On the Right Objectives of Data Flow Testing

Giovanni Denaro
University of Milano Bicocca
Milano, Italy 20126
Email: denaro@disco.unimib.it

Mauro Pezzè
University of Milano Bicocca
and University of Lugano
Lugano, Switzerland 6900
Email: mauro.pezze@usi.ch

Mattia Vivanti
University of Lugano
Lugano, Switzerland 6900
Email: mattia.vivanti@usi.ch

***Abstract*—This paper investigates the limits of current data flow testing approaches from a radically novel viewpoint, and shows that the static data flow techniques used so far in data flow testing to identify the test objectives fail to represent the universe of data flow relations entailed by a program. This paper compares the data flow relations computed with static data flow approaches with the ones observed while executing the program. To this end, the paper introduces a dynamic data flow technique that collects the data flow relations observed during testing. The experimental data discussed in the paper suggest that data flow testing based on static techniques misses many data flow test objectives, and indicate that the amount of missing objectives (false negatives) can be more limiting than the amount of infeasible data flow relations identified statically (false positives). This opens a new area of research of (dynamic) data flow testing techniques that can better encompass the test objectives of data flow testing.**

I. Introduction

The application of data flow analysis to software testing has been proposed in the mid seventies by Herman [1], and data flow testing has attracted a lot of attention in the following decades. Initially data flow testing criteria have been proposed as alternative approaches to classic control flow testing criteria, aiming at more thorough test suites [2], [3], [4], [5], [6], [7], [8], [9]. Lately there has been increasing emphasis on the applications of data flow testing to object oriented systems where the focus is on object interactions both at intra- and inter-class levels [10], [11], [12], [13], [14], [15], [16]. Unfortunately, despite the huge body of research, the experimental data about the effectiveness of data flow testing are still contradictory and inconclusive [6], [7], [8], [9], [17], [18], [19], [20].

This paper investigates the limits of the current approaches to data flow testing. We lay the research hypothesis that the domain of test objectives identified by the traditional (static) data-flow analysis techniques ([21], [22], [14], [23]) is scarcely representative of the data flow relations that are relevant for testing (object oriented) programs according to the foundational assumptions that underlie data-flow testing.

To investigate the effectiveness of classic techniques for data flow testing, we introduce a technique that we call *dynamic data flow analysis* that identifies the relevant data flow relations by monitoring a set of executions of the programs under test, and we compare the outcome of the static (classic) and the dynamic (from this paper) techniques, to show that classic static data flow techniques miss huge amounts of data flow information. Our dynamic data flow analysis detects the program instructions that define values of memory locations during the execution, identifies the class state variables that depend on those values, and traces the propagation of the values of those state variables through the program. It generalizes such information across multiple executions of a program.

As distinctive characteristics, the dynamic data flow analysis defined in this paper suffers much less than the static techniques from the difficulty of accounting for the (in)feasibility of program paths, and it can exploit the precise alias information available from the concrete execution states to relate memory data and class state variables with each other. In this way, we can be dramatically more precise than considering all statically computable aliases, which is the typical over-approximation when integrating alias information in a static data flow technique. Thus, the dynamic analysis defined in this paper represents an interesting reference to quantitatively evaluate the impact of such problems in the static techniques. As confirmed by the data reported in this paper, the scarce robustness with respect to alias relations leads static data flow techniques to miss large amounts of data flow relations.

This paper is organized as follows. Section II exemplifies some of the challenges involved with static data flow techniques. Section III introduces the dynamic data flow analysis for object oriented software that we use as a baseline to evaluate classic data flow analysis techniques. Section IV describes an empirical study that evaluates the completeness and consistency of classic data flow techniques by quantitatively comparing the data flow information identified statically and dynamically. Section V surveys the related work in the fields of data flow testing and dynamic analysis. Section VI summarizes the conclusions of this paper.

II. Difficulties of Classic Data Flow Testing

We illustrate some of the issues involved with data flow testing, through the sample Java program of Figure 1. The examples discussed in this section pinpoint the difficulties of the static data flow techniques to precisely interpret the semantics of the code, in particular when the expected results depend on the (non-)executability of some program paths, the possible dynamic bindings of some method calls, or the occurrence of aliases between program variables and objects in memory.

Static data flow techniques identify the propagation of definitions through the execution of the code both within single methods (intra-procedural analysis) and across method invocations (inter-procedural analysis).

In the Java program of Figure 1, the field `nest.i` is defined by invocations of the methods of the classes `Nest` (lines 34 and 35) and `NestA` (line 39). These definitions propagate intra-procedurally to the end of the respective methods. We denoted these definitions as D_0, D_1 and D_2, respectively. In Figure 1, we have annotated the exit of the methods with a comment that indicates the definitions propagated in each method, that is, the definitions that are possibly executed within the execution flow of that method and not yet overridden by any subsequent assignment until the exit of that method. For example, executing either method `Nest.n()` or `NestA.n()` would propagate D_1 or D_2, respectively. Differently, executing method `NestB.n()` would propagate the definition D_0 that is active at that point because of the execution of the constructor of class `Nest`.

