new WrBST();
      return true;
   }

   /*@normal_behavior
     @requires _createShadowReferenceObj();
   @*/
   public BST() { ... }

   /*@normal_behavior
     @ensures _oldThis.insert(this, \old(m));
   @*/
   public void insert(MyInput m) { ... }

   /*@normal_behavior
     @ensures _oldThis.remove(this, \old(m));
   @*/
   public void remove(MyInput m) { ... }

   /*@normal_behavior
     @ensures _oldThis.contains(this, \old(m), \result);
   @*/
   public boolean contains(MyInput m) { ... }
}
```

Figure 3. `BST` annotated with synthesized JML preconditions and postconditions

the execution and result checking of corresponding methods from two versions of the same class.

3 Framework

We first describe how we represent states of non-primitive-type objects. Based on the state representation, we define the input and output of a method execution. We then present techniques for providing the same inputs to pairs of corresponding methods from the two class versions and for comparing the outputs of the pairs of methods.

3.1 State Representation

When a variable (such as the return or receiver of a method invocation) is a non-primitive-type object, we use concrete-state representation from our previous work [20] to represent the variable's value or state. A program executes on the program state that includes a program heap. The concrete-state representation of an object considers only parts of the heap that are reachable from the object. We also call each part a *heap* and view it as a graph: nodes represent objects, and edges represent fields. Let P be the set consisting of all primitive values, including `null`, integers, etc. Let O be a set of objects whose fields form a set F. (Each object has a field that represents its class, and array elements are considered index-labelled fields of the array objects.)

Definition 1 *A* heap *is an edge-labelled graph* $\langle O, E\rangle$*, where* $E = \{\langle o, f, o'\rangle | o \in O, f \in F, o' \in O \cup P\}$.

Heap isomorphism is defined as graph isomorphism based on node bijection [2].

Definition 2 *Two heaps* $\langle O_1, E_1\rangle$ *and* $\langle O_2, E_2\rangle$ *are* isomorphic *iff there is a bijection* $\rho : O_1 \rightarrow O_2$ *such that:*

$$\begin{aligned} E_2 \quad = \quad & \{\langle \rho(o), f, \rho(o')\rangle | \langle o, f, o'\rangle \in E_1, o' \in O_1\} \cup \\ & \{\langle \rho(o), f, o'\rangle | \langle o, f, o'\rangle \in E_1, o' \in P\}. \end{aligned}$$

The definition allows only object identities to vary: two isomorphic heaps have the same fields for all objects and the same values for all primitive fields.

The state of an object is represented with a *rooted* heap:

Definition 3 *A* rooted heap *is a pair* $\langle r, h\rangle$ *of a root object* r *and a heap* h *whose all nodes are reachable from* r.

Another way of representing an object state is to use the method-sequence-representation technique [8, 20]. The technique uses sequences of method invocations that produce the object. The state representation uses symbolic expressions with the grammar shown below:

exp ::= prim | invoc "`.state`" | invoc "`.retval`"
args ::= ϵ | exp | args "`,`" exp
invoc ::= method "`(`" args "`)`"
prim ::= "`null`" | "`true`" | "`false`" | "`0`" | "`1`" | "`-1`" | ...

Each object or value is represented with an expression. Arguments for a method invocation are represented as sequences of zero or more expressions (separated by commas); the receiver of a non-static, non-constructor method invocation is treated as the first method argument. A static method invocation or constructor invocation does not have a receiver. The `.state` and `.retval` expressions denote the state of the receiver after the invocation and the return of the invocation, respectively.

3.2 Method Execution

The execution of an object-oriented program produces a sequence of method executions.

