Path Slicing per Object for Better Testing, Debugging, and Usage Discovery

Sudeep Juvekar
Jacob Burnim
Koushik Sen

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-132
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-132.html

September 26, 2009

Path Slicing Per Object for Better Testing, Debugging, and Usage Discovery

Sudeep Juvekar
EECS Department
UC Berkeley, CA, USA
sjuvekar@cs.berkeley.edu

Jacob Burnim
EECS Department
UC Berkeley, CA, USA
jburnim@cs.berkeley.edu

Koushik Sen
EECS Department
UC Berkeley, CA, USA
ksen@cs.berkeley.edu

***Abstract*—Given an execution trace of an object-oriented program and an object created during the execution, a path slice per object with respect to the object, or PSPO, is a part of the trace such that (1) the sequence of public methods invoked on the object in the trace is same as the sequence of public methods invoked on the object in the slice, and (2) given a method invocation in the slice, the state of all objects accessed by the method is same in both the trace and slice. A generator for a PSPO (or GPSPO in short) is a program such that its only execution trace is the PSPO. We argue that GPSPOs can be useful for debugging, creating test harnesses, creating regression test suites, discovering usage and construction patterns of a class. We present an algorithm to create GPSPOs given an execution trace and an object. We have implemented the algorithm in a prototype tool for Java, and we present several examples that demonstrate the effectiveness of our algorithm and the utility of GPSPOs.**

I. Introduction

We propose a *dynamic path slicing* technique for object-oriented programs. Given an execution trace of an object-oriented program and an object created during the execution, we show how to capture the sequence of all method calls on the object, in addition to all method calls needed to construct and compute all necessary method parameters. This sequence is a *slice* of the given execution trace for the given object. We call such a sequence a *path slice per object*, or PSPO. Moreover, for such a PSPO, we also construct a *generator*, or GPSPO—a simple, closed executable program whose trace is exactly the given PSPO. For example, a GPSPO of the execution trace of the program in Figure 1 with respect to the `LinkedList` object created at line 4 is shown in Figure 2.

We argue that such GPSPOs have many potential applications:

- GPSPOs can be used to understand how a particular class has been used in a program. Such an understanding could help us to learn the typical usage pattern of the class.
- GPSPOs can also be treated as small unit regression tests for the class of the object. Therefore, whenever we make any change to the class, we can quickly test the class by running the GPSPOs for that class instead of running the whole program from which the GPSPOs were derived.
- GPSPOs can simplify debugging—if an error occurs in an object during an object-oriented execution, we may be able to use the GPSPO of the object for debugging rather than using the full program execution.
- A GPSPO captures a legal sequence of method calls on a target object. Therefore, GPSPOs can be used to generate test harnesses, which may otherwise be difficult to construct manually.

Computation of a PSPO and GPSPO from an execution trace poses two key challenges: (1) a method called on an object could be private and thus cannot be directly included in the GPSPO, and (2) a GPSPO may include a method on an object, but the object's definition is subsumed by another method present in the GPSPO. We illustrates these challenges in the next section through examples. We solve the first problem by including in the GPSPO the immediate public method that encloses the private method. We solve the second problem by including in the GPSPO some method call that returns the undefined object. However, the inclusion of a new method may necessitate the inclusion of yet more methods because the included method may access some other objects whose method invocations are not present in the current GPSPO. In order to resolve these complications, we propose a GPSPO computation algorithm based on a fix-point computation.

We have implemented our GPSPO computation algorithm in a prototype tool in Java and have applied it to several open and closed source Java programs. Our experiments show that, on average, the traces generated by GPSPOs are significantly shorter than the original traces. This suggests that GPSPOs can help in program debugging. Moreover, our experiments suggest that these GPSPOs can provide compact usage examples for complicated APIs. Our experiments also show that test harnesses generated from GPSPOs can increase test coverage while giving few spurious test failures. The test harnesses produced for four benchmarks in our suite resulted in only one such test failure.

Our definition of a GPSPO combines ideas from *dynamic program slicing* [16], [2], [1], [17], [9], [5], [30], [26] and recent work in automatically carving tests from program executions [22], [20], [6], [15]. In both dynamic program slicing and test carving, an execution trace a program is analyzed to determine how the program modifies some target set of variables or objects. In the first case, this analysis is used to produce a simplified, executable subset of the program, called a *program slice*, which captures all of the modifications or uses of the targets. In test carving, this analysis is used to produce a sequence of methods, along with pre- and post-conditions,

```
 1  public class Example1 {
 2    public static void main (String[]) {
 3      TreeSet set = new TreeSet();
 4      LinkedList list = new LinkedList();
 5      Integer I;
 6      for (int i=0; i<3; i++) {
 7          int j = (i * 7) % 11;
 8          I = new Integer(j);
 9          list.addLast(I);
10          set.add(I);
11      }
12    }
13  }
```

Fig. 1. A simple Java program

```
public class GPSPO1 {
  public static void main (String[] args) {
    LinkedList X1 = new LinkedList();
    Integer X2 = new Integer(0);
    X1.addLast(X2);
    Integer X3 = new Integer(7);
    X1.addLast(X3);
    Integer X4 = new Integer(3);
    X1.addLast(X4);
  }
}
```

Fig. 2. A GPSPO with respect to the object created at line 4 in Figure 1

```
public class GPSPO2 {
  public static void main (String[] args) {
    TreeSet X1 = new TreeSet();
    Integer X2 = new Integer(0);
    X1.add(X2);
    Integer X3 = new Integer(7);
    X1.add(X3);
    Integer X4 = new Integer(3);
    X1.add(X4);
  }
}
```

Fig. 3. A GPSPO with respect to the object created at line 3 in Figure 1

```
public class GPSPO {
  public static void main(String[] args) {
    MatchActionProcessor X20 = MatchActionProcessor
        ();
    X20.addAction("<[^>]*>");
    FileInputStream X47 = new FileInputStream("file.
        txt");
    PrintStream X48 = System.out;
    X20.processMatches(X47, X48);
  }
}
```

Fig. 4. Usage scenario for the `MatchActionProcessor` class in the grep application

which can be replayed as program tests. A GPSPO can be viewed as a dynamic program slice of the complete unrolling of the original program. That is, the program obtained by unrolling all loops and specializing all conditional statements for the given execution trace.

