Safe Test Case Reduction

Brad Howes Anonymous

Abstract

Some object-oriented test cases are inefficient: they perform computation that is unnecessary for producing the final, tested result. This is especially true of automatically-generated test cases. Reducing the size of a test case can improve test runtime and simplify debugging when an error is found. Published techniques for detecting inefficient or redundant test cases are unsafe: they rely on assumptions about ways the tested code will not change.

However, developers do intentionally or unintentionally break these assumptions: they introduce additional data dependencies, or make pure methods impure. We present a safe test case reduction technique that produces statically verifiable *guards* which encode the assumptions introduced during reduction. If these guards are violated, the original test case can be run for a safe result. The guarded tests should combine complete soundness with a faster expected runtime and reduced debugging effort.

1. Introduction

For object-oriented systems, the input for a test is a series of method calls, including arguments, against a set of objects, and the oracle is a set of assertions about the objects' behavior in response. For a given implementation of a system, such a test may be inefficient – it may call methods that have no effect on the objects' tested behavior. The same behavior could be tested by a *reduced* test, which issues only a subset of the method calls of the inefficient test. For example, one simple reduction technique would be to remove all calls to pure methods from a test; by definition, they have no effect on the tested outcome.

There are many benefits to running the reduced test instead of the inefficient test. The reduced test may be significantly faster. It may be easier to determine that the reduced test is redundant with a test already in the test suite, obviating the need to run the reduced test at all [12]. If the reduced test fails, it may be easier for a developer to understand the failure if unimportant method calls have been removed.

However, given any test and a reduction of that test, it is always possible to find an implementation of the tested system on which the two tests produce different results. Thus, any reduction of a test implicitly assumes certain ways in which the tested system will not change. If all pure method calls are removed, this assumes that no pure method will ever be changed, intentionally or not, to be impure. Thus, the reduced test is unsafe; certain faults that would have been caught by the inefficient test will not be caught by the reduced test.

We wish, then, to minimize the time required to *safely* guarantee that a new version of a program still passes a given test. We propose to do so by producing both a reduced version of the original test, and a set of static *guards* that are sufficient to guarantee that the reduction is safe. We call the combination of dynamic test and static guards a *guarded test*. A traditional test case can be seen as a guarded test but with no guards. Using guarded reduced tests should significantly reduce test case complexity and test running time and improve developer understanding with no loss in fault-finding capability.

The paper continues with a motivating example in the following section, followed by a discussion of where inefficient tests come from (Section 3). We then detail our technique (Section 4), and give preliminary results (Section 5) and an evaluation plan (Section 6). Finally, we present future work (Section 7) and conclude (Section 8).

2. Example

For our initial study, we make several simplifying assumptions about tests throughout:

1. Tests can be represented as a single-method straight-line sequence of statements, without loops, branches, or meaningful exceptional control flow (except asserting that an exception was thrown or not thrown). This is true of all automated test generation techniques we know of, and is often true of human-generated tests (eg. JUnit tests [3])
2. There is no aliasing between variables within the test.
3. There is only one method call per statement in the generated test scenario. This can be achieved by introducing fresh temporary variables for any intermediate results.
4. The last operation in each test is the assertion of a single equality comparison between an expected value supplied by the test, and a value provided by the object under test. This is not always the case, but our techniques are easily generalized to multiple or more complex assertions.

As an example of guarded test minimization, consider a simple point class:

```
class Point {
  private int _x, _y;
```

```
  Point(int x, int y) {_x = x; _y = y;}

  int getX() { return _x; }
  int getY() { return _y; }
  void setX(int x) { _x = x; }
  void setY(int y) { _y = y; }
  void translate(int x, int y){
    _x += x; _y += y;
   }
  public String toString() {
    return _x + "," + _y;
  }
}
```

Next we have a test case which exercises this class.

```
Point p = new Point(3, 5);
p.getX();
p.getY();
p.setX(4);
p.setY(6);
p.setX(0);
p.getY();
p.translate(1, 1);
assertEquals("1,7", p.toString());
```

If a static analysis uncovers the fact that `Point.getX()` and `Point.getY()` are pure, then the test case could reduce to the following shortened test code plus two guard checks:

```
Guard: Point.getX writes nothing.
Guard: Point.getY writes nothing.

Point p = new Point(3, 5);
p.setX(4);
p.setY(6);
p.setX(0);
p.translate(1, 1);
assertEquals("1,7", p.toString());
```

Now, when the guarded test is run, the first step is to evaluate if the guards still hold. If so, this reduced test is guaranteed to catch any error caught by the original test. If not, the original test must be run to maintain safety. See Figure 1.