Definition 4 *A* method execution *is a six-tuple* e = *(*m, S_{args}, S_{entry}, S_{exit}, $S_{args'}$, r*), where* m *is the method name (including the signature),* S_{args} *are the argument-object states at the method entry,* S_{entry} *is the receiver-object state at the method entry,* S_{exit} *is the receiver-object state at the method exit,* $S_{args'}$ *are the argument-object states at the method exit, and* r *is the method return value.*

Definition 5 *The* input of a method execution *(*m, S_{args}, S_{entry}, S_{exit}, $S_{args'}$, r*) is a pair (*S_{args}, S_{entry}*) of the argument-object states and the receiver-object state at the method entry.*

Definition 6 *The* output of a method execution *(m, S_{args}, S_{entry}, S_{exit}, $S_{args'}$, r) is a triple (S_{exit}, $S_{args'}$, r) of the receiver-object state at the method exit, the argument-object states at the method exit, and the method return value. (Our framework and its implementation consider only the receiver-object state at the method exit and method return value; argument-object states at the method exit are rarely updated and thus often ignored for checking the output.)*

Note that when m's return is $void$, r is $void$; when m is a static method, S_{entry} and S_{exit} are empty; when m is a constructor method, S_{entry} is empty.

3.3 Method-Execution Comparison

The Diffut framework treats one of the two class versions as the class under test and the other version, called a *reference class*, as the class to be checked against during program execution. The framework provides two key steps in supporting differential unit testing: pre-method-execution setup and post-method-execution checking.

3.3.1 Pre-Method-Execution Setup

The step of pre-method-execution setup occurs before the execution of a method in the class under test (called the *method under test*). In this step, Diffut collects and prepares the same input (defined in Section 3.2) to be executed on the corresponding method in the reference class (called the *reference method*). Note that the execution of the reference method can occur in this step (before the execution of the method under test), concurrently with the execution of the method under test, or after the execution of the method under test (in the step of post-method-execution checking). To implement the preceding feature, some existing capture and replay techniques [6, 15, 16] may be applicable.

In general, arguments for the method under test and the reference method can have different types. For example, a string parameter representing an integer value in the reference method can correspond to an integer parameter in the method under test. To enable Diffut to handle different types during differential unit testing, developers need to construct a conversion method to convert the arguments of the method under test to the format of the reference method. In addition, the class under test and reference class may differ in declared fields: a new field may be added, an old field may be removed, or a field name may be changed. In the concrete-state representation (presented in Section 3.1), a receiver-object state (part of the input) is represented based on the object fields declared in the class under test or the reference class. When fields differ, developers need to construct a conversion method to convert the receiver-object state of the method under test to the one of the reference method.

Using the method-sequence representation (presented in Section 3.1), Diffut does not need to reconstruct a receiver-object state before every execution of a reference method; instead, it can reuse the receiver-object state constructed after the execution of the previous reference method. In such an implementation, a shadow object of the reference class may be needed for each object of the class under test, and the framework needs to ensure that the same method sequences are invoked on these two objects during their life time. The example in Section 2 shows one framework implementation based on the method-sequence representation.

In addition, the behaviors of the method under test and the reference method are sometimes intentionally made different for a subdomain of inputs. For example, the reference method that can handle only positive integers is extended to handle additionally negative integers. When inputs fall into the new subdomain (e.g., negative integers in the preceding example), the differences in the outputs of the two method executions should be ignored. If developers would like to exclude the warnings of these differences automatically, they need to construct a predicate method for determining whether an input falls into this subdomain.

3.3.2 Post-Method-Execution Checking

The step of post-method-execution checking occurs after the executions of a method in the class under test and its corresponding reference method. As discussed earlier in the pre-method-execution setup, these two method executions can be concurrent or sequential (in any order). This step compares the outputs (defined in Section 3.2) of the two executions.

The formats of the outputs (especially the receiver-object state) can be different for the method under test and the reference method. As discussed earlier, developers would need to provide conversion methods that convert the output of the method under test to the format of the reference method. If developers already construct a conversion method for receiver-object states used for the inputs (for the concrete-state representation), developers can reuse the same conversion method for the outputs. For checking output only (e.g., when the method-sequence representation is used to represent the receiver-object states in the inputs of method executions), developers may define an abstraction function [12, 20] such as an `equals` method for mapping the receiver-object states that are deemed to be equivalent into the same abstract values.