In summary, we make the following contributions:

- We introduce the notion of a path slice per object and argue that path slices per object can aid effective debugging, testing, and class usage understanding.
- We propose an iterative algorithm for computing a path slice per object.
- We provide an implementation of our technique for Java and evaluate our technique on several real-world Java programs.

II. Overview

An execution trace (or simply a trace) of an object-oriented program is the sequence of methods invoked during the execution. For each method, the trace records the object on which the method is called, the values of the arguments that are passed, the value that is returned, and the trace generated by the body of the method. Given an execution trace of an object-oriented sequential program and an object created during the execution, a *path slice per object* (or PSPO in short) of the execution with respect to the object is a part of the trace such that:

1) the sequence of methods invoked on the object in the trace is same as the sequence of methods invoked on the object in the slice, and
2) given a method invocation in the slice, the state of all objects accessed by the method is same in both the trace and slice.

A *generator* for a PSPO (or GPSPO in short) is a closed program such that its only execution trace is the PSPO.

Consider the Java program in Figure 1. The program creates a `TreeSet` and a `LinkedList` and invokes the `add` and `addLast` methods on them, respectively.

A GPSPO of the execution trace of the above program with respect to the `LinkedList` object created at line 4 is shown in Figure 2. The execution trace of this GPSPO is the PSPO of the execution trace of the program in Figure 1 with respect to the `LinkedList` object. Note that the GPSPO completely eliminates the method invocations on the `TreeSet` object. This is because any method invocation in the PSPO does not access any `TreeSet` object. Figure 3 shows the GPSPO of the execution trace of the original program with respect to the `TreeSet` object created at line 3.

A. Applications of GPSPOs

We believe that GPSPOs have many applications in program testing, understanding, and debugging. These applications include:

a) **GPSPOs as regression unit tests:** A GPSPO shows how a particular class has been used in the program, i.e., it shows how an object of the class has been created and how the various methods have been called on the object. Note that the creation and method invocations on the object may use other objects. A GPSPO creates all such objects and makes sure that they are in the right state. Thus a GPSPO can be thought of as a simple, small program that exercises all methods of the object in the same way as the original program does. Therefore, a GPSPO could be used as a regression unit test for the class of

```
public class GPSPO {
  public static void main(String[] args) {
    File X148391 = new File("test.tex");
    TeXWordFinder X148392 = new TeXWordFinder();
    FileWordTokenizer X148397 = new
        FileWordTokenizer(X148391,X148392);
    ...
  }
}
```

Fig. 5. Code snippet to create `FileWordTokenizer`

```
public class Test1 {
  public static void main (String[] args) {
    LinkedList X1 = new LinkedList();
    Integer X2 = new Integer(readInteger());
    X1.addLast(X2);
    Integer X3 = new Integer(readInteger());
    X1.addLast(X3);
    Integer X4 = new Integer(readInteger());
    X1.addLast(X4);
  }
}
```

Fig. 6. A test harness created from the GPSPO in Figure 2

the object. Note that a GPSPO does not introduce assertions, but they can be introduced based on the return values of the various method invocations in the GPSPO. Moreover, one can assume that the class itself has various programmer written assertions.

b) **Discovering class usage:** Since a GPSPO contains all the method invocations on a particular object, it can be used to learn about the typical usage pattern of the class of the object. For example, a GPSPO shown in Figure 4 from the `grep` benchmark shows a typical usage scenario of the class `MatchActionProcessor`. The usage simply shows how `grep` has been implemented using the class `MatchActionProcessor` in the Apache regular expression library. Note that the GPSPO also shows how the objects `X47` and `X48` are defined. These objects are passed as arguments to the method `processMatches` of the class `MatchActionProcessor`.

c) **Finding code snippets to create an object of a given type:** Mandelin et al. [19] showed in their jungloid mining paper that it is often difficult for a programmer to create an object of a given type. The paper then proposed a static method to help programmers in creating object of a given type. Our technique provides a dynamic technique to come up with such object creation code. This is because a GPSPO for an object shows how the object is created and shows operations on all the objects that are required to create the object. For example, the code in Figure 5 shows how one can create an object of class `FileWordTokenizer` in the Jazzy spell checker, which is otherwise difficult to come up with if the programmer is not familiar with the API of the class `FileWordTokenizer`.

d) **Creating test harnesses from GPSPOs:** We have already argued that a GPSPO could be treated as a regression test, but one can go further and convert them into unit test harnesses, or parametric unit tests [24]. Such test harnesses could be used with an automated test generation tool such as Java Pathfinder [25], DART [7], or CUTE [23], PEX [24] to improve test coverage of the class of the object involved in the GPSPO. In order to convert a GPSPO into a test harness, we simply replace all primitive values used in the GPSPO by inputs. For example, Figure 6 shows the test harness created out of the GPSPO in Figure 2. The function `readInteger` reads an integer from console or file. Since the test harness invokes all the methods on the object in the same order as in the original execution, it will less likely show an invalid test execution. Moreover, an automated test generation tool will improve the coverage of the class. For example, in our experiments, the GPSPO in Figure 2 covered 30 branches, while the test harness in Figure 6 covered 44 branches in the program without exploring any illegal path while testing the `java.util.LinkedList` class.

e) **GPSPOs for better debugging:** Similar to many other slicing techniques, GPSPOs simplify debugging as they create a portion of the original trace that is relevant to debugging an object. For example, in the program in Figure 1, assume that the `addLast` method of `LinkedList` has a bug and violates an assertion. There could be two reasons behind the bug: (1) the program is violating the contract of the `LinkedList` class, or (2) the `LinkedList` class itself has a bug. The former cause could be discovered by looking at all the arguments that are passed to the methods invoked on the `LinkedList` object and the latter cause could be found by executing the `addLast` methods in the right object state. A GPSPO shown in Figure 2 helps to achieve both these goals. This is because the GPSPO shows all method invocations and all method arguments that could have potentially affected the behavior of the `LinkedList` object. The GPSPO is simple without any branching statement and excludes the `TreeSet` object completely. One can repeatedly execute this smaller and simpler GPSPO and debug it. An advantage of debugging the simpler GPSPO is that the programmer does not have to look at the entire program involving the `TreeSet` object. Moreover, condition 2 of our definition of PSPO also guarantees that if a bug appears in the `addLast` method while executing the program in Figure 1, the same bug will also appear while executing the GPSPO in Figure 2. This significantly simplifies the process of debugging in a program that creates millions of objects, but a PSPO only involves hundreds of objects.