Analysis of the behavior of the method `Point.setX()` reveals that this method affects (mutates) the state of the `Point._x` attribute. Since the terminal statement of the test case relies on the value of this attribute, one might assume that all places where this method is invoked must remain. However, note that in the reduced form above there are two locations where `Point.setX()` is called, but with no intervening assert check. Thus, the first invocation is superfluous, and may be removed:

```
Guard: getX writes nothing.
Guard: getY writes nothing.
Guard: setX reads nothing.
Guard: setX writes at least this._x.
Guard: setY reads nothing.

Point p = new Point(3, 5);
p.setY(6);
```

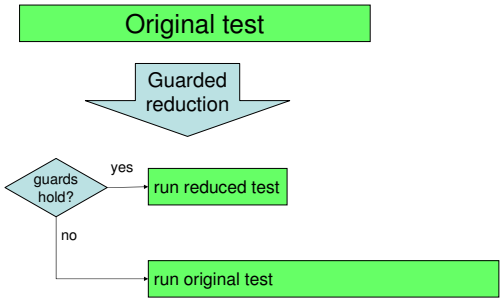

Figure 1: A schematic of the guarded test reduction procedure. The original test is augmented with a faster reduced test, and a set of guards that determine if the reduced test is safe.

```
p.setX(0);
p.translate(1, 1);
assertEquals("1,7", p.toString());
```

The final result is a test case that is smaller and therefore likely to take less time to execute. Furthermore, we have acquired some facts about the underlying behavior of the code under test, which will be used by the testing framework to help determine if and when future code changes invalidate the test case reduction that took place.

3. Sources of Inefficient Test Cases

Developers may manually create tests that appear inefficient. These may be tests of the purity of some method, or they may be actual mistakes. However, it is rare for manually created test to have a significant degree of inefficiency. Automatically-generated tests are another matter.

When automatically generating object-oriented regression tests, the oracle is formed by recording some subset of the behavior of the working system when a method sequence is applied. Regressions are found by comparing the behavior of a new version of the system against that previously recorded. The behavior of a tested object can include both return values from method calls and method calls made to environmental objects; for simplicity, we focus here on return values. Generation techniques fall into two broad categories, which differ in how input sequences of method calls are generated.

Various *test construction* techniques, such as Eclat [8], JCrasher [5], or the commercial tool Agitator [2], generate method sequences one call at a time, using a search algorithm that attempts to produce a suite that maximizes coverage of code, or corner cases, or bugs found. In most cases, the search algorithm is guided by static features of the classes being tested; for example, tests that inefficiently call a currently-pure method will simply not be generated. An implicit assumption of this guided search is that these features are stable.

A *capture/replay* technique, such as test factoring [9], SCARPE [7], or DejaVu [4], captures each input method

sequence from an execution of the system, whether manually or automatically driven. Existing capture/replay techniques capture every call made to the tested objects during an execution, and replay them all. The advantage of this completeness is that generated tests are guaranteed to be safe – any breaking change in the implementation of any method in the tested classes will be detected by these tests. However, these tests can be inefficient, consisting of millions of method calls, and failures can be difficult to debug.

It is possible to trade safety for efficiency in capture/replay tests by eliminating method calls that can be proven to have no effect on the final, tested behavior of the tested component: statically determined pure methods, or method calls that can be dynamically shown not to change the tested state. This is the approach taken by Xie, Marinov, and Notkin in Rostra [12]. However, the safety tradeoff is unnecessary – it is possible to have both efficiency *and* safety, using the technique presented here, which remembers the *static guards* required for safe reduction, and verifies them incrementally and quickly in new versions.

Our technique focuses on safe reduction of tests generated through capture/replay techniques. However, it may be possible to apply similar techniques to tests generated through test construction, by remembering the static guards that guided the initial test search, and recommending a repeat search if these are violated.

4. Approach

In order to safely reduce a test case, we need to know what method calls are extraneous to the tested result, and we must be able to recognize when a reduction no longer applies due to code changes. To satisfy these needs, we propose a technique that augments a test case with *guards*, which are statically evaluated prior to running the reduced test case to verify that the conditions under which the reduction originally took place still hold. To produce the reduced test and the guards, we use a simplified slicing algorithm that makes explicit which properties of which methods are necessary to justify each statement that is removed.

