# Dynamic Data Flow Testing of Object Oriented Systems

Giovanni Denaro[†], Alessandro Margara[*], Mauro Pezzè[*†] and Mattia Vivanti[*]
[*]Università della Svizzera italiana (USI), Lugano, Switzerland
Email: {alessandro.margara, mauro.pezze, mattia.vivanti}@usi.ch
[†]University of Milano Bicocca, Milano, Italy
Email: denaro@disco.unimib.it

***Abstract*—Data flow testing has recently attracted new interest in the context of testing object oriented systems, since data flow information is well suited to capture relations among the object states, and can thus provide useful information for testing method interactions. Unfortunately, classic data flow testing, which is based on static analysis of the source code, fails to identify many important data flow relations due to the dynamic nature of object oriented systems. In this paper, we propose a new technique to generate test cases for object oriented software. The technique exploits useful inter-procedural data flow information extracted dynamically from execution traces for object oriented systems. The technique is designed to enhance an initial test suite with test cases that exercise complex state based method interactions. The experimental results indicate that dynamic data flow testing can indeed generate test cases that exercise relevant behaviours otherwise missed by both the original test suite and by test suites that satisfy classic data flow criteria.**

## I. Introduction

Software testing is an important verification and validation activity to reveal program failures. Test cases may be generated by sampling the execution space either randomly or aiming at specific objectives that derive from the expected behavior, the program structure or some information about faults [1].

Data flow testing is a particular form of testing that identifies data flow relations as test objectives [2]. Data flow testing has been extensively studied since the early seventies with ambivalent results. Some studies indicate that data flow information is very useful for exercising corner cases [3], while other studies do not acknowledge significant advantages of data flow testing over simple branch coverage [4]. Recently, data flow testing has attracted new interest in the context of testing object oriented systems, since data flow information can capture the relations among the object states particularly well [5], [6], [7]. However, despite the current encouraging results, there are no conclusive studies about the validity of data flow testing in the context of object oriented systems yet.

In a recent work, we noticed that classic data flow testing, based on static analysis of the source code, misses a lot of important data flow information [8]. The data reported in [8] indicate that static analysis can miss up to 96% of the data flow information dynamically observed during program execution, out of which over 60% of the missed information depends on dynamic binding and polymorphic relations and another 10% depends on the difficulty of managing complex data structures like arrays.

In this paper, we present DynaFlow, a new approach to generate test cases that exploits useful data flow information that is difficult to compute on the source code. The approach is not yet another data flow coverage metrics, but a constructive way of identifying useful test objectives to generate new test cases, and as such it is a good complement of existing test case generation techniques.

Although DynaFlow works for general programs, we propose it for interprocedural testing of object oriented programs. In fact, the object oriented paradigm encourages programs that consist of stateful classes, whose state is defined as dynamically allocated data structures that are accessed by methods of dynamically bound objects. This high degree of dynamism is the primary source of limitations of classic data flow approaches that can be overcome with a dynamic approach as the one proposed in this paper.

DynaFlow is based on a form of data flow analysis that we refer to as *dynamic data flow analysis*, and that consists of executing the program with one or more test cases and computing data flow information on the resulting execution traces. The collected information represents data flow relations that derive from the interplay between the static structure of the program and the dynamic evolution of the system, and thus cannot be identified with classic data flow approaches. The approach proposed in this paper exploits the dynamic data flow information computed on an initial set of test cases to infer new data flow relations and consequently new test objectives.

Being inferred on dynamically computed data, the new data flow relations both include important information on the execution state that cannot be captured statically, and suffer less from the infeasibility problem than the data flow relations computed with static analysis. We feed the new test objectives into a test case generation tool to produce new test cases, and iterate through the approach until we cannot identify new test objectives to be exercised.

We evaluated the approach by estimating the ability of revealing new faults, using the number of killed mutants as proxy measure of the fault revealing capability [9]. Our experimental results indicate that our approach can significantly increase the number of killed mutants compared to the starting test suite. This indicates that with our approach we can increase the effectiveness of the available test suite by addressing a larger set of potential faults. We also notice that our approach kills more mutants than much larger test suites computed randomly. This indicates that the increased number of killed mutants does not depend on the mere increased size of the test suite, but on the execution of corner cases difficult to identify by simply executing more test cases.

This paper contributes to the state of art by (1) defining a new data flow analysis approach that we refer to as dynamic data flow analysis, and that identifies important information that classic data flow approaches would miss, (2) introducing an iterative approach to augment an existing test suite by exploiting the data flow information computed dynamically during the execution of the original test suite, (3) providing an initial set of experimental results that indicate that the new technique produces test cases that can reveal faults not easily identifiable with other techniques, and (4) opening new opportunities to exploit data flow information for increasing the effectiveness of test suites.

This paper is organized as follows. Section II illustrates with an example the limitations of using static data flow analysis to infer data flow testing objectives, and motivates the intuition behind the use of dynamic data flow analysis as in the approach proposed in this paper. Section III describes our novel approach for test case generation based on dynamic data flow analysis. Section IV discusses the experimental results that support the validity of the approach. Section V reviews the main related work and indicates the differences with respect to the approach proposed in this paper. Section VI summarizes the results of this paper and points to open research questions.

## II. Motivating Example

This section provides some examples to show that the structure of object oriented programs can hamper the effectiveness of static analysis in identifying the data flow relations of a program. Next, it introduces the intuition behind the use of dynamic data flow analysis to identify useful data flow information more effectively than with static analysis.

The most common data flow testing technique requires executing *def-use pairs*. A def-use pair associates a *definition* with a *use*, provided that there exists a *def-clear path* in-between. A definition corresponds to a statement that changes the value of a program variable. A use corresponds to a statement that requires the value of a variable. A def-clear path is a program path that traverses the definition and then reaches the use of the same variable without changing the value of the variable in-between. When applied to object oriented programs, the def-use testing criterion captures the (state-based) interactions between the methods that define and use the state variables of the classes.

We refer to static data flow analysis both with and without alias analysis to identify def-use pairs. Data flow analysis without alias analysis (hereafter *DFA1*) trades completeness for scalability, and is the technique commonly used in most data flow testing approaches [2], [10], [11]. Data flow analysis integrated with may-alias static analysis [12], [13] (hereafter *DFA2*) computes a conservative over approximation of the set of possible def-use pairs, possibly with awareness of polymorphism and dynamic binding. DFA2 produces accurate results with increased, sometimes impractical, complexity.

```
1  class L0 extends Level{
2     boolean v0 = false;
3     Level sub = LevelFactory.makeLevel(1);
4     void doA(){
5        sub.doA();
6        v0 = true;
7     }
8     boolean doB(){
9        boolean ok = true;
10       if(v0) ok = sub.doB();
11       assert ok; //Bug: this assertion can fail!
12       return ok;
13    }
14 }
15 class L1 extends Level{
16    boolean v1 = false;
17    Level sub = LevelFactory.makeLevel(2);
18    void doA() {
19       if(v1) sub.doA();
20    }
21    boolean doB() {
22       v1 = true;
23       boolean ret = sub.doB();
24       return ret;
25    }
26 }
27 class L2 extends Level{
28    boolean v2 = false;
29    void doA() {
30       v2 = true;
31    }
32    boolean doB() {
33       if(v2) return false;
34       return true;
35    }
36 }
37 class L3 extends Level{
38    boolean v3 = false;
39    Level sub = LevelFactory.makeLevel(4);
40    void doA() {
41       if(v3) sub.doA();
42       v3 = sub.doB();
43    }
44    boolean doB() {
45       if(v3) sub.doA();
46       v3 = sub.doB();
47       return true;
48    }
49 }
50 abstract class Level{
51    abstract void doA();
52    abstract boolean doB();
53 }
54 class LevelFactory{
55    static Level makeLevel(int selector){
56       switch(selector){
57       case 0: return new L0();
58       case 1: return new L1();
59       case 2: return new L2();
60       case 3: return new L3();
61       //case 4, case 5, ...
62       default: return null;
63       }
64    }
65 }
```