Note that, unlike traditional program slices, a GPSPO removes all the branching statements and operations on primitive types. This helps a programmer, who is trying to debug an object, to focus only on the method invocations on the object, rather than spending time on the irrelevant control flow and irrelevant operations on the primitives.

```
 1 public class Problem1 {
 2     private int val;
 3     public Problem1() {  val = 0; }
 4     private void decrement(int x) {
 5         val = val - x;
 6     }
 7     public void transfer(Problem1 from) {
 8         val = val + 100;
 9         from.decrement(100);
10     }
11     public void print() {
12         System.out.println(val);
13     }
14     public static void main(String[] args) {
15         Problem1 o1 = new Problem1();
16         Problem1 o2 = new Problem1();
17         o2.transfer(o1);
18         o1.print();
19         System.out.println("Done");
20     }
21 }
```

Fig. 7. An example showing a problem with GPSPO computation due to the presence of private methods.

```
public class GPSPOInvalid {
    public static void main(String[] args) {
        Problem1 X1 = new Problem1();
        X1.decrement(100); //illegal call to
                           //private method
        X1.print();
    }
}
```

Fig. 8. An invalid GPSPO of the only execution of the program in Figure 7 with respect to the object `o1`

```
public class GPSPOValid {
    public static void main(String[] args) {
        Problem1 X1 = new Problem1();
        Problem1 X2 = new Problem1();
        X2.transfer(X1);
        X1.print();
    }
}
```

Fig. 9. A valid GPSPO of the only execution of the program in Figure 7 with respect to the object `o1`

```
 1 public class Problem2 {
 2     private Problem2 next;
 3     private int val;
 4     public Problem2(int val) {
 5         this.val = val;
 6         this.next = new Problem2();
 7     }
 8     private Problem2() { val = 10;}
 9     public Problem2 getNext() { return next; }
10     public void print() {
11         System.out.println(val);
12     }
13     public static void main(String[] args) {
14         Problem2 o1 = new Problem2(9);
15         Problem2 o2 = o1.getNext();
16         o2.print();
17         System.out.println("Done");
18     }
19 }
```

Fig. 10. An example showing a problem with GPSPO computation due to the presence of private methods.

B. Computation of GPSPOs

We present a technique to compute a GPSPO given an object and an execution trace. There are two key challenges in such a computation: (1) a method called on an object could be private and cannot be directly included in the GPSPO, and (2) a GPSPO may include a method on an object, but the object's definition is subsumed by another method present in the GPSPO. We illustrate these problems using two examples.

Consider the program in Figure 7. The program defines a class `Problem1` that has one public constructor and four methods, one of which is private. A GPSPO of the only execution of the program with respect to the object `o1` is shown in Figure 8. However, this program does not compile because the program calls the private method `decrement` on `X1`. This illustrates the first problem associated with the computation of a GPSPO. A possible way to solve this problem is to include in the GPSPO the immediate public method invocation that encloses the private method. For example, in our case we include the invocation of the public method `transfer` in the GPSPO. This method subsumes the private method `decrement`. The modified valid GPSPO for object `o1` is shown in Figure 9. Our GPSPO computation algorithm adopts this solution.

Consider the program in Figure 10. The program defines a class `Problem2` that has a public and a private constructor. A GPSPO of the only execution of the program with respect to the object `o2` is shown in Figure 11. The GPSPO is not valid because it calls the private constructor of `Problem2`. After applying the solution proposed above for private methods, we get the modified GPSPO shown in Figure 12. However, this GPSPO is not a valid program as the variable `X2` is not defined before it is used—the definition of `X2` was implicitly present in the constructor of `X1`. Therefore, in order make the GPSPO a valid program, we include any other method invocation that returns `X2` (see Figure 13.) Note that the inclusion of a new method invocation may necessitate inclusion of other method invocations because the included method may access some objects whose method invocations are not present in the current GPSPO. We show in Section IV that these issues reduce the problem of constructing a GPSPO to a fixpoint computation on the execution trace of the original program.

III. Formal definition of a PSPO and a GPSPO

To simplify our exposition, we present our algorithm for a simplified Java in which every statement is a call to an instance method on an object and every such method returns a value. Many of Java's feature can be realized through our simplified language as follows:

- We treat reads and writes to fields as calls to get and set methods (and static fields as static get and set methods).
- We define a special object representing every class and treat static methods for a class as instance methods on

```
public class GPSPOInvalid1 {
    public static void main(String[] args) {
        // illegal call to the
        // private constructor
        Problem2 X2 = new Problem2();
        X2.print();
    }
}
```

Fig. 11. An invalid GPSPO of the only execution of the program in Figure 10 with respect to the object `o2`

```
public class GPSPOInvalid2 {
    public static void main(String[] args) {
        Problem2 X1 = new Problem2(9);
        X2.print(); // illegal: X2 undefined
    }
}
```

Fig. 12. A invalid GPSPO of the only execution of the program in Figure 10 with respect to the object `o2`

the corresponding special object.

- We treat **`void`** methods as returning a special value **`void`**.
- We treat constructors as special `<init>` methods that return references to the constructed object.

Our algorithm operates on a *finite trace* of an execution of a program. A trace records the hierarchial sequence of method calls made during the program execution. For each call, it records the object on which the method was called, the parameters to the method, the value returned by the method, and the trace of methods invoked during the method's execution. That is, a trace is a sequence of method invocations M of the form: $V_0 = o.m(V_1, V_2, \ldots, V_n)\{M_1, \ldots, M_n\}$, where where m is the name of the invoked method, o is the object on which m is invoked, the V_i are parameter and returns values, and the M_i are the methods invoked by M.