The definitions D_0, D_1 and D_2 also propagate to the methods `m1`..`m5` through the calls to the methods `n`. In these cases, the propagation of the definitions depends on the dynamic binding of the method calls. For example, the call to the method `n` in method `m1` at line 7 can be dynamically bound to any of the methods of the classes `Nest`, `NestA` or `NestB`, and can thus result in the execution of the definitions at lines 34, 35 or 39, and all the three definitions can propagate to the exit point of method `m1`. Similarly, the calls to the method `n` in the methods `m2`..`m5` can be dynamically bound to different sets of methods in `Nest`, `NestA` or `NestB`, and can thus propagate different definitions to the exit of the methods, as we discuss in details in the examples below.

Methods `m1`..`m5` illustrate the impact of dynamic binding, infeasibility and aliases on data flow analysis, thus illustarting the necessity of pairing data flow analysis with other static analysis techniques, and in particular different types of alias analysis techniques [24]. The examples indicate that the choice of different partner techniques can either result in aggressive over approximations that address well some of the problems in some cases, but determine too many false positives in other cases, or can address many specific cases at the cost of increasing the overall computational complexity, and thus question the affordability of the final technique.

Method `m1` may call `nest.n()` at line 7 that, depending on the runtime type of the object `nest`, can result in executing any method out of `Nest.n()`, `NestA.n()` or `NestB.n()`. Accordingly, the execution of method `m1` can propagate any definition out of D_0, D_1 and D_2 to the exit of the method. When considering only the *static type* `Nest` declared for the reference `nest`, we may erroneously conclude that only D_1 propagates until the exit of `m1`, because of the call to `Nest.n()`. Thus, to correctly identify the possible flows of data related to method `m1`, a static data technique must know the possible dynamic bindings of the call `nest.n()`. In cases like this, we can statically identify the correct bindings with a simple and efficient *flow-insensitive may-alias analysis*.

Method `m2` shows that the use of flow-insensitive information may not be sufficient in general, since it can produce over-approximated results. Executing method `m2` may lead to calling method `nest.n()` at line 11, if the condition at line 10 holds. But this condition restricts the dynamic type of `nest` to `NestA`, and thus only definition D_2 can propagate to the exit of method `m2`. In cases like this, flow-insensitive may-alias analysis over-approximates the behavior of `m2`, because the

```
1  class ClassUT {
2     private Nest nest;
3     ClassUT(Nest n){
4        nest = n;
5     }
6     void m1(int p){
7        if(p<0) nest.n();
8     }//D0, D1, D2
9     void m2(){
10       if(!(nest instanceof NestA)) return;
11       nest.n();
12    } //D2
13    void m3(int p){
14       if(!(nest instanceof NestB)) return;
15       m1(p);
16    }//D0
17    void m4(int p){
18       p += 3;
19       if(p > 1) m1(p);
20    } //none
21    void m5(){
22       Nest ref = nest.f();
23       ref.n();
24    }//D0, D1, D2
25
26    //...
27    void doSomething(){
28       int v = nest.i;
29       /*do some computation with v*/
30    }
31 }
32 class Nest{
33    protected int i;
34    Nest(){i = 0;} //D0
35    void n(){i = 1;} //D1
36    Nest f(){/*lot of code;*/ return this;}
37 }
38 class NestA extends Nest{
39    void n(){i = 2;}//D2
40 }
41 class NestB  extends Nest{
42    void n(){}//D0
43 }
```

Fig. 1. A sample program in Java

propagation of the definitions is constrained by flow-sensitive information. *Flow-sensitive alias analysis* can provide the information needed to correctly identify the dynamic propagation of the definitions, but *flow-sensitive alias analysis* is even more computationally expensive than flow-insensitive analysis.

Method `m3` exemplifies the need for further extending the flow sensitive analysis, to handle the invocation context of the method calls. Method `m3` may call `nest.n()` indirectly, as a result of calling method `m1` at line 15. However, while the direct call of `m1` can propagate all the definitions D_0, D_1 and D_2, the call of `m1` in this context propagates only $D0$. In general, program paths and aliases through methods depend on the invocation context. In method `m3`, the condition at line 14 restricts the type of `nest` at line 15 to `NestB`, and thus only the definition D_0 can propagate to the exit of method `m3`. Thus, the already computational expensive *flow-sensitive* analysis must be made *invocation context-sensitive*, at the price of further increased complexity. The interested readers can find an exhaustive discussion of the dimensions of precision of different types of alias analysis and of the related tradeoff in the excellent paper of Barbara Ryder [24].

Method `m4` shows that the propagation of definitions may be affected also by the presence of infeasible inter-procedural paths. In fact, the contradictory conditions on parameter p in methods `m4` and `m1` prevent the call to `nest.n()` at line 7 when method `m1` is called from method `m4` at line 19. Some

combinations of symbolic reasoning and automatic constraint solving could address problems like this, but again with major impact on the complexity and the scalability of the approach.

Method `m5` illustrates the impact of aliases on the propagation of definitions [25]. In this case to correctly compute the dynamic propagation of definitions, data flow analysis should know that method `m5` enforces an aliasing between the local reference `ref` and the field `nest`. This information is essential to identify that the call to `ref.n()` at line 23 can propagate the definitions of `nest.i`. Depending on the code to be analyzed within method `nest.f()` called at line 22, correctly identifying such alias information can become a prohibitive task even for highly sophisticated alias analyses.