Fig. 1: A sample Java program

Let us consider the (Java) object oriented program in Fig. 1. The class under test `L0` (line 1) includes the state variable `sub` (line 3) of class `L1` (line 15). Class `L1` includes the state variable `sub` (line 17) of class `L2` (line 27). The class `L3` (line 37) is unrelated to the class `L0`. The class `Level` (line 50) defines the interface of the classes `L0`, `L1`, `L2` and `L3`. The class `LevelFactory` (line 54) implements the common design pattern factory: It returns new instances of the

TABLE I: A sample definitions and uses computed with DFA2 for the methods of class `L0`

| Pair | Variable [may-alias object type] | Def at | Through call chain | Use at | Through call chain |
|---|---|---|---|---|---|
| p1 | L0.v0 | 6 | L0.doA | 10 | L0.doB |
| p2 | L0.sub[L1].v1 | 22 | L0.doB→L1.doB | 19 | L0.doA→L1.doA |
| p3 | L0.sub[L1].sub[L2].v2 | 30 | L0.doA→L1.doA→L2.doA | 33 | L0.doB→L1.doB→L2.doB |
| p4 | L0.sub[L1].sub[L3].v3 | 42 | L0.doA→L1.doA→L3.doA | 41 | L0.doA→L1.doA→L3.doA |
| p5 | L0.sub[L1].sub[L3].v3 | 42 | L0.doA→L1.doA→L3.doA | 45 | L0.doB→L1.doB→L3.doB |
| p6 | L0.sub[L1].sub[L3].v3 | 46 | L0.doB→L1.doB→L3.doB | 41 | L0.doA→L1.doA→L3.doA |
| p7 | L0.sub[L1].sub[L3].v3 | 46 | L0.doB→L1.doB→L3.doB | 45 | L0.doB→L1.doB→L3.doB |
| p8 | L0.sub[L1].sub[L3].sub[L2].v2 | 30 | L0.doA→L1.doA→L3.doA→L2.doA | 33 | L0.doB→L1.doB→L3.doB→L2.doB |
| p9 | L0.sub[L2].v2 | 30 | L0.doA→L2.doA | 33 | L0.doB→L2.doB |
| p10 | L0.sub[L3].v3 | 42 | L0.doA→L3.doA | 41 | L0.doA→L3.doA |
| p11 | L0.sub[L3].v3 | 42 | L0.doA→L3.doA | 45 | L0.doB→L3.doB |
| p12 | L0.sub[L3].v3 | 46 | L0.doB→L3.doB | 41 | L0.doA→L3.doA |
| p13 | L0.sub[L3].v3 | 46 | L0.doB→L3.doB | 45 | L0.doB→L3.doB |
| p14 | L0.sub[L3].sub[L1].v1 | 22 | L0.doB→L3.doB→L1.doB | 19 | L0.doA→L3.doA→L1.doA |
| p15 | L0.sub[L3].sub[L1].sub[L2].v2 | 30 | L0.doA→L3.doA→L1.doA→L2.doA | 33 | L0.doB→L3.doB→L1.doB→L2.doB |
| p16 | L0.sub[L3].sub[L2].v2 | 30 | L0.doA→L3.doA→L2.doA | 33 | L0.doB→L3.doB→L2.doB |
| ... | ... | ... | | ... | |

TABLE II: Definitions and uses dynamically revealed against the execution of method sequences

| Test sequence | Observed definitions | Observed uses | Inferred pairs* | Covered pairs* |
|---|---|---|---|---|
| L0.doA(); | L0.v0 | L0.sub.v1 | | |
| L0.doB(); | | L0.v0 | p1 | |
| L0.doA();L0.doB(); | L0.sub.v1 | L0.sub.sub.v2 | p2 | p1 |
| L0.doA();L0.doB(); L0.doA(); | L0.sub.sub.v2 | | p3 | p2 |
| L0.doA();L0.doB(); L0.doA();L0.doB(); | | | | p3 |

* def-use pairs identified as in Table I.

type `Level` by instantiating some of the subtypes according to a selector value passed as parameter.

The analysis DFA1 produces poor insights on the behavior of class `L0`. The only information obtained with DFA1 is that method `L0.doA()` may define variable `L0.v0` at line 6, and method `L0.doB()` may use this variable at line 10. The def-use criterion can be satisfied with a test suite that includes a single test case that calls these two methods in sequence and exercises their set-get behavior. Unfortunately, DFA1 does not reveal the effects of the calls of methods `sub.doA()` (line 5) and `sub.doB()` (line 10), because the static binding of these methods through the variable `sub` of class `Level` does not link to any concrete implementation of the methods.

The analysis DFA2 produces more accurate results than DFA1 thanks to the integration with may-alias analysis. The may-alias analysis infers the set of objects that may bind to the variables used throughout the program. We refer to this set of objects as the *may-alias set* of a variable at a program point. For instance, the may-alias set of variable `L0.sub` used at the call-point `sub.doA()` (line 5) consists of the objects instantiated within the method `LevelFactory.makeLevel()` at lines 57, 58, 59 and 60. For the program in Fig. 1, all call points depend on the return values of the method `LevelFactory.makeLevel()`. Therefore, all call points share the same may-alias set.

The analysis DFA2 exploits the may-alias sets to compute a larger and more complete set of def-use pairs than DFA1. The results of DFA2 are partially listed in Table I. Each row lists a def-use pair composed of an identifier (column *Pair*), the variable name (column *Variable*), the line of code that corresponds to the variable definition within the call chain that may lead to the definition (column *Def*) and the line of code that corresponds to the use within the corresponding call chain (column *Use*). The variable name indicates also the associations between class fields and objects according to the may-alias sets. For example the pair $p2$ refers to the variable `L0.sub.v1` because the field `L0.sub` can be alias of an object of type `L1`.

A test suite that covers the def-use pairs identified with DFA2 outperforms the single test case that satisfies the criterion according to def-use pairs computed with DFA1. In fact, DFA2 augments the pair $p1$ identified with DFA1 with new pairs that correspond to relevant object interactions. The pairs $p2$ and $p3$ capture the effects of the calls of methods `sub.doA()` (line 5) and `sub.doB()` (line 10) that are missed by DFA1. In particular, when executing a test case that covers $p3$, variable `L0.sub.sub.v2` (line 30) is assigned value `true`, and method `L2.doB()` returns `false` (line 33), thus revealing the violation of the assertion at line 11.

DFA2 identifies many pairs, but only pairs $p1$, $p2$ and $p3$ of Table I are feasible. All the other pairs listed in the table and many others that are not listed there are infeasible. They derive from alias relations that are inferred statically by the conservative may alias analysis and that do not occur at runtime. Thus, DFA2 produces important useful information missed by DFA1, but also a lot of false positives that can cause the divergence of the testing effort.

The example witnesses a well known phenomenon: the data flow information computed without alias information misses important test objectives, while an expensive conservative analysis may hide useful information in a lot of false positives.

The technique proposed in this paper computes relevant test objectives by relying on what we call *dynamic data flow analysis*, and generates test cases that satisfy such goals. Dynamic data flow analysis computes data flow information on execution traces. In this way, it captures the aliasing relations that manifest at runtime, thus both mitigating the under approximation of DFA1 and avoiding the many false positives that result from DFA2.

With reference to the example discussed in this section, Table II reports the information computed with the dynamic analysis technique that we present later in the paper, as an example of how dynamic data flow analysis produces important and accurate information for generating test cases. Each row reports the information gathered dynamically in one execution, that is: the executed test sequence (column *Test sequence*), the definitions and uses observed on the execution trace (columns *Observed definitions* and *Observed uses*, respectively), the def-use pairs that can be inferred based on the observed information (column *Inferred pairs*) and the def-use pairs covered in the execution (column *Covered pairs*). At each run, the technique infers new pairs that become test objectives for the next runs.