For example, Figure 14 contains the trace generated by the program in Figure 7. The trace is a single, top-level method invocation:

$$M^1: \ \texttt{void = Problem1.main() } \{ \ M^2, M^4, M^6, M^{12}, M^{16}, M^{17} \ \}$$

Several, of the methods invoked by M^1 further invoke methods themselves (i.e. M^3, M^4, M^6, and M^{12}). Note that field

```
public class GPSPOValid {
    public static void main(String[] args) {
        Problem2 X1 = new Problem2(9);
        Problem2 X2 = X1.getNext();
        X2.print();
    }
}
```

Fig. 13. A valid GPSPO of the only execution of the program in Figure 10 with respect to the object `o2`

```
M1: void = Problem1.main(null) {
M2:   o1 = o1.<init>() {
M3:      void = o1.setVal(0) { }
      }
M4:   o2 = o2.<init>() {
M5:      void = o2.setVal(0) { }
      }
M6:   void = o2.transfer(o1) {
M7:      0 = o2.getVal() { }
M8:      void = o2.setVal(100) { }
M9:      void = o1.decrement(100) {
M10:        0 = o1.getVal() { }
M11:        void = o1.setVal(-100) { }
         }
      }
M12:  void = o1.print() {
M13:     o3 = System.getOut() { }
M14:     -100 = o1.getVal() { }
M15:     void = o3.print(-100) { }
      }
M16:  o3 = System.getOut() { }
M17:  o3.print(``Done'') { }
    }
```

Fig. 14. An execution trace of the program in Figure 7

accesses are treated as calls to getter and setter methods, and that static field `out` is treated as an instance field of special object `System`. Further, primitive operations such as integer addition and subtraction are omitted.

The PSPO for this trace with respect to `o1` is given by the sequence M^2, M^4, M^6, M^{12} and its corresponding GPSPO is shown in Figure 9.

Given a method invocation M of the form $V_0 = o'.m(V_1, V_2, \ldots, V_n)\{M_1 M_2 \ldots M_k\}$, we define the following functions.

Definition III.1. *We define target*(M) *as* o' *(the object on which a method is invoked), method*(M) *as* m *(the name of the method), and return*(M) *as* V_0 *(the object returned by the method).*

For example, in the trace in Figure 14, *target*(M^6) is `o2`, *method*(M^6) is `transfer`, and *return*(M^6) is **`void`**.

Definition III.2. *We define accessedObjects*(M) *as the set* $\{o'\} \cup \{V_i \mid i \in [1, n]$ *and* V_i *is an object* $\} \cup \bigcup_{i=1}^{k}$ *accessedObjects*(M_i)*. We extend the definition to a trace* $\tau = M_1 M_2 \ldots M_k$ *as follows.*

$$\textit{accessedObjects}(\tau) = \bigcup_{i=1}^{k} \textit{accessedObjects}(M_i)$$

That is, *accessedObjects*(M) contains all the objects that have been accessed by the method invocation M. For example, for the trace in Figure 14, *accessedObjects*(M^6) is the set `{o1, o2}`.

Definition III.3. *We define freeObjects*(M) *as the set* $\{o'\} \cup \{V_i \mid i \in [1, n]$ *and* V_i *is an object* $\}$ *if method*(M) *is not* `<init>` *and as the set* $\{V_i \mid i \in [1, n]$ *and* V_i *is an object* $\}$ *if method*(M) *is* `<init>`

For example, for the trace in Figure 14, *freeObjects*(M^2) is the empty set and *freeObjects*(M^6) is the set {`o1, o2`}. In general, *freeObjects*(M) contains the objects that are passed as arguments to the method invocation M. These objects need visible definition in a GPSPO.

Definition III.4. *We say that $M' \sqsubset M$, iff the following holds*

$$(M' == M) \vee \bigvee_{i=1}^{k} (M' \sqsubset M_i)$$

We extend the definition to a trace $\tau = M_1 M_2 \dots M_k$ as follows. We say $M' \sqsubset \tau$ iff there exists a $i \in [1, k]$ such that $M' \sqsubset M_i$.

For example, for the trace in Figure 14, $M^2 \sqsubset M^1$, $M^9 \sqsubset M^1$, $M^9 \sqsubset M^6$. Informally, we say $M' \sqsubset M$, if the invocation of M encloses the invocation of M'.

Definition III.5. ***[PSPO]*** *Given a trace τ and an object o_s, a* path slice per object *(or PSPO) of o_s is a trace $\sigma = M_1 M_2 \dots M_n$ such that*

- **[AllPublic]** *for each $i \in [1, n]$, M_i is a public method invocation,*
- **[ValidSlice]** *for all $i, j \in [1, n]$, if $i < j$, then M_i is invoked before M_j in τ, $M_i \not\sqsubset M_j$, and $M_j \not\sqsubset M_i$,*
- **[Containment]** *for all $M \sqsubset \tau$, if target(M) is o_s then $M \sqsubset \sigma$,*
- **[ValidState]** *for all $i \in [1, n]$ and for all $o \in$ accessedObjects(M_i), if there exists an M such that target(M) is o and M is invoked before M_i in τ, then $M \sqsubset \sigma$,*
- **[AvailableAtUse]** *for all $i \in [1, n]$ and for all $o \in$ `freeObjects`(M_i), there exists a $1 \leq j < i$ such that return(M_j) is o.*

A requirement for PSPO is that all top-level method invocations in the PSPO are public. This ensures that we can create a valid program out of a PSPO. The condition **[AllPublic]** in the definition of PSPO fulfills this requirement. **[ValidSlice]** ensures that a PSPO is a slice of the original trace, i.e. all elements of the PSPO are present in the same order as in the original trace τ. **[Containment]** ensures that all the method invocations on o_s are present in the PSPO. Condition **[ValidState]** ensures that any object o on which a method has been invoked in the PSPO is in the right state. This can be ensured if all the method invocations on o in the original trace is also present in the PSPO. In order to ensure that we can create a program from a trace, we need to make sure that a visible object in a PSPO is defined before it is used. **[AvailableAtUse]** ensures this.

Proposition III.6. *Let σ be the PSPO of the object o_s and trace τ. Then the sequence of methods invoked on o_s in σ is same as the sequence of the methods invoked on o_s in τ.*

The above proposition follows from the conditions **[ValidSlice]** and **[Containment]** in Definition III.5. Informally, the proposition states that all methods invoked on o_s in the original execution are also present in the PSPO. This is one of the primary requirements for a PSPO.

Proposition III.7. *Let σ be the PSPO of the object o_s and trace τ. If M is a method invocation in σ, then the state of all the objects accessed by M is same in both σ and τ.*

The above proposition follows from the condition **[ValidState]** in Definition III.5.

Definition III.8. ***[GPSPO]*** *Given a PSPO, a GPSPO is a program whose exact execution trace is the PSPO.*

IV. GPSPO Computation Algorithm

The GPSPO computation algorithm works in two steps. In the first step, we compute a PSPO. The PSPO is then used to generate GPSPO. We next describe the two steps.