4 Implementation

Figure 4 shows the overview of our implementation of Diffut. Diffut takes as inputs the class under test, `C`, and its reference class, `R` (likely an old version of `C`). Diffut gener-

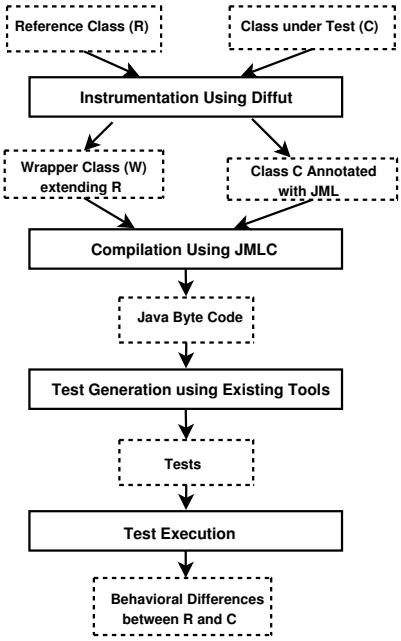

Figure 4. Diffut implementation overview

ates a wrapper class that extends `R`. Our current implementation of Diffut uses the annotations of Java Modeling Language (JML) [11] to implement the steps of pre-method-execution setup and post-method-execution checking. Diffut annotates each constructor or public method of `C` with JML preconditions and postconditions. The preconditions for constructors create shadow reference objects for later method execution and behavior comparison, while postconditions for public methods invoke the corresponding methods in the wrapper class to compare the behaviors of the corresponding methods in `C` and `R`. Developers can customize the wrapper class or the JML annotations to adjust for the intended changes of inputs and outputs between `C` and `R`.

Diffut compiles the wrapper class and class `C` (annotated with JML annotations) using the JML compiler that generates Java bytecode with runtime checking code. When there is any behavioral difference between classes `C` and `R`, the runtime checking code throws an exception signifying JML postcondition violations. We can feed the generated bytecode to the existing Java test-generation tools [20, 21] or new specially developed test-generation tools to conduct *differential test generation* that intends to generate tests on which the two program versions exhibit behavioral differences, if any exist.

The core of our implementation is an automatic instrumenter developed using the Java Tree Builder (JTB) framework [1]. The instrumenter uses JTB to determine the structure of a Java file. We modified the `Visitor` class to add to the code both the JML annotations and the extra fields or methods. The instrumenter also synthesizes a wrapper class for each reference class. Although our current implementation handles one class under test and its single reference class, we can easily extend our implementation to handle one class under test and its multiple reference classes. Note that our current implementation has already supported analysis of multiple classes under test by analyzing one of them at a time. Our framework has been applied in facilitating mutation testing in assessing fault-detection capability of a test suite in our previous work [20].

5 Discussion

5.1 Implementation Variants

Besides the implementation of Diffut that we present, several other implementation variants are possible. Different implementation variants have different tradeoffs in providing a stronger guarantee for finding unintended behavioral differences of two program versions or for filtering out warnings for intended behavioral differences. For example, the current implementation instruments only public methods with JML annotations; an alternative implementation can additionally instrument private methods.

The current implementation uses the method-sequence representation (Section 3.1) to represent and maintain the receiver-object state in the input of a method execution. An alternative implementation is to copy the receiver-object state of the class under test to the receiver-object state of the reference class before each execution of the method under test. As discussed in Section 3.3.1, if object fields of the two class versions are different, a user-defined conversion method is needed to convert an object of the class under test to an object of the reference class in the alternative implementation. Note that for both the current and alternative implementations, we still need an abstraction function or conversion function for comparing the receiver-object states after the execution of the two method versions. Alternatively, we can skip comparing receiver-object states and focus only on comparing return values to provide weaker oracle checking. In general, to reduce the manual efforts in creating abstraction or conversion functions across versions, we may develop some heuristics to construct an initial version of abstraction or conversion functions for developers to start with.