In the first two runs, the test cases simply call the two methods of class `L0` in isolation, respectively. Executing `Lev0.doA()` reveals the definition of variable `L0.v1` and the use of variable `L0.sub.v1`. Executing `Lev0.doB()` reveals the use of variable `L0.v0`. While the information produced in the first run does not identify any pair, the additional information from the second run allows us to infer the pair $p1$ (Table I). The next runs cover the inferred and not-yet-covered pairs incrementally. The third test case covers pair $p1$, reveals a new definition and a new use, and infers a new pair to be covered $p2$. The fourth test case covers $p2$, reveals a new definition and infers the pair $p3$, which is covered by the last test case reported in the table. Since the last test case does not reveal new definitions or uses, the process terminates. We can see that in this case the technique identifies all the feasible and important data flow relations without false positives, and guides the incremental generation of test cases to find the failure. The experimental evaluation in Section IV confirms this results.

## III. Dynamic Data Flow Testing

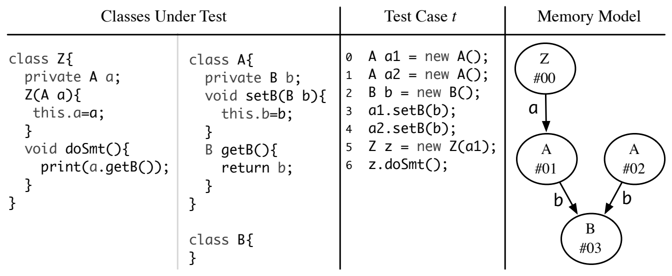
This section presents DynaFlow, a new approach for exploiting data flow information to generate test cases. DynaFlow iteratively improves an original test suite with new test cases. At each iteration DynaFlow (*A*) dynamically analyzes the program executed with the current test suite to compute dynamic data flow information, (*B*) statically combines the data flow information to identify new test objectives, and (*C*) generates new test cases that satisfy the new objectives. The new test cases are added to the current test suite for the new iteration. DynaFlow terminates when either step *B* does not identify new test objectives or the execution budget is over.

Below, we describe these three steps in details. Section III-A presents our dynamic data flow analysis, Section III-B focuses on the static approach used to compute test objectives, and Section III-C discusses the automated generation of test cases from such goals.

### *A. Dynamic Data Flow Analysis*

In the first step, DynaFlow computes data flow information of the program under analysis. As suggested by the example in Section II, dynamic data flow analysis can identify useful information to exercise cases that may lead to subtle failures.

DynaFlow targets method interactions. Since methods interact primarily through the class state, we focus on data flow relations between the *class state variables* that comprise the



```
class Z{
  private A a;
  Z(A a){
   this.a=a;
  }
  void doSmt(){
    print(a.getB());
  }
}
```

```
class A{
  private B b;
  void setB(B b){
    this.b=b;
  }
  B getB(){
    return b;
  }
}

class B{
}
```

```
0  A a1 = new A();
1  A a2 = new A();
2  B b = new B();
3  a1.setB(b);
4  a2.setB(b);
5  Z z = new Z(a1);
6  z.doSmt();
```

Fig. 2: Example of memory model built by executing a test case $t$. The model represents the internal state of the memory after the execution of line 6 of test $t$.

*class state*. Object oriented programs define class states as sets of *class fields* that may be of primitive or structured types. In this latter case, the fields refer to objects that in turn include other fields. DynaFlow represents the class state as a set of *class state variables*: Each class state variable univocally identifies a chain of field signatures that characterizes a field that is part of the class state at some nesting level. For example, class `Z` in Fig. 2 includes a field `a` of type `A`, and class `A` contains a field `b` of type `B`. Intuitively, the state of class `Z` includes the field `A.b` nested in the object `Z.a`. DynaFlow expresses this relation by saying that class `Z` has a state variable `Z.a.b`.

DynaFlow focuses on relations between class state variables that depend on dynamic conditions, such as dynamic bindings and aliases between (nested) class fields and objects created at runtime, and that can be computed by means of dynamic data flow analysis. DynaFlow relies on a proper extension of DReaDs, a dynamic data flow analyzer we developed in our lab [8]. DReaDs computes reaching definitions and reachable uses of the class state variables dynamically on the program traces. Next, we introduce DReaDs, focusing on the novel extensions introduced in this paper. DReaDs is composed of three main components: (1) a component that maintains a memory model to map both memory-load and memory-store events to the involved class state variables, (2) a component that exploits the memory model to monitor the propagation of definitions and the reachability of uses of the state variables along the execution traces, (3) a component that merges the reaching definitions and the reachable uses computed across multiple traces to produce a final set of dynamically identified data flow relations.

**Memory model.** DReaDs takes as input a program and a related test suite, and executes all the test cases in the test suite. While executing each test case, DReaDs intercepts all the data flow events, that is, memory-load and memory-store instructions, and creates a memory model that represents the references between the objects active in memory. The memory model is built and maintained at runtime, and serves to identify the state variables that are affected by load and store events.

The memory model is a directed graph, where the nodes represent the distinct object instances in memory, identified by their address, and the edges represent references between instances. An edge from a node $n_1$ to a node $n_2$ with label $l$ represents a field $l$ in $n_1$ that refers to $n_2$.

As an example, Fig. 2 reports the memory model created by DReaDs when the classes in the figure are executed with the test case *t*. Such memory model represents the state of the objects in memory after executing the test case. We can see that the memory model includes the node *A*#01 that represents the object of type `A` instantiated at line 0 (where #01 is its identity hashcode). The object is part of the state of *Z*#00 as captured by the edge *a*, reflecting the fact that the state of class `Z` includes the field `a` that refers to the object *A*#01. The memory model includes also an object of type $B$ (*B*#03) and another object of type *A* (*A*#02) and the relative relations.

DReaDs builds and maintains the memory model incrementally. It initializes the model to an empty graph, and updates the model after each execution step. The updating mechanism adds nodes and either adds or removes edges, according to the memory related operations observed during the execution.

DReaDs adds a node to the model whenever it observes a memory reference related to an object instance that is not represented yet. To this end, it relies on a runtime monitoring framework that detects each referenced object, and augments the model with a new node for each object that is not already represented in the model. In this way, DReaDs creates a memory model that includes a node for each object instance that has been accessed at least once at runtime.

DReaDs adds and removes edges to the model when observing assignments to instance fields. If the model already contains an edge that represents the field, then DReaDs removes it. If the value assigned to the field is not `null`, then DReaDs augments the model with a new edge from the field owner instance to the node identified with the assigned value.

DReaDs implements specific strategies to deal with primitive fields and arrays. Primitive fields are represented as edges from the corresponding instance to a special sink node that stands for any primitive value. Array structures are treated as special instances that include a field for each offset in the array.

**Dynamic analysis of reaching definitions and reachable uses.** DReaDs exploits the memory model at runtime to map memory-load and memory-store events to the class state variables they involve.

When a memory event occurs on an object *o*, DReaDs identifies all the objects affected by the event that include the object $o$ itself and all the objects that contain a direct or indirect reference to *o* as a field value. First, DReaDs identifies the node $n$ in the memory model corresponding to *o*. Then, starting from this node, it traverses the graph in a depth-first fashion to identify all the nodes $n_0$, ..., $n_k$ that are directly or indirectly connected to $n$, and that represent all the objects $o_0$, ..., $o_k$ that own a reference to the object *o*. Each path from a node $n_{i=0..k}$ to $n$ identifies a state variable that corresponds to an object $o_i$ that includes $o$ as part of its state. For example, in the memory model shown in Fig. 2, the path from *Z*#00 to *B*#03 identifies the class state variable `Z.a.b` that represents the inclusion of the object *B*#03 in the state of the object *Z*#00.

DReaDs distinguishes between events that represent *definitions* and events that represent *uses* of state variables. *Definition* events correspond to changes of the internal state of an object, which is triggered by the assignment of a field. *Use* events correspond to accesses to the value of state variables in program statements.

In Fig. 2, the execution of the line 5 of test $t$ leads to the creation of the node *Z*#00, and of the edge *a*. Navigating the graph, DReaDs identifies the occurrence of the direct definition of the state variable `Z.a`, and of the nested definition of the state variable `Z.a.b` that is part of the state of *Z*#00. Similarly, line 6 loads the value of *B*#03. DReaDs interprets this operation as a use of the class state variables `A.b` and `Z.a.b`.