Algorithm 1 PSPO(τ, o_s)

```
1: Input: τ and o_s
2: σ ⇐ M such that M is the last method invoked on o in τ
3: repeat
4:     σ' ⇐ σ
5:     σ ⇐ addAccessedObjects&TheirMethods(τ, σ)
6:     σ ⇐ addDefinition(τ, σ)
7: until σ == σ'
8: return σ
```

A. PSPO Computation Algorithm

The pseudo-code of the algorithm is shown in Algorithm 1. The algorithm essentially computes a fix point trace such that all the conditions in Definition III.5 are satisfied. Specifically, given an execution trace τ and an object o_s created in the trace τ, we start with an initial slice σ that contains the last method invocation on o_s in τ. Then we expand the slice by incorporating other method invocations until we find a slice that satisfies all the conditions of being a PSPO. We add other method invocations in the slice by calling the functions *addAccessedObjects&TheirMethods* and *addDefinition*.

Function *addAccessedObjects&TheirMethods* adds all method invocations on objects that have been accessed by at least one method invocation in σ. In order to add a new method invocation to the slice σ, we call the function *addToSlice*. *addToSlice* ensures that we do not add a private method—if the last argument passed to *addToSlice* is a private method, then we find the immediate enclosing public method invocation in τ that contains the private method invocation and add the public method invocation to the slice. *addToSlice* also ensures that whenever we add a method invocation M to the slice, any other method invocation in the slice is removed if the invocation is contained in M.

Function *addDefinition* ensures that the last condition in Definition III.5 is satisfied. Note that whenever we add a method invocation to the slice using *addToSlice*, one of the conditions in Definition III.5 may get violated. This is because the newly added method invocation may access some objects that we have not considered yet or the method invocation may

Algorithm 2 addAccessedObjects&TheirMethods(τ, σ)

1: **repeat**
2: **let** $\sigma = M_1 M_2 \dots M_n$
3: $\sigma' \Leftarrow \sigma$
4: **for** $i = 1$ to n **do**
5: **for each** $o \in$ *accessedObjects*(M_i) **do**
6: **for each** $M' \sqsubset \tau$ such that *target*$(M') == o$ and M' is invoked before M_i in τ **do**
7: $\sigma \Leftarrow$ *addToSlice*(τ, σ, M')
8: **end for**
9: **end for**
10: **end for**
11: **until** $\sigma == \sigma'$
12: **return** σ

Algorithm 3 addDefinition(τ, σ)

1: **let** $\sigma = M_1 M_2 \dots M_n$
2: **if** $\exists i \in [1, n]$ and $\exists o \in$ *freeObjects*(M_i) such that $\forall j \in [1, i-1]$. *return*$(M_j) \neq o$ **then**
3: **let** $M' \sqsubset \tau$ such that M' is invoked before M_i in τ and $\neg(M' \sqsubset \sigma)$ and *return*$(M') == o$
4: $\sigma \Leftarrow$ *addToSlice*(τ, σ, M')
5: **end if**
6: **return** σ

need definition of some objects. Therefore, we keep iterating the process until we get a valid PSPO.

Theorem IV.1. *Algorithm 1 terminates.*

We provide a sketch of the proof for the above theorem. Essentially, the size of the set $\{M \mid M \sqsubset \sigma\}$ is increased by at least 1 in each iteration (except the last iteration) of the **repeat–until** loops of both Algorithm 1 and Algorithm 2 and the size is upper bounded by the size of the (finite) set $\{M \mid M \sqsubset \tau\}$.

Theorem IV.2. *The slice returned by Algorithm 1 satisfies Definition III.5.*

We again provide a brief proof sketch. The proof requires us to show that σ returned by Algorithm 1 satisfies all the five conditions in Definition III.5. The condition **[AllPublic]** is an invariant and is ensured by lines 1, 2, and 3 of Algorithm 4. The condition **[ValidSlice]** is also an invariant and is ensured by line 5 of Algorithm 4. The condition **[Containment]** is ensured by line 2 in Algorithm 1 and by the Algorithm 2. The condition **[ValidState]** is ensured by Algorithm 2 and the condition **[AvailableAtUse]** is ensured by Algorithm 3.

B. Generating a GPSPO from a PSPO

The second step of the algorithm generates a simple program whose only execution generates the PSPO. Let $\sigma = M_1 M_2 \dots M_n$ be a PSPO. Our GPSPO generating algorithm performs simple syntactic transformations on each statement in σ, prints it to a file and thus creates a compilable Java code. The pseudo code for creating GPSPO from a PSPO is given in algorithm 5. The **Print** method in the algorithm prints the text in the quotes to a file. Before printing, it converts the objects (V_i's) from their internal representation to unique variable names using the \$ macro. For every V_i, $\$V_i$ first checks if

Algorithm 4 addToSlice(τ, σ, M)

1: **if** M is private **then**
2: $M \Leftarrow M'$ where M' is the shortest public method invocation in τ such that $M \sqsubset M'$
3: **end if**
4: **let** $\sigma = M_1 M_2 \dots M_n$
5: $\sigma \Leftarrow M_1 \dots M_k M M_h \dots M_n$, if M is invoked after M_k in τ and $\forall i \in [k+1, h-1]. M_i \sqsubset M$ and $M_h \not\sqsubset M$

Algorithm 5 Algorithm to Compute a GPSPO from a PSPO

1: **Print** `public class GPSPO {`
2: **Print** `public static void main(String[] args){`
3: *declared* $\Leftarrow \emptyset$
4: **for** $i = 1$ to n **do**
5: **if** M_i is of the form $o = o.$<init> $(V_1, V_2, \dots, V_n)\{M^*\}$ **then**
6: **Print** "T \$$o$ = **new** $T(\$V_1, \$V_2, \dots, \$V_n)$;" where T is the type of o.
7: *declared* $=$ *declared* $\cup\{o\}$.
8: **else if** M_i is of the form $V_0 = o.m(V_1, V_2, \dots, V_n)\{M^*\}$ **then**
9: **if** V_0 is an object and $V_0 \notin$ *declared* **then**
10: **Print** "T $\$V_0 = \$o.m(\$V_1, \$V_2, \dots, \$V_n)$;" where T is the type of V_0.
11: *declared* $=$ *declared* $\cup\{V_0\}$
12: **else if** V_0 is an object and $V_0 \in$ *declared* **then**
13: **Print** "$\$V_0 = \$o.m(\$V_1, \$V_2, \dots, \$V_n)$;"
14: **else if** V_0 is a primitive or **`void`** **then**
15: **Print** "$\$o.m(\$V_1, \$V_2, \dots, \$V_n)$;"
16: **end if**
17: **end if**
18: **end for**
19: **Print** `}}`

V_i is a primitive. If so, it returns the value of V_i. Otherwise, it takes the address of V_i, appends the character 'X' in front of it and returns the result. The character is appended to the front of the address to follow the Java naming convention which does not allow a variable name to start with a number.