These examples show that static data flow analysis cannot handle well dynamic information. They also indicate that enhancing data flow analysis with alias analysis techniques does not solve the problem, due to the difficulty of finding the right level of precision (flow-, context- and value-sensitivity) of both types of analyses. In general, the designer of the data flow technique must confront themselves with the trade-off between precision and affordability on several design decisions related to the data flow analysis, the alias analysis, or to the combination of the two. Often, they end up with embracing a mix of (different) under and over-approximations that makes it unclear the degree of approximation of the final technique.

III. Dynamic Data Flow Analysis

This section defines the technique for program analysis that we refer to as DReaDs, *dynamic reaching definition analysis*, and that we use to evaluate the precision of static data flow techniques. DReaDs identifies data flow relations in the execution traces of object oriented programs. The distinctive characteristic of DReaDs is to exploit dynamic analysis, meaning that it monitors programs at runtime and tracks data flow information in the observed execution traces.

DReaDs focuses on data flow relations of class variables that underlie the interesting state based interactions between methods. DReaDs traces the interesting actions on the class state variables during the program executions by maintaining a model of the relevant relations between the objects in memory, and monitors the propagation of the state data accordingly. It merges the data flow relations observed across different execution traces to derive information compatible with the results of static data flow techniques and enable the comparison between the two approaches.

Below, we describe the concept of class state variable as addressed in DReaDs, formalize the dynamic analysis along each execution trace, and explain how DReaDs merges the data flow relations observed across multiple execution traces.

A. Class State Variables

In object oriented programming, the *class state* indicates an assignment of the attributes (the fields) declared in a class, in the context of some object that instantiates the class. A class state can be *structured* if the related class includes (at least) an attribute with non-primitive value, i.e., an attribute defined as a data structure, possibly declared with reference to the type of other classes. For example, the state of the class `ClassUT` in Figure 1 includes the object `nest` whose state is the attribute `i`, and thus the state of the class `ClassUT` includes an assignment of `nest.i`. A structured state can include several recursive nesting levels.

DReaDs aims to identify the data flow relations between the class state and the execution of the class methods. It represents the class state as a set of *class state variables*, each corresponding to an assignable (possibly nested) attribute that comprises the state of the class under test. State variables are identified by the class they belong to and the chain of field signatures that characterise the attribute.

Definition 3.1: A class state variable is a pair $\langle class_id, field_chain\rangle$, where $class_id$ is a class identifier and $field_chain$ is a chain of field signatures that navigate the data structure of the class up to an attribute declared therein.

For example, in the program of Figure 1, the field `nest.i` of the class `classUT` is identified in DReaDs as the class state variable $\langle ClassUT, nest.i\rangle$.

DReaDs aims to both identify the definitions of the class state variables, i.e., the code locations that assign values to class state variables, and analyze the propagation of the assigned values throughout the code that can be executed thereafter. According to the classical terminology of data flow analysis, a *definition* is a pair $\langle state_variable, code_location_of_assignment\rangle$, and a *reaching definition* is a definition that may propagate from the assignment to a subsequent code location.

A class state variable can refer to either a primitive or a non-primitive attribute. Non-primitive class state variables reference an entire data structure as a whole. In some context of the DReaDs analysis (§ III-B2), an action on a non-primitive class state variable can result in actions on attributes nested in the corresponding data structures. For example, the assignment `nest = n` in the constructor of the class `ClassUT` in Figure 1 determines the definition of the non-primitive class state variable $\langle ClassUT, nest\rangle$ at that code location, and the propagation of the values previously assigned to the fields of the object `n`.

B. Reaching Definitions along an Execution Trace

DReaDs monitors the execution of the program under analysis to identify the reaching definitions of the class state variables. To this end, DReaDs includes three main components: 1) A component that maintains a model of the relations between the object instances in memory at runtime to identify the class state variables involved in the execution; 2) A component that monitors the definitions of the class state variables and the related propagations along each execution trace; 3) A component that merges the reaching definitions computed across multiple traces. The pseudocode of Algorithm 1 summarizes the work flow of the technique. We use the pseudocode to frame the presentation, pointing the reader to the sections that describe the components of the technique.

DReaDs takes as input a program and a related test suite, and loops through executing all the test cases in the test suite (Algorithm 1, lines 2 and 3). For each test case, it executes the program under analysis step-by-step (lines 6 and 7) until the execution of the test case terminates (line 11).

For each execution step, DReaDs invokes the three components, here referred to as `maintainM(emory)Model` (line 8), `applyKillGen` (line 9), and `mergeR(eaching)Defs` (line 10) that are described in Sections III-B1, III-B2 and III-C, respectively. DReaDs summarizes the results in a report (line 13).