DReaDs analyzes both the propagation of the assigned values through the code that can be executed thereafter (reaching definition analysis) and the reachability of the accessed values (reachable uses analysis). To this end, DReaDs maintains a map of active definitions in memory and computes the dynamic reaching definitions and reachable uses incrementally for each basic block of each object, according to the classical data-flow analysis equations [1]. Furthermore, DReaDs maps each use of a state variable to the last active definition of the state variable in the execution trace, thus identifying the executed def-use pairs.

**Generalization across multiple traces.** As discussed above, DReaDs maintains a high degree of precision by computing and storing data flow information for each single object instance observed in each execution trace. At the end of the execution of all test cases, DReaDs generalizes the information observed across multiple traces and objects by merging the information collected on instances of the same class. In other words, while at runtime DReaDs distinguishes objects using their identity (memory address), at the end of execution DReaDs distinguishes only between classes.

In this generalization step, DReaDs computes the set of reaching definitions and reachable uses for each basic block of the control flow graph, together with the executed def-use pairs. Intuitively, this corresponds to all the elements that have been observed in at least one execution trace for an instance of the class under test.

## *B. Inference of Test Objectives*

In the second step, DynaFlow uses the data flow information computed dynamically in the first step to identify new test objectives that corresponds to data flow elements that have not been executed yet.

As in classic data flow testing techniques [2], [14], DynaFlow considers never executed def-use pairs as test objectives. As discussed above, we focus on interactions of methods through state variables and consider def-use pairs of class fields that correspond to inter- and intra-class def-use pairs according to Harrold and Rothermel [5].

DynaFlow uses both the reaching definitions and the reachable uses computed in the first step to identify such new def-use pairs. For any class state variable $v$ and any method $m$, DynaFlow first identifies both the definitions of $v$ that reach the exit of $m$ along at least one execution trace (defs@exit) and the uses of $v$ reached from the entry of $m$ along at least one execution trace (uses@entry). It then pairs the defs@exit with the uses@entry for the same class state variable to identify the def-use pairs that can be executed by calling a pair of methods in sequence in a particular class state.

More precisely, we define the set $defs@exit_{v,m}$ of a class state variable $v$ with respect to a class method $m$ as the set of the definitions of $v$ that are executed within $m$ and reach the exit of $m$ according to the information computed dynamically in the first step. Similarly, we define the set $uses@entry_{v,m}$ of a class state variable $v$ with respect to a class method $m$ as the set of the uses of $v$ that have been reached from the entry of $m$ according to the information computed dynamically in the first step. For each class state variable $v$, DynaFlow produces the set of def-use pairs to be executed that correspond to the test objectives, by computing the cartesian product of $defs@exit_{v,m'}$ and $uses@entry_{v,m''}$ for any possible pair of methods $m'$ and $m''$, and removing all def-use pairs that were already executed.

The cartesian product of defs@exit and uses@entry may contain some infeasible def-use pairs, but, differently from classic static analysis, the cartesian product is computed on dynamically produced information and this guarantees the feasibility of the definitions and uses that occur in the pairs. Thus the result suffers only from the possible infeasibility of the pairing, but not of the single elements.

### C. Test Cases Generation

In the third and last step, DynaFlow generates test cases that exercise the new def-use pairs identified in the former step. These test cases exercise execution paths that are not exercised by executing the current test suite.

There exist many techniques to automatically generate test cases. DynaFlow requires a technique that (1) can generate test cases that exercise inter-procedural paths in object oriented systems, (2) can generate test cases that cover some given def-use pairs, (3) can distinguish between paths that exercise each given def-use pair and paths that do not, for example because they kill the definition before reaching the use, or because they execute the definition and the use on different instances of the class under test. The modern technology of search-based test generation tools [15] can be adapted to satisfy these requirements.

To evaluate our technique we adapted Evosuite, a tool for automatically generating test cases based on genetic algorithms [16], to generate test cases that satisfy the test objectives identified with DynaFlow. We discuss the implementation details in Section IV.

DynaFlow iterates through the three steps described in this section until it cannot identify any additional test objective or generate any additional test case.

## IV. Evaluation

This paper starts from the observation that classic data flow analysis of object oriented code misses a lot of useful information for testing, and is grounded on the hypothesis that data flow information computed dynamically on the execution traces can overcome the limitations of static approaches and produce useful test objectives. Our empirical evaluation addresses the main research question:

*To what extent is it possible to enhance an initial test suite by exploiting dynamic data flow information?*

To answer this question, we implemented the DynaFlow technique for Java programs, and executed a set of experiments on a benchmark of classes. We compare the test suites generated with DynaFlow with the original test suites, to evaluate the ability of DynaFlow to enhance the initial suite. We also compare the DynaFlow test suites with the test suites generated with a state-of-art test generator based on static data flow analysis, to understand the improvement of dynamic over classic data flow testing. Finally, we compare the DynaFlow test suites with large test suites generated randomly to verify that the good results of DynaFlow do not depend on the size of the test suite. We compare the test suites in terms of their ability to reveal failures, and we approximate this ability as the amount of mutants killed by the test suites. The more mutants a test suite can kill, the more effective the test suite is in revealing the corresponding faults.

In the following, we present the prototype of DynaFlow, detail the design of the experiments, analyze the obtained results, and discuss the threats to the validity of our findings.

### A. Prototype Implementation

We designed a prototype implementation of the DynaFlow technique for Java programs relying on a prototype implementation of DReaDs for dynamic data flow analysis of Java programs and on a modified version of EvoSuite for test case generation.

DReaDs computes data flow information dynamically while executing JUnit test cases. It is implemented on top of the DiSL framework for dynamic analysis [17], and relies on dynamic instrumentation to capture data flow events and maintain the memory model.

A dedicated module uses the information computed with DReaDs to derive a set of new test objectives in the form of def-use pairs. Such objectives are fed to EvoSuite that we modified to generate test cases that satisfy the new objectives.

EvoSuite generates test cases using genetic algorithms, iteratively evolving an initial population of test suites by applying crossover and mutation operators to the individuals. It uses a *fitness function* to measure the distance of the individuals from the optimal solution, and retains the best individuals with high probability. We modified EvoSuite to generate test cases that execute the DynaFlow test objectives by defining an ad-hoc fitness function. EvoSuite provides a node-node fitness function to steer the test case generation first towards a node of the control flow graph and then towards another node that can be reached thereafter [15]. We encoded DynaFlow def-use pairs as pairs of nodes of the control flow graph, and adapted the node-node fitness function of EvoSuite to execute these pairs of nodes. We further adapted the fitness function both to ensure that the definition and the use of a def-use pair are executed against the same object instance and to exclude the paths that contain a kill of the definition.

Our modified fitness function distinguishes definitions and uses according to their invocation context that correspond to the chain of method invocations that leads to the definition or the use. In this way, we can identify definitions and uses of nested class state variables, captured as the modifications of the internal state of a class $C$ that occur through a chain of invocations that starts from $C$ itself and calls methods on objects that are directly or indirectly part of the state of $C$.

For instance in Fig. 2, object *B*#03 is part of the state of object *A*#01, which in turn is part of the state of object *Z*#00. Our modified fitness function considers a definition (use) that occurs on *B*#03 as a definition (use) of the class state variable *Z.a.b* when *B*#03 is accessed from a chain of method invocations that starts from *Z*#00, for instance at line 6 of the test case of Fig. 2.

## B. Experimental Setting

We evaluated DynaFlow on 30 Java classes extracted from some projects of the SF100 set of programs [18]. The benchmark reflects the expected usage scenario of DynaFlow, which targets the testing of classes with complex state. We manually selected classes among the ones that include one or more non-primitive fields and implement one or more methods that access or modify the value of such fields in a non trivial way. To avoid biases, we generated the initial test suite with EvoSuite for branch coverage that represents the most common use of EvoSuite.