We perform a static analysis of the code under test in order to obtain the set of data elements (class fields and method arguments) that a method reads from and writes to when it executes. Collectively, these data sets are known as *method summaries*. The elements of the summary sets uniquely name the field or argument used or manipulated within the method or within any call chain started from the method. Section 4.1 discusses in detail how we generate method summaries and manage name conflicts.

Once we have the method summaries for the code under test, we can then proceed to reduce the test case. Our approach is to use a backwards static slicing of the test case in order to determine whether a method has the potential to influence the outcome of the test case. If the method summaries of a method call indicate that it cannot affect any of the data elements used by subsequent method calls, including the terminating test assertion, then the method is removed from the test case. Section 4.2 describes the slicing and reduction steps in detail.

Regardless whether the method call is kept or not, we annotate the test case with a set of static guards generated from the call's method summary. For removed method calls, the guards determine when the removal is *potentially* no longer safe, while guards for kept calls identify *potential* added dependencies on earlier statements. Details of guard generation are covered in Section 4.3.

4.1 Generating Method Summaries

Method summaries are the result of a static analysis performed on the code under test. The analysis determines which fields and method arguments the method reads from or writes to. This information will be used during the slicing of the test case (Section 4.2) and during static guard generation (Section 4.3).

Our approach for generating the summaries is to execute a points-to analysis over the code under test to precisely identify the set of objects a method method may encounter during its execution. We next execute a intra-procedural side-effect analysis of the method using the points-to graph, and statically evaluate each statement in the method to see if it reads from or writes to an object field or method argument. If the statement is a method invocation (call-site), the analysis will proceed to analyze the called method, with any attributes found within the call-chain for the call-site propagated back up to the calling method. Finally, the individual statement read/write attributes are collected together to become the method summary.

For example, given the `Point` class show in Section 2, the extracted method summaries for the class methods are shown in Table 1.

Table 1: Point Method Summaries

Key	Reads	Writes
Point		_x, _y
setX		_x
getX	_x	
setY		_y
getY	_y	
move		_x, _y
xlate	_x, _y	_x, _y
toString	_x, _y	

The propagation of read/write attributes presents a naming issue. Consider the following class:

```
class PointPair {
  private Point _a;
  private Point _b;
  PointPair( Point a, Point b ) {
    _a = a, _b = b;
  }
  void setX( int x ) {
    _a.setX( x );
    _b.setX( x );
  }
}
```

Within the method `PointPair.setX`, there are two invocations of `Point.setX` applied to two different objects. The analysis of `Point.setX` reveals that it writes to `Point._x`, but when this fact is propagated up to the call-sites in `PointPair.setX`, they appear the same. Our solution is to transform, if possible, the names of propagated attributes with the class and field the attribute actually refers to. For the example above, the write attribute `Point._x` becomes

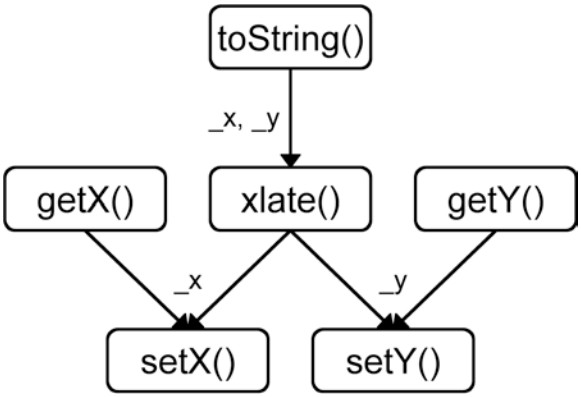

Figure 2: Data-dependency graph for test case from Section 2. The root of the graph is `toStriing()`.

`PointPair._a._x` and `PointPair._b._x` for the respective call-sites, resulting in two distinct write attributes in the method summary for `PointPair.setX`.

4.2 Test Case Slicing and Reduction

To identify which statements may be removed from a test case, we need to understand which statements are required to satisfy the data needs of the test's final assertion. Intuitively, one needs to build a data-dependency graph starting with the method call in the assertion statement of the test case, where the graph nodes are the method calls in the test case, and the presence of a directed edge between two nodes indicates that a child writes to one or more fields used by the parent it is connected to. Once the graph is complete, any nodes that do not have a path from the root (the assertion statement) by definition have no effect on the behavior of the method in the assertion statement, and thus no affect on the assertion itself.