The variables `R(eaching)Defs`, `C(urrent)Defs` and `M(emory)Model` are intermediate results that DReaDs computes incrementally, and highlight the most relevant information flows through the analysis steps. `RDefs` contains the information relative to the reaching definitions. It is initially empty (line 1), is incrementally updated by the merging mechanism, and is eventually exploited to report the computed reaching definitions at the end of the analysis. `CDefs` contains the information relative to the current definitions. It is initialized to empty when DReaDs starts analyzing an execution trace (line 4), is incrementally updated by the kill-gen data flow logics, represents the (not yet merged) reaching definitions at any state of the current trace, and is exploited within the merging process. `MModel` is the memory model. It is initialized to empty at the beginning of an execution trace (line 5), is updated by the model maintenance mechanism, and represents the incrementally computed memory model that serves to implement the kill-gen semantics.

Algorithm 1 DReaDs(PUA, TS)

```
Require: Program: The program under analysis
Require: TestSuite: A test suite for the program under analysis
 1: RDefs = EMPTY
 2: while hasNextTestCase(TestSuite) do
 3:     TestCase = nextTestCase(TestSuite)
 4:     CDefs = EMPTY
 5:     MModel = EMPTY
 6:     repeat
 7:         State = executeStep(Program, TestCase)
 8:         MModel = maintainMModel(MModel, State)   ▷ § III-B1
 9:         CDefs = applyKillGen(CDefs, MModel, State) ▷ § III-B2
10:         RDefs = mergeRDefs(RDefs, CDefs, State)    ▷ § III-C
11:     until ¬atEndOfProgram(State, Program)
12: end while
13: reportReachingDefs(RDefs, Program)               ▷ § III-C
```

The next sections describe in detail the components of DReaDs.

1) Memory Model: DReaDs maintains a runtime model of the relations between the object instances in memory at runtime, to identify the class state variables involved in assignments along an execution trace. The model is a directed graph, where the nodes represent the distinct object instances in memory and the edges represent references between instances. The distinct object instances in memory are univocally identified by their identity in memory (their address). An edge from a node $n1$ to a node $n2$ with label l represents a field l in $n1$ that refers to $n2$. The primitive fields are represented in the model as edges from the corresponding instance to a special sink node that stands for any primitive value.

DReaDs builds and maintains the memory model incrementally, while monitoring the execution of the program under analysis. It initializes the model to an empty graph for each test case (Algorithm 1, line 5) and updates the model after each execution step (line 8) to correctly represent the status of the objects that exist in memory at that point of the execution. The updating mechanism consists of adding nodes and/or adding/removing 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 in the model yet. To this end, it relies on a runtime monitoring framework that detects whether the parameters of the last executed statement contain memory references that correspond to some object instances in the memory, and retrieves the identity (the memory address) of those instances. Then, it augments the model with a new node for each identity not represented in the model yet. In this way, DReaDs lazily enforces the memory model to include a node for each object instances 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 non `null`, then DReaDs augments the model with a new edge from the field owner instance to the node identified by the assigned value.

DReaDs addresses array structures as special instances that comprise a field for each offset in the array. Thus, the above representation and handling generalize to arrays as well.

2) Dynamic Reaching Definitions Analysis: The core of DReaDs is a dynamic analysis that computes the reaching definitions related to single executions of the program under analysis. In a nutshell, the analysis consists of observing the definitions due to assignments of instance fields, and tracking the propagation of those definitions until they are eventually overridden by any subsequent assignment. The dynamic analysis described in this section specializes the concept of class state variable introduced in Definition 3.1 to capture the concrete states of the object instances at runtime, and adapts the concepts of definition and reaching definitions accordingly.

Definition 3.2: An *instance state variable* is a pair $\langle instance_id, field_chain \rangle$, where $instance_id$ is the identity of an object instance and $field_chain$ is a chain of field signatures that navigate the data structure of the instance up to an attribute declared therein.

Definition 3.3: A *definition of instance state variable* is a pair $\langle instance_state_variable, assignment_statement \rangle$, indicating that an instance variable is the target of an assignment statement.

Definition 3.4: The *dynamic reaching definitions* are a map $\{i_1, i_2, ..., i_n\} \rightarrow \{D_1, D_2, ..., D_n\}$, where each i_* is an executed statement and each D_* is a set of definitions of instance variable. The dynamic reaching definitions indicate that the values set by some definitions propagate unchanged up to an statement, that is, the definitions in D_k propagate values up to the statement i_k ($0 \leq k \leq n$).

DReaDs computes the dynamic reaching definitions progressively for each statement executed along a program execution. It relies on the classical data flow analysis that links the reaching definitions at a prior statement to the reaching definitions at the immediately next statement [21]. Formally, the equation states that $NEXT = (PRIOR - KILL) \cup GEN$, where $NEXT$ and $PRIOR$ denote the set of prior and

next reaching definitions, respectively, and GEN and $KILL$ denote the sets of definitions that start and stop propagating because of the last executed statement. DReaDs assumes an empty set of reaching definitions before executing the program (Algorithm 1, line 4), and updates this set according to the above equation after executing each statement (line 9).

Only the statements that assign instance fields can (re-)define instance state variables. Specifically, an assignment of instance field defines the state variables that both relate to the assigned instance by their $instance_id$, and depend on the assigned field by their associated $field_chain$. Any other statement propagates the exact set of dynamic reaching definitions computed at the previous statement. The next paragraphs explain how DReaDs computes the sets GEN and $KILL$ when the program executes the assignment of an instance field.