Table III reports the relevant characteristics of the classes used in our experiments: the number of lines of code (*LOC*), the number of dependent classes as computed using the Dependency Finder tool[1] (*Reachable code – # classes*) and the sum of lines of code of the class under test and its dependent classes (*Reachable code – LOC*). The analysis domain of DynaFlow is the union of the class under test and its dependent classes, that is, the classes directly or indirectly called from the class under test. Thus the number of dependent classes and their LOCs indicates the size of the DynaFlow analysis domain. The last column reports the branch coverage obtained when executing the initial test suite generated with EvoSuite for branch coverage.

For each subject class, we executed DynaFlow to enhance the initial test suite, with a maximum budget of three DynaFlow iterations. We generated two additional test suites for each class: a large suite generated randomly and a suite that covers the def-use pairs computed statically. We generated the large test suite using Randoop with a limit of 1000 test cases, and the static def-use test suite using EvoSuite for static data flow testing [7].

We evaluated the effectiveness of the test suites as the amount of mutants killed when executing the suites. We use these data as a proxy measure of the amounts of failures that can be revealed by the test suites. We generated mutants for the classes under test and their dependent classes with the PiTest mutation analysis tool,[2] a tool commonly adopted in recent related work [19], [20].

TABLE III: Benchmark classes

| Class | Project | LOC | Reachable code # classes | Reachable code LOC | Branch Coverage |
|---|---|---|---|---|---|
| AttributeRegistry | freemind | 371 | 68 | 15995 | 76% |
| ColorImage | jiggler | 1273 | 43 | 11758 | 11% |
| HandballModel | jhandballmoves | 814 | 110 | 9979 | 11% |
| Robot | at-robots2-j | 417 | 109 | 7411 | 29% |
| ComplexImage | jiggler | 872 | 20 | 7153 | 50% |
| MealList | caloriecount | 388 | 30 | 4685 | 97% |
| GameState | gangup | 472 | 23 | 3813 | 95% |
| DecadalModel | corina | 295 | 15 | 3680 | 70% |
| BattleStatistics | twfbplayer | 578 | 29 | 3665 | 83% |
| Hero | dsachat | 349 | 19 | 3576 | 57% |
| FoodList | caloriecount | 146 | 24 | 3464 | 100% |
| MoveEvent | jhandballmoves | 247 | 22 | 3076 | 75% |
| Knight | feudalismgame | 393 | 16 | 2471 | 34% |
| Challenge | dsachat | 309 | 7 | 2370 | 22% |
| FieldInfo | fixsuite | 367 | 14 | 2228 | 100% |
| Formation | gfarcegestionfa | 104 | 13 | 2195 | 100% |
| ComponentInfo | fixsuite | 276 | 13 | 2119 | 100% |
| ObjectChartData Model | jopenchart | 252 | 10 | 1922 | 100% |
| ProductDetails | a4j | 518 | 19 | 1783 | 90% |
| ProductInfo | a4j | 96 | 7 | 1338 | 100% |
| HardwareBus | at-robots2-j | 144 | 13 | 1310 | 100% |
| HL7FieldImpl | openhre | 172 | 10 | 1078 | 95% |
| GroupInfo | fixsuite | 179 | 6 | 1021 | 80% |
| StackedChartData ModelConstraints | jopenchart | 164 | 5 | 910 | 69% |
| EmailFacadeImpl | bpmail | 414 | 13 | 854 | 7% |
| MemoryRegion | at-robots2-j | 48 | 7 | 839 | 100% |
| ListChannel | caloriecount | 78 | 14 | 728 | 62% |
| InventorySavePet | petsoar | 91 | 6 | 496 | 88% |
| HL7TableImpl | openhre | 60 | 5 | 396 | 100% |
| DefaultDataSet | jopenchart | 123 | 2 | 187 | 75% |

## C. Experimental Results

Table IV reports the results of our experiments: For each subject class (first column), the table indicates the number of def-use pairs (*number of test objectives*) identified by DynaFlow, the *number of mutants killed* by the generated test suites and the size (*number of test cases*) of the test suites.

The columns under *number of test objectives* report the number of def-use pairs executed i) by the initial test suite generated with EvoSuite for branch coverage (column *Initially covered*), ii) after three iterations of DynaFlow (column *Covered DynaFlow*) and iii) not yet covered after the third iteration. In our experiment the enhanced test suite always executes a higher number of def-use pairs than the initial suite. In total, the enhanced test suites execute 44% more pairs than the initial suites. Per class, we observe a median increment of 83%, with the first quartile being 27%, the third 162%, and the minimum and a maximum increments 1% and 391%, respectively. Thus, the iterative process of DynaFlow can both identify new test objectives and generate new test cases to execute them.

The test objectives that are identified but not executed by DynaFlow amount to 37% of all the identified pairs. Some of these test objectives are identified at the third iteration step for which no test generation attempt has been taken, others might not have been executed due to known limitations of EvoSuite [18], yet others are possibly infeasible.

[1]http://depfind.sourceforge.net

[2]http://pitest.org

TABLE IV: Experimental Results

| | Number of test objectives | | | Number of killed mutants | | | | Number of test cases | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Class | Initially covered | Covered DynaFlow | Not covered | EvoSuite-branch | EvoSuite-data flow | Randoop | DynaFlow | EvoSuite-branch | EvoSuite-data flow | Randoop | DynaFlow |
| AttributeRegistry | 331 | 375 | 129 | 125 | 143 | 0 | 145 | 19 | 19 | 1000 | 48 |
| ColorImage | 230 | 441 | 1439 | 165 | 107 | 34 | 270 | 19 | 33 | 1000 | 32 |
| HandballModel | 98 | 99 | 160 | 395 | 264 | 213 | 423 | 36 | 63 | 1000 | 60 |
| Robot | 91 | 188 | 129 | 43 | 32 | 60 | 63 | 20 | 60 | 1000 | 41 |
| ComplexImage | 105 | 185 | 354 | 126 | 75 | 11 | 165 | 16 | 23 | 1000 | 37 |
| MealList | 23 | 108 | 101 | 125 | 20 | 49 | 162 | 21 | 11 | 1000 | 61 |
| GameState | 2133 | 2203 | 198 | 150 | 154 | 57 | 203 | 7 | 53 | 1000 | 51 |
| DecadalModel | 198 | 556 | 381 | 66 | 79 | 34 | 69 | 23 | 38 | 1000 | 72 |
| BattleStatistics | 317 | 380 | 95 | 294 | 175 | 93 | 334 | 20 | 34 | 1000 | 62 |
| Hero | 44 | 66 | 20 | 166 | 113 | 0 | 177 | 14 | 24 | 1000 | 80 |
| FoodList | 14 | 64 | 34 | 35 | 28 | 14 | 51 | 16 | 50 | 1000 | 46 |
| MoveEvent | 74 | 194 | 90 | 86 | 15 | 0 | 102 | 20 | 48 | 1000 | 120 |
| Knight | 76 | 89 | 106 | 134 | 99 | 135 | 166 | 23 | 65 | 1000 | 201 |
| Challenge | 54 | 103 | 66 | 89 | 59 | 5 | 132 | 19 | 15 | 1000 | 58 |
| FieldInfo | 13 | 50 | 24 | 36 | 40 | 31 | 56 | 3 | 15 | 1000 | 18 |
| Formation | 10 | 49 | 8 | 17 | 23 | 11 | 18 | 5 | 37 | 1000 | 46 |
| ComponentInfo | 42 | 123 | 31 | 64 | 77 | 55 | 69 | 8 | 34 | 1000 | 46 |
| ObjectChartData Model | 34 | 38 | 87 | 105 | 0 | 55 | 141 | 11 | 39 | 1000 | 76 |
| ProductDetails | 108 | 228 | 30 | 314 | 294 | 295 | 498 | 13 | 58 | 1000 | 105 |
| ProductInfo | 113 | 281 | 15 | 246 | 61 | 215 | 289 | 4 | 8 | 1000 | 67 |
| HardwareBus | 45 | 57 | 10 | 27 | 21 | 14 | 29 | 5 | 15 | 1000 | 16 |
| HL7FieldImpl | 24 | 46 | 6 | 64 | 74 | 0 | 75 | 8 | 38 | 1000 | 28 |
| GroupInfo | 32 | 113 | 40 | 85 | 26 | 81 | 98 | 9 | 34 | 1000 | 53 |
| StackedChartData ModelConstraints | 102 | 179 | 136 | 142 | 7 | 110 | 209 | 10 | 4 | 1000 | 39 |
| EmailFacadeImpl | 34 | 56 | 3 | 41 | 44 | 0 | 61 | 12 | 12 | 1000 | 25 |
| MemoryRegion | 44 | 54 | 9 | 53 | 38 | 0 | 58 | 7 | 9 | 1000 | 48 |
| ListChannel | 23 | 44 | 0 | 9 | 27 | 8 | 51 | 1 | 10 | 1000 | 26 |
| InventorySavePet | 12 | 20 | 3 | 16 | 5 | 24 | 31 | 8 | 33 | 1000 | 21 |
| HL7TableImpl | 13 | 19 | 0 | 34 | 34 | 35 | 38 | 5 | 7 | 1000 | 13 |
| DefaultDataSet | 9 | 14 | 2 | 12 | 25 | 25 | 23 | 9 | 20 | 1000 | 23 |
| Sum | 4446 | 6422 | 3706 | 3264 | 2159 | 1664 | 4206 | 391 | 909 | 30000 | 1619 |