Figure 2 shows a data-dependency graph for the example test case in Section 2. Note that the top-down graph flow is opposite of the top-down linear flow of the test case. There is no directed path from the root of the graph to the routines `Point.getX()` and `Point.getY()`, indicating no data-dependencies between them.

Because we have restricted our approach to linear test cases, we can do the data-dependency analysis and method culling in one iteration over the test case, starting at the final assertion and working backwards. We manage a set of *active reads* (**A**) that reflect the unconnected or unsatisfied data elements forward of the current analysis point. If the method summary of the method call at the current analysis point does not match any of the elements in **A**, then the method will not have any affect on the behavior of succeeding methods, and it may be removed from the test case. In short, we perform a backwards static slice of the test case, and our abstract state is the contents of **A**.

A brief outline of the steps involved in our test case slicing and reduction is as follows:

1. Obtain the method summary of the current method.
2. Determine whether the method satisfies any data dependencies
3. Update propagated data dependency set **A**
4. Install any guards for the method
5. Move to previous method call, or stop when done

Again, the method summaries tell us whether a method *may read* (**R**), *may write* (**M**), or *must write* (**W**) zero or more fields. For the slicing algorithm, we are interested in the contents of **R** and **W**, and whether the set **M** is empty.

We start the slicing with the **A** set containing the *may read* **R** values from the method call in the assertion statement:

$$A_0 := R_n \tag{1}$$

where **n** is the number of method calls executed within the test case. For each preceding method call $\mathbf{C}_{n-i} \mid \mathbf{i} \in \{1,...,n\}$, we calculate the intersection of the method's **W** attribute set and the **A** set:

$$A_{i-1} \cap (W_{n-i} \cup M_{n-i}) \tag{2}$$

To determine whether a **W** attriibute satisfies an entry in **A**, we iterate over the values of **A**, looking for an exact match in **W**, or an element in **W** that is an *effective* match: the effect of the write is the same as if the attribute had completely matched the read attribute in **A**. For instance, in the `PointPair` example of Section 4.1, a **W** attribute of `PointPair._a` would satisfy the attribute `PointPair._a._x`.

If the resulting intersection set is empty and the method has an empty **M** set, then we may safely remove the method call from the test case. Otherwise, we keep the method call, and update the **A** set by first removing from it the result of the intersection calculated above, followed by a union of **A** with the method's **R** set:

$$A_i := R_{n-i} \cup (A_{i-1} - (A_{i-1} \cap W_{n-i})) \tag{3}$$

In other words, we remove all fields guaranteed to be satisfied by the method call and add in any fields the method itself may be dependent on. Note that the contents of **M** has no affect on **A**; it only inhibits a method from being removed when it is not empty.

If at any time the **A** set becomes empty, we stop the slicing, since there is no possibility that any preceding method calls would affect the test case assertion. This constraint should be satisfied even if we work through the entire test case ($\mathbf{i} = \mathbf{n}$), since the test case begins by creating the object under test, and the constructor for the test object would have initialized the field, even if only to a default value.

4.3 Guard Generation

As our algorithm visits each method call in the test case, we annotate the call with one or more *guards*. The set of guards for the test case is the union of the guards for each of the method calls in the unreduced test case. Prior to executing the test case, the test case's guards are evaluated to see if the property they represent still holds. If all guards pass, then the reduced test case is executed; otherwise, the original, unreduced, test case runs. In short, guards protect the test case from future code changes, signaling the fact that the conditions under which a test case was previously reduced no longer apply.

We have identified the following set of guards in our approach:

1. *reads at most* - the contents of the **R** set from the method summary
2. *writes at least* - the contents of the **W** set from the method summary
3. *writes at most* - the contents of the **M** + **W** sets

The first guard states that the method does not rely on any more data than found in the **R** set. If a code change were to result in an $\mathbf{R}' \subset \mathbf{R}$, then the reduced test case is still safe and should be run. The next time the test is reduced, it may now prove possible to reduce it even further.

The second and third guards state the minimum and maximum set of fields that the method may change. The minimum set contains the fields that the method always writes to when it is invoked, whereas the latter is the set of fields that the method may write to. If either or both of these bounds change, then the dependency graph generated for test case is no longer valid and the original test case is run.