The set $KILL$ includes the definitions that, propagated from the previous to this statement, will not propagate further because the related state variable is re-defined. To compute $KILL$, DReaDs just matches the instance and the field referred in the assignment statement against those of each currently propagating definition, and selects the definitions that match.

To compute the set GEN, DReaDs exploits the memory model to identify which state variables have been defined by the assignment statement in the current memory state. (Let us recall that the memory model has been updated just after the last statement and before this step of the analysis § III-B1.) An assignment of an instance field f defines the state variables that either identify f as part of the state of an instance in memory, or represent the sub-fields of f that now refer to the values of a data structure as a result of the assignment of f to that data structure.

To determine the former state variables, DReaDs uses the memory model to retrieve the instance that owns the assigned field, and performs a backward depth-first traversal of the memory model starting from the field owner instance. By construction, each node visited through the traversal and the reversed path of edges traversed up to that node represent the $instance_id$ and the $field_chain$ of a defined state variable. Thus, DReaDs augments GEN with a definition for each of those instance state variables.

When the assigned value refers to a data structure, DReaDs determines the definitions of the state variables that relate values in this data structure. First, it selects the propagating definitions whose $instance_id$ is equal to the assigned value, that is, the pre-existing definitions of the values in the data structure. Then, for each state variable v that encloses the assigned field (that is, the variables determined at the previous step) and for each definition d selected above, DReaDs builds a state variable v_{sub} to represent the subfield sub of v that now refers to the value defined by d. Specifically, it builds v_{sub} such that $v_{sub}.instance_id$ is equal to $v.instance_id$, and $v_{sub}.field_chain$ results from concatenating $v.field_chain$ to $d.state_variable.field_chain$. DReaDs augments GEN with a further definition for each instance state variable v_{sub} built as above.

C. Generalized Analysis across Multiple Traces

DReaDs interprets the information on the reaching definitions of the instance state variables produced with the dynamic analysis described in the previous section, to establish which reaching definitions of the *class* state variables have been observed across a set of executions. The reaching definitions of the class state variables generalize Definition 3.4 as follows:

Definition 3.5: The *reaching definitions* are a map $\{i_1, i_2, ..., i_n\} \rightarrow \{D_1, D_2, ..., D_n\}$, where each i_* is an statement and each D_* is a set of definitions of class state variable. The reaching definitions indicate that the values set by the definitions in D_k propagate unchanged up to the statement i_k ($0 \leq k \leq n$), *in the context of some class instance and some execution.*

After each step of dynamic analysis, DReaDs matches the concrete facts observed for the instance state variables to facts that hold for the class state variables that specify those instances in the code (Algorithm 1, line 10). In particular, DReaDs 1) matches the dynamic data about the reaching definitions of the instance state variables at the current statement to observations of reaching definitions of the corresponding class state variables at the same statement, and 2) merges the resulting definitions to the ones that were computed at previous traversals of the statement.

Matching a (reaching) definition of an instance state variable to a corresponding (reaching) definition of a class state variable simply consists of rewriting the data for the former definition, and substituting the $instance_id$ of the instance with the $class_id$ of the dynamic type of the instance. Merging the definitions that reach an statement multiple times amounts to maintaining only one representative for the definitions with exactly the same data. DReaDs indexes the observed definitions of the class state variables in a global cache, and uses bit vectors over the cached indices to efficiently represent the sets of reaching definitions associated with the statements.

As last step, DReaDs summarizes and reports the results of the analysis (Algorithm 1, line 13). In this paper, we focus on the reaching definitions related to the state variables of a class that occur when invoking the methods of that class, since this result is compatible with the one computed by the static techniques for data flow testing, which we aim to compare with. Thus, we configured DReaDs to report, for each statement, only the reaching definitions of the class state variables of the class that encloses that statement in the code. In general, the reporting mechanism depends on the intended use of the reports, and DReaDs can provide other types of reports to address different goals and levels of data flow testing, e.g., inter-class (other than intra-class) data flow interactions and integration (rather than class) testing.

IV. Evaluating Data Flow Analysis

The main goal of this paper is to study the impact of *static* data flow techniques on data flow testing. We already observed in Section II that the design of static data flow techniques is generally jeopardized by a trade-off between precision and affordability, and we hypothesized that those decisions likely result in unacceptable degrees of approximation of the test objectives computed with these techniques. Following our

hypothesis, we question whether the current approaches to data flow testing identify the right testing objectives without missing important ones. In Section III we introduced a dynamic data flow analysis technique, DReaDs, that monitors the reaching definitions of class state variables that occur through the execution of object oriented programs. Being computed from execution traces, these definitions represent feasible targets for data flow testing, and we compare them with the testing targets computed with static data flow techniques to evaluate the testing objectives computed statically.

In this section, we empirically evaluate the precision and the recall of the set of data flow relations identified with a consolidated static data flow technique, by comparing the data flow relations identified by the static technique with the (homogeneous) data flow relations that we observed at runtime with DReaDs. For the emprirical evaluation, we have implemented a prototype tool of DReaDs for Java. The prototype works on top of the DiSL[1] framework for dynamic program analysis [26].