All the experiments have been repeated 6 times with different seeds to govern the random mechanisms of the test generation. The 90th confidence interval was 5.5% of the measured value on the average, with a maximum of 11.5%. The 95th confidence interval was 6.5% of the measured value on the average, with a maximum of 13.7%.

The second part of Table IV (columns *Number of killed mutants*) shows the number of mutants killed by the generated test suites. The DynaFlow enhanced test suite (column *DynaFlow*) always kills more mutants than the test suite generated by EvoSuite for branch coverage (column *EvoSuite-branch*) that was used to seed DynaFlow. In total, DynaFlow kills 942 (29%) more mutants than the original test suite. Per class, we observe a median increment of 27%, with the first quartile being 12%, the third 48%, and the minimum and a maximum increments 5% and 467% respectively. These results indicate that the new test cases generated by exploiting the dynamic data flow information effectively enhance the initial test suite in terms of fault detection.

Column *Evosuite-data flow* reports the mutants killed by test suites generated with EvoSuite for static data flow coverage. The DynaFlow test suites kills more mutants that the ones generated for static data flow coverage for most classes, with an average of twice as many killed mutants. Per class, we observe a median increment of number of mutants killed with DynaFlow over Evosuite for static data flow coverage of 69%, with the first quartile being 32%, and the third 152%. This result supports our claim that dynamic data flow analysis selects better test objectives than static data flow analysis. Counter-intuitively the data in the table indicate that EvoSuite-branch slightly outperforms EvoSuite-data flow, differently to what reported in previous work [7]. This may be caused by the fact that this benchmark includes complex classes that are difficult to analyze with static analysis.

The third part of the table (columns *Number of test cases*) reports the amount of test cases generated with the different approaches, and indicates that DynaFlow generates larger test suites than the other approaches, thus raising the question of the importance of the size of the test suites. To check the impact of the site size on the test effectiveness, we compared the results of DynaFlow test suites with the results of very large test suites generated randomly with Randoop. The results are reported in columns *Randoop*. We can see that Randoop is much less effective than any of the other techniques, despite the very large size of the test suites generated with Randoop (1000 test cases for each suite). In total, the DynaFlow test suites kill 2.5 times more mutants than Randoop test suites. We observe that Randoop kills zero or few mutants for some classes. This happens for classes that require a complex initialization or a complex interaction with other classes to be thoroughly exercised. This indicates that the larger amount to mutants killed with DynaFlow test suites does not depend on the size of the suite, but on the quality of the test cases.

The dynamic analysis component of DynaFlow generated the test objectives executing the test cases of each class within 10 seconds in most of the cases, and within 45 seconds in few worst cases, with performances fully acceptable in the context of automated test case generation. The current main cost factor in the experiments is the test case generation step that depends on our naïve customization of EvoSuite.

```
1  class BattleStatistics {
2   private SidesCounter swaps = new SidesCounter();
3   public int totalSwaps(final CombatantSide side) {
4    return swaps.getSideValue(side); // Mutant: return 0;
5   }}
6
7  class SidesCounter {
8   private Map perSideCounters = ...;
9   public void incrementPerSideCounters(...){...}
10  public int getSideValue(CombatantSide side) {
11   int sum = 0;
12   if (side == null) {
13    for (Counter counter : perSideCounters.values()) {
14     sum += counter.getValue();
15    }
16    return sum; // Mutant: return 0;
17   }
18   return perSideCounters.get(side).getValue();
19   // Mutant: return 0;
20  }}
```

Fig. 3: Example of mutants killed only by DynaFlow test cases

## D. Discussion

The results presented in the former subsection indicate that dynamic data flow testing can augment an initial test suite with test cases that reveal faults that would otherwise go undetected, and thus enable us to positively answer our research question. To support our hypothesis that dynamic data flow analysis can identify interactions among methods that are difficult to find otherwise, we manually inspected the mutants killed only by the DynaFlow test cases and investigated their nature.

Indeed, we found out that most of the mutants killed only by DynaFlow test cases are characterised by particular combinations of method calls that are triggered by test objectives that involve interactions through dynamically instantiated state variables. These interactions require the methods of the class under test to be executed in multiple invocation contexts and with different values of the (nested) class state variables. DynaFlow can identify such interactions dynamically, and generate test cases that kill the corresponding mutants, while the other approaches cannot identify these interactions and thus fail to generate the required test cases.

Fig. 3 shows two classes and a set of mutants that are killed only by DynaFlow test cases. The listing shows the class under test `BattleStatistics` that owns a reference to class `SidesCounter` through its `swap` field. The method `totalSwaps()` in class `BattleStatistics` invokes the method `getSideValue()` in class `SidesCounter`.

Three mutants, obtained with the PiTest operator that substitutes the return statements in method `totalSwaps()` and `getSideValue()` with `return 0`, are indicated with comments in the figure. In our experiments, only DynaFlow test cases kill these three mutants. This happens because DynaFlow targets the interactions on the subfields of variable `swap` that are nested in the state of the class under test: DynaFlow requires the two uses of the state variable `Battle-Statistics.swaps.perSideCounters` at lines 13 and 18 to be coupled with observed definitions of that state variable. For example, DynaFlow explicitly requires the use at line 13 to be executed after the definition within the method at line 9 (not showed in the figure for space reasons) that modifies the content of the map `perSideCounter`. This increases the chances of returning a `sum` different from 0 and exposing the mutant at line 16. None of the other approaches identifies test objectives that capture this combinations of method calls, and thus may not generate test cases that exercise such combinations.

## E. Threats to Validity

This subsection acknowledges the threats that may limit the validity of our experimental results, and briefly discusses the countermeasures that we adopted to mitigate such threats.

Threats to internal validity may derive from the evaluation setting and execution, and depend on the DynaFlow prototype and EvoSuite. Although we tested the DynaFlow prototype, we cannot exclude the presence of faults. We are aware that the limitations of EvoSuite could prevent the execution of feasible def-use pairs, and thus our results may be a pessimistic approximation of the effectiveness of DynaFlow. Current work on improving EvoSuite could reduce this limitation, potentially increase the effectiveness of our prototype, and improve the accuracy of the results. The randomness nature of EvoSuite could also impact on the results, to reduce such impact we repeated each experiment 6 times.

The selection of the initial test suite may also affect the results. We selected the initial suite automatically to avoid biases due to manual generation, and we used EvoSuite for branch coverage, being a common tool and a common setting for the tool. We also conducted some experiments with different initial test suites and we did not reveal major differences in the obtained results. Thus we decided to perform the main core of the experiments with initial test suites generated automatically.