Intuitively, the relative sizes of the $\mathbf{M}_i$ and $\mathbf{W}_i$ sets for method call $\mathbf{C}_i$ may provide a measure of how likely the call would be removed. The more elements there are in $\mathbf{M}_i$, the greater the likelihood that one of the elements will be a member of the propagated **A** set, in which case the call must remain in the test case since the **M** set only indicates potential writes, and not the guaranteed writes of **W**. Furthermore, for each method $\mathbf{C}_i$ kept in a test case, one could reasonably expect **A** to grow as it is joined with the call's $\mathbf{R}_i$ set, thus increasing the potential to keep methods $\mathbf{C}_j$ | $\mathbf{j} \in \{0,...,i\}$.

5. Preliminary Results

We have created a simple test bed to exercise the individual components of our approach. The test scenario uses class `Point`:

```
public class Point {
    private int _x;
    private int _y;

    public Point( int x, int y )  {
        _x = x; _y = y;
    }
    public int getX() {
         return _x;
    }
    public int getY() {
        return _y;
    }
    public void moveBy( int x, int y ) {
        _x += x;
        _y += y;
    }
    public void moveHorizontally( int x ) {
        _x += x;
    }
    public void moveVertically( int y ) {
         _y += y;
     }
    public String toString() {
        return "(" + _x + "," + _y + ")";
    }
}
```

There exist two hand-crafted driver programs, one for each of the above classes. These driver programs exist to simulate the running of a test scenario, but outside of a testing infrastructure. First, the driver for testing the Point class:

```
public class TestPoint {
    static public void
        assertEqual( boolean condition ) {
        System.out.println(
            "test case returned " + condition );
    }
    public static void main(String[] args) {
        Point p =
            new Point( 10, 20 );
        p.moveBy( 4, 4 );
        p.getX();
        p.getY();
        p.moveBy( 3, 8 );
        p.getX();
        p.getY();
        p.setX( 5 );
        p.getY();
        p.getX();
        p.setY( 10 );
        p.moveHorizontally( 1 );
        p.moveVertically( 2 );
        p.getX();
        p.getY();
        System.out.println( p.toString() );
        assertEqual(
             p.toString().equals( "(6,12)" ) );
    }
}
```

The test case simply exercises the `Point` API, with a final `assertEqual` to check that the previous calls produced the expected final state of the `Point` object.

When processing is done, the following attributes are revealed:

```
Point: int getY()
  <Read Point:_y>
Point: void setX(int)
  <Write Point:_x>
Point: int getX()
  <Read Point:_x>
java.lang.String: boolean equals(java.lang.Object)
  <Read java.lang.String:offset>
  <Read java.lang.String:value>
  <Read java.lang.String:count>
Point: void setY(int)
  <Write Point:_y>
Point: void moveHorizontally(int)
  <Read Point:_x>
  <Write Point:_x>
Point: void moveBy(int,int)
  <Write Point:_y>
  <Read Point:_y>
  <Read Point:_x>
```

```
  <Write Point:_x>
Point: void moveVertically(int)
  <Write Point:_y>
  <Read Point:_y>
```

5.1 Performance

Rather discouragingly, our implementation of summarization takes 2.25 minutes to run on an Apple PowerBook 1.5GHz with 1GB of RAM (the Java VM is limited to a maximum 400MB heap with the -Xmx400m option). Enabling verbose logging reveals that a majority of the time is spent generating Soot Jimple representations for the Java runtime and support classes. Using a combination of Soot options, we were able to obtain a set of Jimple files for all of the runtime classes reached by the call graph rooted by our TestPoint class. We then tried to have Soot use these files instead of dynamic Jimple generation, we encountered errors in the Jimple processing. Apparently, there are discrepancies (bugs) between what Soot writes out in its Jimple representation and what it reads in.

We also submitted the TestPoint class to a Purity Analysis Kit [10]. Interestingly, we obtained similar timing results.

We have briefly investigated generating the call graph iteratively. Previous work by Souter, Pollock [11] showed promising results using enhancements to the Flex Compiler Infrastructure [6]. Their call graph generation algorithm is based on the Cartisian Product Algorithm of Agesen [1], with modifications to support reanalysis of only the code that changed.