We collect data across a benchmark of more than 1,500 Java classes. We analyze the amount of data flow relations that are statically missed though dynamically observed, and the amount of data flow relations that are statically identified but never observed. Below, we detail the research questions, the design and the results or the study, and discuss the interpretation and the threats to the validity of our findings.

A. Research Questions

Our study addresses two main research questions:

- Does a static data flow technique identify a set of data flow relations that approximates sufficiently well the universe of the data flow relations entailed by a class under test?
- To what extent is the outcome of a static technique affected by false-positive (statically identified, though infeasible) or false-negative (statically missed, though existing) data flow relations?

B. Design of the Study

The main independent variable of the study is the data flow technique applied to identify the data flow relations of the sample classes, i.e., either a static technique or DReaDs. The depended variable of each observation is the set of data flow relations identified with one of the techniques for a class. Other sources of variability include the classes and the test cases used in the experiments.

The study instantiates the static data flow approach after DaTeC, a mature and consolidated state-of-the-art technique for data flow testing of object oriented software that we developed in previous work and extensively experienced over the years [15], [23], [20]. DaTeC embodies an inter-procedural reaching definition analysis, handles Java programs, and targets reaching definitions of class state variables that comply with Definition 3.1. Thus, comparing the results of DaTeC and DReaDs is well grounded. To the best of our knowledge, we are not aware of other publicly available tools for inter-procedural data flow analysis of object oriented programs.

[1]DiSL is available on its official website: http://disl.ow2.org/

TABLE I. STATISTICS OF THE SUBJECT APPLICATIONS

Application	Eloc	#Classes	#Tests	Coverage*
Jfreechart	55k	619	26484	0.93
Collections	13k	444	20604	1.00
Lang	11k	150	3254	1.00
Jtopas	3k	63	1562	0.98
JgraphT	6k	255	1009	1.00
	88k	1531	52913	

* Median value of Eloc coverage per class, computed with Cobertura.

As data flow relations, the study measures the set of definitions of class state variables that are (may be) executed within a class method, and (may) then reach the exit of that method. Both DReaDs and DaTeC compute this information as part of the respective reaching definition analyses. Hereafter we refer to these particular sets of reaching definitions as the `defs@exit` of a class method. We measure the `defs@exit` for a class by aggregating the `defs@exit` measured for the methods of the class.

To pinpoint the difference between the results of DaTeC and DReaDs, we compare pairwise the sets of `defs@exit` computed by either technique for each method of each subject class, and filter out of each set the definitions that are computed by both techniques. Two definitions computed with DReaDs and DaTeC, respectively, are the same if they denote an assignment of the same class state variable made at the same code location. When matching between the class state variables identified by DaTeC and DReaDs, we implement the most aggressive matching between fields of polymorphic types, whose signature is identified by DaTeC and DReaDs as the type declared in the code or the type of the instance observed at runtime, respectively. The types dynamically identified by DReaDs match any compatible type statically identified by DaTeC.

When a class state variable includes an array, we consider the access to a single element of the array that we can compute dynamically as a general access to the array structure to maintain compatibility between the information computed statically and dynamically.

In the study, we execute both DaTeC and DReaDs against the classes of a set of Java applications, and collect the `defs@exit` data flow relations identified by the two techniques accordingly. Table I shows the statistics related to the subject applications. The considered applications (column *Application*) range between 3,000 and 55,000 executable lines of code (column *Eloc*), and result to a sample of 1,531 classes (column *#Classes*) that we regard as a statistically significant number of observations.

We analyzed the classes with DReaDs by executing them with test suites that include both the bundled test cases available for the classes and test cases synthesized automatically with Randoop [27] and EvoSuite [28]). [2] Table I reports the cumulative number of bundled and generated test cases that we ran for each subject application (column *Tests*), and the median value of the coverage rates (column *Coverage*) that indicates that the test suites cover a large portion of the classes.

[2]We configured Randoop and EvoSuite with time limits of 120 seconds per application and 180 seconds per class, respectively. In EvoSuite we used the branch fitness function. In Randoop we set a maximum length of 300 lines of code per generated test case. These configurations correspond to test suites that produce high statement and branch coverage figures across all our subjects. To account for the variability associated with the random-nature of the techniques, we ran Randoop and EvoSuite several times.

C. Results

We computed the static data flow relations of the classes in our sample with DaTeC, and the dynamic data flow relations of the classes executed with the test suites with DReaDs. We executed the experiments on a OSX 10.7 MacBook Pro with 2.2 GHz Intel Core i7 and 8GB of RAM memory. We did not plan precise measures of the execution time, since they are not relevant to our experiment.

Table II shows the total number of `defs@exit` computed by DReaDs and DaTeC, respectively (column *Total*), and their distribution across the classes considered in our study (columns Q_1 = lower quartile, *Median* and Q_3 = upper quartile). We observe that DReaDs reveals the existence of about an order of magnitude more `defs@exit` data flow relations than the ones identified with DaTeC, and on average up to 6 times more `defs@exit` than DaTeC per class, almost consistently through the subject applications.