Threats to construct validity involve how we measure the effectiveness of DynaFlow. We approximate the fault detection capability as the amount of killed mutants that we generated with the PiTest tool. Approximating fault detection in terms of killed mutants is common practice in current research projects and is widely accepted as a reasonable proxy measure. PiTest has been adopted in recent work [19], [20], and we modified it to prevent undesired approximations of the results. We plan to repeat our experiments with different mutation analysis tools and with real program faults.

Threats to external validity may derive from the selection of the benchmark classes. We mitigated this thread by randomly selecting classes from a well known corpus (SF100). We plan to extend the experiments to a larger set of classes.

## V. Related Work

In this paper we present a test case generation approach that benefits from a new application of data flow analysis that we call dynamic data flow testing. In this section we discuss the main automated test case generation techniques, and briefly survey the work in data flow testing and dynamic analysis.

**Automated Test Generation.** Automated test case generation approaches either randomly sample the possible method sequences [21], [22], [23], [24] or aim at specific test goals [25], [26], [27], [28], [29], [30], [31], [32], [16], [33], [34], [7].

Random based test generation techniques either guide the generation of incrementally improved test cases by feeding back the random process with newly built valid objects, like Eclat and Randoop [22], [24], or statically identify methods that can return objects to pass as inputs to other methods, like JCrasher [21]. In general, purely random approaches can experience difficulties in exploring method interactions and program paths that depend on constrained inputs.

Techniques that generate test cases from test objectives, consider objectives derived either from formal specifications or structural characteristics of the classes, and include techniques that address intra- or inter-procedural goals. Intra-procedural test objectives refer to some structural coverage of methods, such as statement and branch coverage. Symbolic Path Finder, Bogor/Kiasan and JBSE exploit symbolic execution to explore the path structure of the methods and generate test cases by solving path constraints [27], [29], [34]. EvoSuite relies on genetic algorithms with fitness functions designed to address different structural coverage criteria [16]. TestEra enumerates all non-isomorphic method inputs over some bounded domain [25]. The approach of parametrized unit testing applies concolic heuristics to generalize unit test cases over their inputs [28]. The common weakness of these techniques is the (often unrealistic) assumption that generating test cases to cover single methods can result, causally or coincidentally, in test cases that exercise inter-related methods and classes.

Inter-procedural test objectives explicitly identify interactions between methods. MSeqGen addresses pairs of methods that are called in sequence from the source code [31]. Seeker aims to method sequences that reach specific object states [35]. RecGen [32] and Palus [33] investigate the idea that methods with high textual similarity are likely to interact on common elements. The approach of Buy et al. and a recent extension of EvoSuite rely on static data flow analysis to address interactions between methods that define and use the same objects, respectively [30], [7]. These latter approaches share with DynaFlow the idea that def-use pairs of class state variables identify relevant state dependent behaviors in object-oriented programs, but compute the data flow relations statically and can thus miss many important relations. DynaFlow shares with previous work test objectives that identify interactions between methods, but considers a novel class of test objectives that are difficult to compute and address with existing technique.

Though focusing on test objectives, most above techniques still exploit randomized mechanisms to generate the test cases. For instance, Palus [33] is implemented on top of Randoop [24], biasing the distribution that governs the random sampling. Search based approaches [36], [16], [7] are randomized procedures guided by a fitness function over the test objectives. At the state of the art, the randomized mechanisms generate self-contained test cases more effectively than systematic techniques (like [30], [25], [26], [27], [29], [34]). These latter rely on direct heap manipulation to construct the input objects and may thus result in unrealistic test cases.

**Data Flow Testing.** Data flow analysis [37], [38], [39] has been applied to software testing since the mid seventies [2], [14], [40], [41]. Many authors have investigated the differences and the complementarities between data flow, branch and mutation testing, often with contrasting results [4], [3], [42], [43], [44], [45], [46]. Other authors have focused on the applicability of data flow analysis in presence of pointers and aliases [12], [47]. The potential of data flow abstractions for enhancing the testing of object oriented software has increasingly attracted the attention of researchers in recent years [48], [30], [49], [50], [6], [11], [51]. However, some studies indicate that the inter-procedural structure of object oriented programs that exploit dynamic binding exacerbates the problems of the precision of static data flow analysis [52], [8]. DynaFlow generates test cases that exploit data flow information derived from execution traces. Being derived dynamically, this information is less affected by the computational costs and high false-positive rates that affect static data flow analysis.

**Dynamic Analysis.** Several dynamic techniques trace the accesses to program variables to reveal anomalous uses of the memory [53], [54], [55], security vulnerabilities [56], [57], [58], or data dependencies [59], [60], [61]. Existing approaches to test automation monitor method sequences and parameter instances at runtime to automatically construct scaffolding for regression testing [62], [63], [64]. Memory graphs similar to the one implemented in DynaFlow have been used to support program comprehension and advanced functionality of debuggers [65], [66]. To the best of our knowledge, DynaFlow is the first technique that exploits dynamically observed data flow information in the context of test case generation.

## VI. Conclusions

In this paper, we present *dynamic data flow testing*, a novel approach to generate inter-procedural test cases for object oriented software systems.

Our research is grounded on the empirical observation that the data flow information computed with classic analysis of the source code misses a lot of information that corresponds to relevant dynamic behaviors that shall be tested. Our technique exploits data flow information computed dynamically on the execution traces to identify relevant method interactions and generates test cases to execute them. The approach considers an initial test suite and iteratively enhances it with new test cases. At each iteration the approach analyzes the execution traces of the available test cases to derive new test objectives, and generates test cases that cover them.

Our experiments indicate that (1) the test cases identified with our approach reveal failures that go undetected with the initial test suite, (2) the enhanced test suite is more effective than classic data flow testing, (3) the effectiveness of the generated test suite does not depend on the size of the test suite itself, but on the relevance of the identified test cases.

We believe that this work could represent the first step towards a new testing paradigm, which exploits dynamic data flow information observed at runtime to effectively exercise interesting states and interactions that could cause subtle failures and that would go otherwise undetected.

## References

[1] M. Pezzè and M. Young, *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 2007.

[2] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Transactions of Software Engineering*, vol. 11, no. 4, pp. 367–375, 1985.

[3] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proceedings of the 16th International Conference on Software Engineering*, ser. ICSE '94. IEEE, 1994, pp. 191–200.

[4] P. G. Frankl and S. N. Weiss, "An experimental comparison of the effectiveness of branch testing and data flow testing," *IEEE Transactions on Software Engineering*, vol. 19, pp. 774–787, 1993.

[5] M. J. Harrold and G. Rothermel, "Performing data flow testing on classes," in *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, ser. FSE '94. ACM, 1994, pp. 154–163.

[6] G. Denaro, A. Gorla, and M. Pezzè, "Contextual integration testing of classes," in *Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering*, ser. FASE '08. Springer, 2008, pp. 246–260.

[7] M. Vivanti, A. Mis, A. Gorla, and G. Fraser, "Search-based data-flow test generation," in *Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering*, ser. ISSRE '13. IEEE, 2013.

[8] G. Denaro, M. Pezzè, and M. Vivanti, "On the right objectives of data flow testing," in *Proceedings of the 7th International Conference on Software Testing, Verification and Validation*, ser. ICST 2014. IEEE, 2014, pp. 71–80.

[9] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the Symposium on the Foundations of Software Engineering*, ser. FSE '14. ACM, 2014.

[10] J. Laski, "Data flow testing in STAD," *Journal of Systems and Software*, vol. 12, no. 1, pp. 3–14, 1990.

[11] G. Denaro, A. Gorla, and M. Pezzè, "Datec: Dataflow testing of java classes," in *ICSE Companion '09: Proceedings of the International Conference on Software Engineering (Tool Demo)*. ACM, 2009, pp. 421–422.

[12] T. J. Ostrand and E. J. Weyuker, "Data flow-based test adequacy analysis for languages with pointers," in *Proceedings of the 4th Symposium on Testing, Analysis, and Verification*, ser. TAV '91. ACM, 1991, pp. 74–86.