The algorithm maintains an auxiliary set *declared* (initialized to $\emptyset$ on line 3) that maintains the objects declared so far in the GPSPO. For every $M_i \in \sigma$, the algorithm performs following actions: (1) If M_i is a constructor (line 5), then it prints the appropriate return-type declaration (line 6) and adds the object created by the constructor to the *declared* set. (2) Otherwise, if M_i is any other method invocation, the algorithm first checks if the returned object of the method is already declared (line 9). If not, it prints the appropriate return type and adds the returned object to the *declared* set (line 11).

V. IMPLEMENTATION

We have implemented the GPSPO creation algorithm in a prototype tool for Java. We instrument a Java program to insert hooks before and after every method call. When the instrumented program is executed, these hooks generate the trace of the execution. Specifically, for every method call, we record the object on which the method is invoked, the parameters to the method and the return value of the method. For each object, we record a unique identifier representing that object, and for each primitive type (`int`,

Program Name	|PSPO|/ |Trace|	unique(PSPO)/ unique(Trace)	Average |GPSPO|	# of GPSPOs	# of classes
grep	0.29%	36.7%	2.92	12	11
awk	0.03%	4.3%	2.27	283	15
JNotePad	0.34%	7.8%	3.30	27	14
Search	0.48%	63.5%	2.33	56	5
jTidy	0.66%	7.7%	29.95	1435	55
jazzy	1.25%	0.3%	21.02	3890	15

TABLE I
STATISTICS ON GPSPOS COMPUTED FOR SEVERAL BENCHMARKS

`boolean`, `float`, `byte`, `short`, `char`, `long`, `byte`, `double`) and `java.lang.String`, we record the concrete value. At the end of the execution, the trace is analyzed either to generate GPSPOs of all objects created during the execution or the GPSPOs of all objects of a particular class created during the execution.

Note that unlike dynamic program slicing where all statements of a program needs instrumentation, we only instrument the statements that invoke a method. As such we generate shorter traces and use far less memory than traditional dynamic program slicing. Algorithm 1 for PSPO computation has many loops and seems to have huge runtime complexity. However, an implementation that traverses the trace in reverse direction would ensure that Algorithm 2 has a runtime complexity of $O(n)$, where n is the length of the trace. Moreover, one can show the **repeat-until** loop in Algorithm 1 runs for at most $O(n)$ iterations. Therefore, the algorithm has a total complexity of $O(n^2)$. In practice, however, we observed that only a few iterations of this **repeat-until** loop were needed for generating the PSPOs.

The algorithm to create a GPSPO from a PSPO requires us to know the address of an object, so that each object is identified by unique numeral. In case of Java, the address of an object cannot be obtained directly. Therefore, we maintain a `WeakIndentityHashMap` that maps each object, as it is created, to a unique id. We use this unique id for each object as its address.

VI. EVALUATION

In this section, we describe our efforts to experimentally evaluate the utility of PSPOs and GPSPOs for program debugging, for creating test harnesses, and for discovering class usage.

A. GPSPOs for Debugging

Suppose a programmer wishes to debug the interaction between a program and a particular program object. For example, in some particular program execution, a data structure may throw an unexpected exception—is this exception due to a bug in the data structure itself, or due to the program violating the contract of the structure? How can a GPSPO of the execution of the data structure aid in debugging this problem?

To investigate such an issue, a programmer might add additional assertions or logging code, or might step through parts of the buggy execution, inspecting the target data structure and arguments passed to its methods. We argue that much of the same analysis could be easier and more effective if done instead on the GPSPO for this target data structure object. Such a path slice is an executable program and performs all of the same updates on the object as the original execution. However, the GPSPO's execution ideally eliminates much of the original execution that does not directly affect the target object, including unnecessary methods and control flow.

It is difficult to directly measure the benefit a GPSPO could provide for a particular debugging task. However, we believe the following properties of GPSPOs can help indicate their debugging utility:

- *Length of Trace of GPSPO.* The trace of a GPSPO (i.e. its PSPO) is a subset of the original execution trace. The shorter a GPSPO's trace, the more unnecessary parts of the original execution have been sliced away. This may lead to easier debugging by enabling a programmer to focus on the simpler execution of the GPSPO, rather than the full original execution. Thus, in our experiments, we measure the ratio of the length of our PSPO's to the lengths of the original execution.
- *Unique Statements in PSPO.* A second measure of the complexity of a GPSPO is the number of unique program statements in the trace of a GPSPO. This is a rough measure of how much of the original static program source is executed by the GPSPO. Thus, we also measure the ratio of the number of unique statements in each GPSPO's trace to the number of unique statements executed by the original trace. This indicates of what fraction of the static program exercised by the original execution trace is sliced away in the GPSPO.

In order to explore the debugging utility of GPSPOs, we generated GPSPOs for several benchmark programs. Specifically, four open-source Java utilities—`grep` and `awk`, implemented on top of Apache's regular expression library, the `jTidy` HTML-cleaning tool, and the `jazzy` spell checker—and two closed-source applications—the `JNotePad` text editor and `Search`, a command-line web search engine. For each of these benchmarks, we generated several typical execution

traces and computed a GPSPO for every object in each trace.

For each benchmark, we averaged over all GPSPOs and their PSPOs: (1) the ratio of the length of the PSPO to the length of the original trace, (2) the ratio of the number of unique statements in the PSPO to the number of unique statements in the original trace, and (3) the length of the GPSPO. These quantities are tabulated in Table I. Further, Table I gives the total number ofGPSPOs generated for each benchmark and the number of non-primitive classes seen in each benchmark.

Table I shows that, for each of these benchmarks, a GPSPO removes, *on average*, more than 98% of the original execution. Further, on average anywhere from 36% and to 90%+ percent of the unique static program statements in the original trace are removed. We believe these initial results are promising: for these benchmarks, the average GPSPO is significantly simpler than the full trace from which it is derived. Thus, some debugging could be performed on these GPSPOs rather than on the full original programs.