Our current implementation uses the same class loader [17] to run two different classes (the class under test and the reference class) that likely have the same name. To address the naming conflicts, we rename the reference class to distinguish it from the class under test. However, such global renaming can lead to some syntactic or semantic problems. One implementation variant is to copy the original class as well as other classes under the same original package to another new package and update the package names declared in these classes. This variant may ad-

dress some previously encountered problems but may still be fragile for some specific cases, causing syntactic or semantic problems. Note that in the implementation based on class or package renaming, we compare the object states by synthesizing `equals` methods in the wrapper class for the reference class (Figure 2).

An alternative way is to use different class loaders to synchronize the execution of two classes with the same package and class names. For example, Ferastrau [13], a mutation testing tool, can use different class loaders for the original program and each mutant. This Ferastrau implementation uses serialization through a memory buffer to compare objects across class loaders. But the implementation based on multiple class loader may pose restrictions of using specialized class loaders as the execution environment.

5.2 Differential Test Generation

We next describe how the class bytecode compiled from the Diffut-instrumented code can be fed to existing test generation tools to conduct differential test generation: generating tests to exhibit behavioral differences of two class versions, if any differences exist. There is a lot of recent research on automating test generation for object-oriented programs. For the brevity, we illustrate test generation using only two of our previous approaches [20, 21] that generate tests based on exploring states with method sequences: Rostra [20] explores concrete states by executing method sequences with concrete arguments and Symstra [21] explores symbolic states by executing method sequences with symbolic arguments. Both approaches take as input class bytecode of the class under test and generate tests up to a user-specified bound of method sequences.

In concrete-state exploration [20], the inserted field `_oldThis` of the reference-class type (more precisely, the wrapper-class type) in the instrumented class under test allows the states being explored to additionally include the object states of the reference class.

In symbolic-state exploration [21], we explore each path in the method under test and accumulate a path condition, i.e., constraints on symbolic arguments that must hold for the execution to follow the path. The postconditions instrumented on the method under test allow to explore the reference method with the same symbolic arguments, relating the path condition explored in the method under test with the one explored in the reference method.

6 Related Work

McKeeman [14] used the name *differential testing* for testing several implementations of the same functionality, specifically testing different implementations of C compilers. Lämmel and Schulte [10] developed a C#-based test-data generator called Geno and applied it in differential testing of grammar-driven functionality. Groce et al. [7] applied random testing on flash file system software and conducted differential testing on the software and its reference implementation. These previous approaches focus on differential testing of whole systems, whereas our Diffut framework focuses on differential testing of units in object-oriented programs. It is usually easier to provide the same inputs to two versions and to compare two outputs for overall systems as such inputs and outputs do not involve internal states encountered in unit testing.

Sometimes the quality of the existing tests might not be sufficient enough to expose behavioral differences between two program versions by causing their outputs to be different. Then some previous differential test generation approaches generate new tests to expose behavioral differences. DeMillo and Offutt [5] developed a constraint-based approach to generate unit tests that can exhibit program-state deviations caused by the execution of a slightly changed program line (in a mutant produced during mutation testing [4]) in Fortran programs. Korel and Al-Yami [9] created driver code that compares the outputs of two C program versions, and then exploited existing white-box test-generation approaches to generate tests for which the two versions produce different outputs. Winstead and Evan [18] proposed an approach for using genetic algorithms to generate tests that differentiate versions of C programs. Although the preceding approaches for procedural programs can be applied on individual methods of the class under test, our Diffut framework provides a general solution for allowing object-oriented test-generation tools (which generate sequences of method calls) to conduct differential unit testing of object-oriented programs.

Some existing capture and replay techniques [6, 15, 16] capture the inputs and outputs of the unit under test during system-test execution. These techniques then replay the captured inputs for the unit as less expensive unit tests, and can also check the outputs of the unit against the captured outputs. Our Diffut framework can be used in combination with these techniques, but Diffut does not need to capture or save the outputs of the unit.