[13] O. Lhoták and L. Hendren, "Scaling java points-to analysis using spark," in *Proceedings of the 12th International Conference on Compiler Construction*, ser. CC '03. Springer, 2003, pp. 153–169.

[14] P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," *IEEE Transactions of Software Engineering*, vol. 14, no. 10, pp. 1483–1498, 1988.

[15] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[16] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE '11, 2011, pp. 416–419.

[17] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi, "Disl: a domain-specific language for bytecode instrumentation," in *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, ser. AOSD '12. ACM, 2012, pp. 239–250.

[18] G. Fraser and A. Arcuri, "Sound empirical evidence in software testing," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE, 2012, pp. 178–188.

[19] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE '14. ACM, 2014, pp. 435–445.

[20] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE '14, 2014, pp. 72–82.

[21] C. Csallner and Y. Smaragdakis, "JCrasher: An automatic robustness tester for Java," *Software—Practice and Experience*, vol. 34, no. 11, pp. 1025–1050, Sep. 2004.

[22] C. Pacheco and M. D. Ernst, "Eclat: automatic generation and classification of test inputs," in *Proceedings of the 19th European conference on Object-Oriented Programming*, ser. ECOOP'05, 2005, pp. 504–527.

[23] I. Ciupa and A. Leitner, "Automatic testing based on design by contract," in *In Proceedings of Net.ObjectDays 2005 — 6th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World*, 2005, pp. 545–557.

[24] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07, 2007, pp. 75–84.

[25] D. Marinov and S. Khurshid, "Testera: A novel framework for automated testing of java programs," in *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ser. ASE '01. IEEE Computer Society, 2001, pp. 22–31.

[26] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: automated testing based on java predicates," in *Proceedings of the 11th ACM SIGSOFT international symposium on Software testing and analysis*, ser. ISSTA '02. ACM, 2002, pp. 123–133.

[27] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *Tools and Algorithms for Construction and Analysis of Systems*, ser. LNCS 2619. Springer, 2003.

[28] N. Tillmann and W. Schulte, "Parameterized unit tests," in *Proceedings of the 13th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE '05. ACM, 2005, pp. 253–262.

[29] X. Deng, J. Lee, and Robby, "Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems," in *International Conference on Automated Software Engineering*, 2006, pp. 157–166.

[30] U. Buy, A. Orso, and M. Pezzè, "Automated testing of classes," in *Proceedings of the 9th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '00. ACM, 2000, pp. 39–48.

[31] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Mseqgen: Object-oriented unit-test generation via mining source code," in *Proceedings of the the 7th Joint Meeting on The Foundations of Software Engineering*, ser. ESEC/FSE '09. ACM, 2009, pp. 193–202.

[32] W. Zheng, Q. Zhang, M. Lyu, and T. Xie, "Random unit-test generation with mut-aware sequence recommendation," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. ACM, 2010, pp. 293–296.

[33] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst, "Palus: A hybrid automated test generation tool for java," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 1182–1184.

[34] P. Braione, G. Denaro, and M. Pezzè, "Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. ACM, 2013, pp. 411–421.

[35] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su, "Synthesizing method sequences for high-coverage testing," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. ACM, 2011, pp. 189–206.

[36] P. Tonella, "Evolutionary testing of classes," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '04. ACM, 2004, pp. 119–128.

[37] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Communications of the ACM*, vol. 19, no. 3, pp. 137–148, 1976.

[38] J. Rodriguez, "A graph model for parallel computation," Massachusetts Institute of Technology, Tech. Rep. MIT/LCS/TR-6, 1969.

[39] J. Dennis, "First version of a data flow procedure language," in *Programming Symposium*, ser. Lecture Notes in Computer Science. Springer, 1974, vol. 19, pp. 362–376.

[40] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil, "A formal evaluation of data flow path selection criteria," *IEEE Transactions on Software Engineering*, vol. 15, 1989.

[41] E. J. Weyuker, "The cost of data flow testing: An empirical study," *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 121–128, 1990.

[42] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang, "An experimental evaluation of data flow and mutation testing," *Software-Practice & Experience*, vol. 26, pp. 165–176, 1996.

[43] A. P. Mathur and W. E. Wong, "An empirical comparison of data flow and mutation-based test adequacy criteria," *Software Testing, Verification and Reliability*, vol. 4, no. 1, pp. 9–31, 1994.

[44] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses vs mutation testing: an experimental comparison of effectiveness," *Journal of Systems and Software*, vol. 38, no. 3, pp. 235–253, 1997.

[45] M. M. Hassan and J. H. Andrews, "Comparing multi-point stride coverage and dataflow coverage," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE, 2013, pp. 172–181.

[46] G. Denaro, M. Pezzè, and M. Vivanti, "Quantifying the complexity of dataflow testing," in *Proceedings of the International Workshop on Automation of Software Test*, ser. AST '13. IEEE, 2013, pp. 132–138.

[47] A. Orso, S. Sinha, and M. J. Harrold, "Classifying data dependences in the presence of pointers for program comprehension, testing, and debugging," *ACM Transactions on Software Engineering and Methodology*, vol. 13, no. 2, pp. 199–239, Apr. 2004.

[48] A. Orso and M. Pezzè, "Integration testing of procedural object-oriented languages with polymorphism," in *Proceedings of the 16th International Conference on Testing Computer Software: Future Trends in Testing*, ser. TCS '99, 1999.

[49] V. Martena, A. Orso, and M. Pezzè, "Interclass testing of object oriented software," in *Proceedings of the 8th IEEE International Conference on Engineering of Complex Computer Systems*, ser. ICECCS '02. IEEE, 2002, pp. 135–144.

[50] A. L. Souter and L. L. Pollock, "The construction of contextual def-use associations for object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 29, no. 11, pp. 1005–1018, 2003.

[51] R. T. Alexander, J. Offutt, and A. Stefik, "Testing coupling relationships in object-oriented programs," *Journal of Software Testing, Verification, and Reliability.*, vol. 20, no. 4, pp. 291–327, 2010.

[52] D. Grove and C. Chambers, "A framework for call graph construction algorithms," *ACM Transactions on Programming Languages and Systems*, vol. 23, no. 6, pp. 685–746, 2001.

[53] J. C. Huang, "Detection of data flow anomaly through program instrumentation," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 3, pp. 226–236, 1979.

[54] J. Clause, I. Doudalis, A. Orso, and M. Prvulovic, "Effective memory protection using dynamic tainting," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ser. ASE '07. ACM, 2007, pp. 284–292.

[55] J. Seward and N. Nethercote, "Using valgrind to detect undefined value errors with bit-precision," in *Proceedings of the annual conference on USENIX Annual Technical Conference*. USENIX Association, 2005, pp. 2–2.

[56] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. IEEE, 2009, pp. 474–484.

[57] T. Wang, T. Wei, G. Gu, and W. Zou, "Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution," *ACM Transaction of Information System Security*, vol. 14, no. 2, pp. 1–28, 2011.

[58] J. Newsome, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, 2005.

[59] B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3, pp. 155–163, 1988.

[60] F. Tip, "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, pp. 121–189, 1995.

[61] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," *SIGSOFT Software Engineering Notes*, vol. 30, no. 2, pp. 1–36, 2005.

[62] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, "Automatic test factoring for java," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05. ACM, 2005, pp. 114–123.

[63] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil, "Carving differential unit test cases from system test cases," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '06. ACM, 2006, pp. 253–264.

[64] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang, "Ocat: Object capture-based automated testing," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10. ACM, 2010, pp. 159–170.

[65] T. Zimmermann and A. Zeller, "Visualizing memory graphs," in *Revised Lectures on Software Visualization*. Springer, 2002, pp. 191–204.

[66] A. Lienhard, T. Gîrba, and O. Nierstrasz, "Practical object-oriented back-in-time debugging," in *Proceedings of the 22nd European conference on Object-Oriented Programming*, ser. ECOOP '08. Springer, 2008, pp. 592–615.