B. GPSPOs for Discovering Class Usage

Large object-oriented software systems and libraries often have complex APIs, which can make discovering the usage of a particular class difficult or tedious. There are several static approaches for automatically synthesizing small code snippets to demonstrate the usage of a class. For example, Prospector [19] is a system which takes a user query of the form $T_{in} \rightarrow T_{out}$, where T_{in} is the source object and T_{out} is the destination object (i.e. object to be synthesized) and generates a sequence of method calls that returns T_{out} starting with T_{in}. Note that Prospector is a static technique that generates code based on static method signatures.

GPSPOs can be seen as a dynamic alternative to Prospector or other such static systems, where a sequence of method calls is generated dynamically by observing execution traces of a program, rather than statically from method signatures or program source code.

In order to measure the utility of GPSPOs for discovering class usage, we ran our GPSPOs computation on all objects in several executions of `jtidy`, a Java tool for cleaning up HTML. One of the authors, having familiarity with the `jtidy` API, manually generated queries of the form $T_{in} \rightarrow T_{out}$ for three commonly used classes in `jtidy`. GPSPOs are complete programs and as such do not provide any facility to discover a sequence of method calls that convert an object of class T_{in} to an object of class T_{out}. Hence, we manually collected the GPSPO source files that contained both T_{in} and T_{out} classes. For each query, we then picked the source file containing the fewest number of lines and manually examined its suitability as a code snippet highlighting how to create an T_{out} object given a T_{in}.

The results for the three queries are shown in Table II. The first two columns of the table give the T_{in} and T_{out} classes for each query. The third column gives the exact lines of code from the source file that convert an object of type T_{in} to the object of type T_{out}. Finally, the last column gives the number of lines of code in the GPSPO that was selected as described above.

Test Name	**| GPSPO |**	**branches covered by GPSPO**	**branches covered by Harness**	**Number of false positives**
TreeMap	7	22	43	0
List	2	1	1	0
Vector	4	3	4	1
LinkedList	7	30	44	0

TABLE III
TEST HARNESS QUALITY USING GPSPOS

These results show that on all three queries, the required code to convert an object of class T_{in} to an object of class T_{out} was found in the minimal GPSPO we examined. Note that some of these transformations are fairly hard to achieve and involve several intermediate objects. For example, in the second query, to generate an object of class `Node`, one needs to create a `FileInputStream`, a `StreamInImpl`, and a `Lexer` object, and pass them to the `parseDocument` method along with the `Configuration` object. It could be quite hard to discover this usage using documentation alone. Our GPSPOs, however, were able to correctly instantiate these objects before passing them to the `parseDocument` method. Moreover, the GPSPOs contained the relevant sequence without containing too much extraneous code (27 lines in the worst case).

These preliminary results suggest that GPSPOs can be used by programmers to help dynamically discover class usage examples. These techniques could perhaps form the basis for a more effective and automatic system for dynamically discovering class usage.

C. GPSPOs for Test Harness Generation

A test harness for a class is often constructed as a closed program that non-deterministically calls all legal sequences of methods of the class with legal inputs for method parameters [10], [24]. Generating an ideal test harness for a class that only permits legal behaviors and prohibits illegal behaviors is often tedious. We argue that GPSPOs could be used to generate good quality test harnesses, i.e. test harnesses that allow most legal behaviors (i.e. give good coverage) and prohibit most illegal behaviors (i.e. avoid false positives.) We convert a GPSPO into a test harness by replacing all primitive values with inputs (see Figure 6 for an example.) Note that a GPSPO always exhibits a legal behavior of the class, but it only shows a single behavior (or a single execution path.) By replacing the constant primitives in the GPSPO by inputs, we introduce more behaviors.

In our experiments, we evaluate whether the conversion of GPSPOs to test harnesses: (1) increases the number of legal behaviors (i.e. increases test coverage), while (2) not introducing too many illegal behaviors (i.e. avoids false positives.) In order to evaluate the first criterion, we use the test

$\mathbf{T_{in}}$	$\mathbf{T_{out}}$	Code for transforming $\mathbf{T_{in}} \rightarrow \mathbf{T_{out}}$ obtained using GPSPOs	Number of lines in the file
java.lang.String	org.w3c.tidy. Attribute	`AttrCheckImpl.CheckUrl X10 = new AttrCheckImpl.CheckUrl();` `Attribute X38 = new Attribute("id", 28, (AttrCheck)X10);`	8

org.w3c.tidy.Configuration	org.w3c.tidy.Node	`Configuration X3 = new Configuration();`
		`FileInputStream X574 = new FileInputStream("test.html");`
		`StreamInImpl X587 = new StreamInImpl(X574, 1, 4);`
		`Lexer X588 = new Lexer((StreamIn)X587, X3);`
		`Node x594 = ParserImpl.parseDocument(X588);`
			32

org.w3c.tidy.ParserImpl. ParseInline	org.w3c.tidy.Dict	`ParserImpl.ParseInline X171 = new ParserImpl.ParseInline();` `Dict X242 = new Dict("dt", 31, 294976, (Parser)X171, null);`	28

TABLE II
GPSPOs FOR CLASS USAGE DISCOVERY

generation tool CUTE [23] on the test harness and check if CUTE increases the test coverage over the coverage attained by simply running the GPSPO. In order to evaluate the second criterion, we record the number of exceptions thrown by the test inputs generated by CUTE. Note that an exception indicates that we have used the class illegally, provided that the class has no bugs. This requirement is met by picking well-known classes from the java.util framework that are well tested, relatively bug-free, and are only expected to throw exceptions on illegal usage. In our experiments, we picked four random GPSPOs on objects from the java.util Collections framework and transformed them into test harnesses.

Table III reports the results. Column 1 gives the name of the test benchmark. Column 2 reports the size of the GPSPO from which we have created the test harness. Column 3 gives the branch coverage that we achieve if we simply execute the GPSPO. Column 4 gives the branch coverage that we achieve if we test the harness with CUTE. Column 5 reports the number of false positives (or illegal test inputs) generated by CUTE. The results show that test harnesses created using these GPSPOs gave only one false positive on one of the benchmarks. Moreover, the branch coverage obtained using GPSPOs was improved by almost 44% on average by converting to test harnesses. For one harness, we observed an improvement of almost 100% in the branch coverage. This suggests that we can increase branch coverage by a reasonable factor by creating test harnesses using GPSPOs.