Cook and Dage [3] proposed the HERCULES deployment framework for upgrading components while keeping multiple versions of a component running. The specific subdomain that a new version of a component correctly addresses is formally specified. For each invocation of the component, multiple versions of the component are run in parallel and the results from the version whose specified domain contains this invocation's arguments are selected. Both HERCULES and Diffut support N-version executions. HERCULES is used to ensure reliable upgrading of components whereas Diffut is used to conduct differential unit testing or differential unit-test generation.

Our previous Orstra approach [19] automatically augments an automatically generated test suite with extra assertions for guarding against regression faults. Orstra first runs the given test suite and collects the return values and receiver-object states after the execution of methods under test. Based on the collected information, Orstra synthesizes and inserts new assertions in the test suite for asserting against the collected method-return values and receiver-object states. Orstra instruments a given test suite (with new assertions), whereas Diffut instruments the class under test (with JML annotations), which can be executed by any test suite or can be fed to existing test-generation tools for conducting differential test generation.

7 Conclusion

Differential testing such as regression testing, N-version testing, and mutation testing considers two (or more) versions of the software and seeks test inputs that exhibit behavioral differences between these versions. To reduce the manual effort in checking the outputs between versions and generating inputs that expose behavioral differences, we have proposed the Diffut framework for differential unit testing of object-oriented programs. Given a class under test and another version of the same class, Diffut automatically generates wrapper classes and inserts annotations written in the Java Modeling Language (JML) into the class under test. For each public method in the class under test, these annotations invoke the corresponding method in the other version of the class (with the cached method arguments) and compare the return values and receiver-object states of the two corresponding method executions. We can run existing tests on the Java code instrumented by Diffut to detect behavioral differences between two versions. Moreover, the Java code instrumented by Diffut can be fed to test-generation tools to conduct differential test generation.

References

[1] JTB: Java tree builder, 2005. `http://compilers.cs.ucla.edu/jtb/`.

[2] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis*, pages 123–133, 2002.

[3] J. E. Cook and J. A. Dage. Highly reliable upgrading of components. In *Proc. the 21st international conference on Software engineering*, pages 203–212. IEEE Computer Society Press, 1999.

[4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[5] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17(9):900–910, 1991.

[6] S. Elbaum, H. N. Chin, M. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *Proc. 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 253–264, 2006.

[7] A. Groce, G. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *Proc. 29th International Conference on Software Engineering*, 2007.

[8] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *Proc. 17th European Conference on Object-Oriented Programming*, pages 431–456, 2003.

[9] B. Korel and A. M. Al-Yami. Automated regression test generation. In *Proc. 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 143–152, 1998.

[10] R. Lämmel and W. Schulte. Controllable combinatorial coverage in grammar-based testing. In *Proc. 18th IFIP International Conference on Testing Communicating Systems*, pages 19–38, 2006.

[11] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, June 1998.

[12] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[13] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard. An evaluation of exhaustive testing for data structures. Technical Report MIT-LCS-TR-921, MIT CSAIL, Cambridge, MA, September 2003.

[14] W. M. McKeeman. Differential testing for software. *Digital Technical Journal of Digital Equipment Corporation*, 10(1):100–107, 1998.

[15] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *Proc. 3rd International ICSE Workshop on Dynamic Analysis*, pages 29–35, May 2005.

[16] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *Proc. 21st IEEE International Conference on Automated Software Engineering*, pages 114–123, November 2005.

[17] Sun Microsystems. Java 2 Platform, Standard Edition, v 1.4.2, API Specification. Online documentation, Nov. 2003. `http://java.sun.com/j2se/1.4.2/docs/api/`.

[18] J. Winstead and D. Evans. Towards differential program analysis. In *Proc. ICSE 2003 Workshop on Dynamic Analysis*, pages 37–40, May 2003.

[19] T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *Proc. 20th European Conference on Object-Oriented Programming*, pages 380–403, July 2006.

[20] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. 19th IEEE International Conference on Automated Software Engineering*, pages 196–205, Sept. 2004.

[21] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381, April 2005.