Table II also report the amount of `defs@exit` data flow relations that were statically identified but never observed (column *never observed*) and the amount of `defs@exit` data flow relations that were revealed by DReaDs but missed by DaTeC (column *statically missed*). The data indicate that the imprecision in the outcome from the static technique is dominated by the statically missed relations (false negatives) over the never observed relations (potential false positives): The data reported in the table indicate that about 96% of all dynamically observed relations are missed by the static technique, while about the 23% of the relations indicated by the static analyzer are never exercised by the test cases.

The last two columns of the table further refine the datum on the statically missed relations with the information that two thirds of the statically missed relations include polymorphic references (column *impacted by polymorphism*), while only a limited amount of missed relations (12%) can be tracked to the handling of arrays (column *impacted by arrays*).

To characterize in further detail our estimations of the relative impact of the false positives and false negatives on the (data flow) testing of the classes, we inspected the data for each class. Figure 2 plots the distribution of the amount of data flow relations that are missed and observed either statically or dynamically across the classes in our sample. Figure 2-a indicates the amounts of data flow relations identified dynamically but not statically (*statically missed*), the ones identified statically but not dynamically (*never observed*), all the ones *observed dynamically* and all the ones *identified statically*; Figure 2-b plots the proportions of the statically missed relations over all the observed ones, and of the never observed relations over all the statically identified ones. We observe that the static technique misses very large amounts (indeed between the 56% and the 99%) of the relations that we observed dynamically with DReaDs for the large majority of the classes (Figure 2-b), while only a small portion of the statically identified relations remain unseen after executing the test cases. The distributions of the figures at the numerators of those proportions (Figure 2-a) indicate that there are at most 3 never observed relations per-class across the three fourths of the sample, and that DaTeC misses up to 60 `defs@exit` per-class in the three fourths of the classes with less missed relations.

Using the Student's t-test, we found statistically significant support for the (alternative) hypotheses that the number of statically missed relations exceeds by a factor of 2.4 the number of occurring relations that are also statically identified ($p_i = 0.0293$), and exceeds by a factor of 4.2 the number of statically identified relations that are never observed ($p_i = 0.0468$) across the classes in our sample.

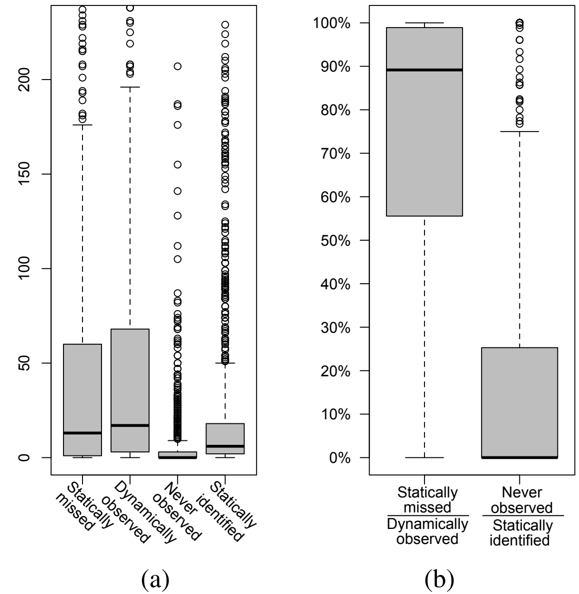

Fig. 2. Distributions of the statically missed over the dynamically observed `defs@exit`, and of the never observed over the statically identified `defs@exit`, across the sample classes

D. Discussion

The results of the study support a clear and negative answer to the first research question about the ability of static data flow techniques to approximate the universe of the data flow relations, as stated at the beginning of this section, and confirm the hypothesis of this paper on the limits of static data flow techniques to identify test targets relative to actual data flow relations. The empirical data indicate that the set of data flow relations that can be identified statically is most likely incomplete to a large extent. In other words, if we base data flow testing on a static data flow technique, we must be aware that we miss a considerable amount of data flow relations that shall be accounted as test objectives.

With reference to the second research question about the impact of false-positive and false-negative relations, as stated at the beginning of this section, this study indicates that the false negatives (statically missed, though existing data flow relations) weight much more than the false positives (statically identified, though infeasible data flow relations) on the imprecision of the static data flow techniques.

Thus, this paper suggests that the most promising research direction to improve the success of data flow testing is probably to handle the huge amount of data flow relations that hide when using the static techniques. And that is possibly counterintuitive with respect to what happens with other structural criteria, like branch coverage, where the static coverage domain is an over-approximation of the possible test objectives, and therefore the challenge is to investigate the reachability of the not yet covered elements.

TABLE II. `defs@exit` IDENTIFIED WITH DATEC AND DREADS, WITH STATISTICS PER CLASS AND CUMULATIVE SIZE OF THE RESPECTIVE DIFFERENCE SETS

	d^{DT}: `defs@exit` with DaTeC				d^{DR}: `defs@exit` with DReaDs				Never observed:	Statically missed:	Impacted by	Impacted by
Application	Total	Q_1	Median	Q_3	Total	Q_1	Median	Q_3	$\#(\in d^{DT} \wedge \notin d^{DR})$	$\#(\in d^{DR} \wedge \notin d^{DT})$	polymorphism	arrays
Jfreechart	20,513	2	9	38	89,415	3	18	75	3,480 (17%)	85,079 (95%)	47,968 (54%)	11,900 (13%)
Collections	3,908	2	4	12	63,460	4	26	81	1,779 (46%)	62,169 (98%)	55,295 (87%)	5,580 (9%)
Lang	1,227	2	3	14	1,638	2	5	13	409 (33%)	1,122 (69%)	605 (37%)	1 (0%)
Jtopas	1,481	3	12	16	8,380	6	39	320	600 (41%)	8,026 (96%)	5,085 (61%)	2,018 (24%)
JgraphT	1,800	2	4	16	6,602	1	7	44	505 (28%)	6,080 (92%)	5,259 (80%)	35 (1%)
	28,929	2	6	18	169,495	3	17	68	6,773 (23%)	162,476 (96%)	114,212 (67%)	19,534 (12%)