These preliminary experiments suggest that GPSPOs can be good starting points for creating test harnesses exhibiting few false positives and providing additional test coverage.

VII. RELATED WORK

Closely related work includes a variety of novel techniques for automatically carving unit or subsystem tests from recorded program executions [22], [20], [6], [15]. Most closely related is the work of Jorde et al. [15] on A-DUTs, in which method calls are recorded during the execution of a system test and the dependencies between these calls are computed. These dependencies are used to replay only the sequence of methods invoked on some specific object, as well as the methods to construct any needed parameters. These techniques, however, typically involve additional replay machinery which yield tests robust against code changes, but at the cost of additional complexity. GPSPOs, on the other hand, are simple programs which can be directly compiled and executed. They are useful not just in constructing program tests, but in program debugging and understanding.

Our technique is closely related to program slicing [27] (both dynamic [16], [2], [1], [17], [9], [5], [30], [26] and static [21], [13], [12], [8], [18]). Similar to program slicing, GPSPO also comes up with a smaller program. However, in case of a program slicing, the output is the original program with some statements removed from the program. In contrast, a GPSPO is a simple program containing only method calls, and containing no branching or looping statements. A program slice ensures that we get a minimal slice of the program so that the statement of interest gets executed under similar state as in the original program. In contrast, a GPSPO ensures that for a given object, the same methods get invoked on the object in the same order and under the same state as in the original program execution. A program slice contains all dependent statements, whether they are operations on primitives or objects. We remove any operation on primitives from a GPSPO. Note that this does not mean that whatever methods a GPSPO calls are devoid of operations on primitives—the code of those methods remain exactly same as the original program.

Jhala et al. [14] proposed path slicing to make software model checking efficient. Path slicing takes as input a possibly infeasible static execution path to a target location, and eliminates all the operations that are irrelevant towards the reachability of the target location. Although the two technique have similar sounding names, they are different from each other both in respect to their goals and their methodology—path slicing operates on a static (and possibly infeasible) path, whereas path slicing per object operates on a real execution path.

Prospector [19] helps programmers write API client code

more easily by synthesizing jungloid code fragments automatically given a simple query that describes that desired code in terms of input and output types. GPSPOs can also be used for this purpose in many situations. Being static in nature, Prospector can come up with many suggestions and they may not be precise in complex situations. Being dynamic in nature, GPSPOs can only come up with suggestions that they see in an execution, but they are precise as they are carved out from real execution traces.

Specification mining [4], [29] and interface synthesis [3], [11], [28] are related techniques which try to infer the usage of a class API. Specifically, these techniques either statically or dynamically compute a finite state machine denoting the legal method call sequences on an object of a given class. In order to infer such finite state machines, they either look at client programs using the class or look at the implementation of the class. Unlike GPSPOs, these techniques may not be useful in debugging a particular execution. Moreover, if a method on an object takes objects of other classes as arguments or uses them internally, then these techniques cannot help to figure out the right state of these objects.

VIII. Conclusion

We introduced the concept of generator for path slice per object (GPSPOs.) We provided an algorithm to compute GPSPOs and an implementation of the algorithm. We believe that path slice per object is an important concept that could help us to perform better debugging, testing, and understanding class APIs.

References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. An execution-backtracking approach to debugging. *IEEE Software*, 8(3):21–26, 1991.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proc. of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 246–256. ACM Press, 1990.

[3] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *Proc. of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 98–109. ACM, 2005.

[4] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1):4–16, 2002.

[5] Árpád Beszedes, T. Gergely, Z. M. Szabó, J. Csirik, and T. Gyimothy. Dynamic slicing method for maintenance of large c programs. In *Proc. of the Fifth European Conference on Software Maintenance and Reengineering*, page 105. IEEE Computer Society, 2001.

[6] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *Proc. of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 253–264. ACM, 2006.

[7] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[8] R. Gupta and M. L. Soffa. Hybrid slicing: an approach for refining static slices using dynamic information. In *Proc. of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 29–40. ACM Press, 1995.

[9] T. Gyimóthy, Árpád Beszédes, and I. Forgács. An efficient relevant slicing method for debugging. *SIGSOFT Softw. Eng. Notes*, 24(6):303–321, 1999.

[10] T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *Proc. of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 31–40. ACM Press, 2005.

[11] T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *Proc. of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 31–40. ACM, 2005.

[12] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[13] J. Hwang, M. Du, and C. Chou. Finding program slices for recursive procedures. In *Proceedings of the Twelfth International Computer Software and Applications Conference (COMPSAC 88)*, pages 220–227. IEEE, Oct. 1988.

[14] R. Jhala and R. Majumdar. Path slicing. *SIGPLAN Not.*, 40(6):38–47, 2005.

[15] M. Jorde, S. Elbaum, and M. Dwyer. Increasing test granulatiry by aggregating unit tests. In *ASE '08: Proceedings of the 23th IEEE/ACM international Conference on Automated software engineering*, New York, NY, USA, 2008. ACM.

[16] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[17] B. Korel and J. Rilling. Application of dynamic slicing in program debugging. In *Automated and Algorithmic Debugging*, pages 43–58, 1997.

[18] L. Larsen and M. J. Harrold. Slicing object-oriented software. In *Proc. of the 18th international conference on Software engineering*, pages 495–505. IEEE Computer Society, 1996.

[19] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. *SIGPLAN Not.*, 40(6):48–61, 2005.

[20] A. Orso and B. Kennedy. Selective capture and replay of program executions. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.

[21] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proc. of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184. ACM Press, 1984.

[22] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for java. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 114–123, New York, NY, USA, 2005. ACM.

[23] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*. ACM, 2005.

[24] N. Tillmann and W. Schulte. Parameterized unit tests. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 253–262, 2005.

[25] W. Visser, C. Pasareanu, and S. Khurshid. Test Input Generation with Java PathFinder. In *Proceedings of ACM SIGSOFT ISSTA'04*, pages 97–107, 2004.

[26] T. Wang and A. Roychoudhury. Hierarchical dynamic slicing. In *Proc. of the 2007 international symposium on Software testing and analysis*, pages 228–238. ACM Press, 2007.

[27] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[28] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proc. of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 218–228. ACM, 2002.

[29] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal api rules from imperfect traces. In *Proc. of the 28th international conference on Software engineering*, pages 282–291. ACM, 2006.

[30] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *Proc. of the 25th International Conference on Software Engineering*, pages 319–329. IEEE Computer Society, 2003.