6. Evaluation

Our evaluation is based on monitoring actual developer activity without the benefit of automatically generated and reduced tests, and then simulating the impact of introducing test reduction into the development process. This allows us to get an idea of the benefits of our approach before a full-fledged tool has been built. We can also simulate scenarios that real developers would not stand for, such as using an inefficient test generation technique without reduction.

We have monitored a single developer performing development and maintenance on `fdanalysis`, a Java program of about 9000 lines of code. `fdanalysis` is a package for analyzing data collected during development and debugging sessions. It performs mainly text processing and time calculation. We have captured 1600 snapshots of the state of the program during development, which include the introduction and fixing of 12 regression errors. Most of the running time of the existing test suite operates on the high-level API of the package, sending in text files and making assertions on text output.

In our simulated scenario, the developer would like to generate unit tests for the edited component of the program by using a capture/replay test generation technique, test factoring [9], on the current suite of system tests, most of which operate on the high-level API of the package, sending in text files and making assertions on text output. However, the unit tests generated by test factoring are inefficient, and not a significant improvement on the current tests. We would like to evaluate whether our safe test reduction technique, in combination with test factoring, would produce a safe, efficient unit regression test suite.

In our simulated scenario, test factoring is run each night overnight to produce inefficient unit tests, which are then reduced by safe test reduction into guarded reduced tests. For each snapshot that we have of development during the next day, we can compare running the guarded reduced tests with the unreduced generated tests, and the original system tests. Our hypotheses are:

- The guarded reduced tests are an order of magnitude smaller (in number of method calls against the tested component) and faster than either the unreduced generated tests.
- In about 1% of the captured snapshots, a static guard is violated. When a static guard is violated, the reduced test is useless, and an unreduced test must be run to guarantee the system is correct. If this happens very often, the overhead of evaluating the guards will eliminate any gains from test reduction. However, if it happens very seldom, we may have to conclude that the danger of using unguarded reduced tests is not very great, which reduces the value of the overhead of generating and evaluating guards.

Unfortunately, in the time allotted for this course, we have not been able to produce an implementation of our approach that correctly handles all of the generated unit tests, so this evaluation remains as future work.

7. Future Work

There are several ways that this work can be expanded upon.

1. Our implementation must be made robust enough to be evaluated against real-world test cases.
2. The basic slicing algorithm should be enhanced to work correctly in the presence of aliasing.
3. A usable implementation of our technique requires an incremental mechanism for evaluating guards. We believe that this is easily achieved, by caching in memory or on disk the call graph required for guard generation, and propogating changes in place. However, our current implementation must re-run the entire summarization, a time requirement that overwhelms the runtime benefits of reduction.

8. Conclusion

Automated test generation appears poised to become a much more common development tool. It can provide a "second opinion" on code, considering cases that the developers' own assumptions and biases may cause them to overlook. It can also dramatically reduce the effort and improve the effectiveness of testing code.

We have introduced a technique for reducing the average runtime of inefficient test cases, whether automatically or manually generated, without making any unsafe assumptions about the ways in which the tested code may change. These properties should be an essential guarantee of future test generation techniques.

References

[1] O. Agesen. The Cartesian Product Algorithm. In *ECOOP'95 Conference Proceedings*, 1995.

[2] Agitar software. http://www.agitar.com, 2005.

[3] M. Albrecht. Testing Java with JUnit. http://www.ddj.com/documents/s=1679/ddj0302b/, 2003.

[4] J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 48–59, New York, NY, USA, 1998. ACM Press.

[5] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Softw. Pract. Exper.*, 34(11):1025–1050, 2004.

[6] Flex compiler infrastructure. http://www.flex-compiler.lcs.mit.edu/.

[7] A. Orso and B. Kennedy. Selective Capture and Replay of Program Executions. In *Proceedings of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005)*, pages 29–35, St. Louis, MO, USA, may 2005.

[8] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, pages 504–527, Glasgow, Scotland, July 25–29, 2005.

[9] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *ASE 2005: Proceedings of the 21st Annual International Conference on Automated Software Engineering*, Long Beach, CA, USA, November 9–11, 2005.

[10] A. Salcianu and M. Rinard. Purity and side effect analysis for Java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, 2005.

[11] A. Souter and L. Pollock. Incremental call graph reanalysis for object-oriented software maintenance. In *Proceedings International Conference on Software Maintenance*, pages 682–691, 2001.

[12] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 04)*, pages 196–205, September 2004.