Following this idea, we believe that the DReaDs technique that we introduced in this paper is an interesting candidate to complement the static approaches in the context of a data flow testing technique. We read the data collected in the study as an initial body of evidence that the set of data flow relations identifiable with DReaDs largely extends the data flow information that can be identified statically. Extending DReaDs for data flow testing is not straightforward and is a major milestone of our future work.

As a consequence of the results presented in this paper, the inconclusive results about comparing data flow and structural testing approaches refer to a set of data flow relations identified statically that do not represent the many data flow relations that occur in object oriented programs. Finding a better way to identify the possible data flow relations in object oriented programs opens the way to new results about the mutual effectiveness of data flow and structural testing criteria.

E. Threats to Validity

We are aware of threats to the internal, construct and external validity of our study. The internal validity can be threatened by a scarce control on factors that may influence the results. The construct validity requires that the operational implementation of the variables properly captures the intended theoretical concepts, and that the measurements are reliable. The external validity relates the generalizability of the findings.

The main *internal* threat in this study relies in the difficulty of measuring the consistency of our results across different implementations of static data flow techniques. In our study, we refer to the technique embodied in DaTeC that implements a mature and consolidated approach to data flow testing of object oriented programs consistent with the approaches proposed in the main recent studies on this topic. We already commented on the range of design choices that can underlie the implementation of a static data flow technique: As many other static analyzers, DaTeC relies on both conservative choices, like choices concerning the feasibility of statements, paths or matching array offsets, and approximations due to the impossibility of accounting for all alias relations, as needed in particular to precisely solve polymorphic method calls. We are aware that implementations of the data flow technique different than DaTeC can lead to identify differently approximated sets of data flow relations. The current unavailability of other tools for inter-procedural data flow analysis of object oriented programs inhibited us from extending our observations beyond DaTeC. In the future, we aim to integrate DaTeC with different types of alias analysis, to further validate our conclusions, though, based on the compelling evidence gathered so far, we hardly expect a confutation of the conclusions of this paper.

Another important *internal* threat to validity refers to how well `defs@exit` reaching definitions appropriately represent the objectives of data flow class testing. Classic data flow class testing addresses the interactions (*def-use* relations) between the methods of a class that can define and use the same class state variables, when invoked sequentially in a test case [10]. We observe that identifying the `defs@exit` reaching definitions is a pre-requisite for a static technique to identify the def-use relations, since only the definitions that reach the method exit may propagate to uses in other methods. Thus, the set of `defs@exit` reaching definitions well indicate the ability of identifying def-use interactions. Measuring directly the def-use interactions would lead to less interpretable results in our study. The reason is twofold. First, since DaTeC computes the def-use relations by pairing the `defs@exit` reaching definitions and the reachable uses of the methods, after computing either information separately, the imprecisions of the data flow analysis would be reflected with combinatorial confounding effects in the measurements of the def-use relations. Second, the def-use relations predicate on the combined execution of pairs of methods, and this increases the dependence of the dynamic data on the test suites, an effect that our study aims to minimize.

A threat to the *construct* validity refers to the dependency of the data flow relations that we observed dynamically in our study on the test cases. We have executed all the test cases bundled with the subject applications augmented with test cases generated with the most popular open source automatic test case generators. There is no indication that the bundled test cases have been built with data flow testing in mind, and neither the random nor the search-based tools used in the experiment address data flow criteria directly. Thus, we cannot exclude that our set of dynamic observations can be incomplete. However, because the study reveals a huge disproportion between the amounts of statically identified and dynamically observed data flow relations, we are confident that further (currently missed) dynamic observations would not significantly alter the current results.

Another important threat to the *construct* validity refers to the reliability of our measurements that depend on the reliability of the data computed with DaTeC and DReaDs. We extensively tested and used DaTeC over the last years. We developed DReaDs only recently, and we have tested it by manually inspecting the outcome produced on a sample of the classes considered in this experiment.

The main threat to the *external* validity concerns the limits of our subjects that include only open-source applications. Thus, we shall restrict the scope of our conclusions to open-source software. In general, we are aware that the results of a single scientific experiment cannot be directly generalized.

We are currently planning replications of this study on further applications and data flow techniques.

V. Related Work

The empirical study discussed in this paper relates to previous work on data flow testing and dynamic analysis.

Data flow analysis has been investigated since the late sixties in many different contexts, starting from program optimization [29] and computer architectures [30], [31], and proposed for application to software testing since the mid seventies [1], [2], [3], [4], [5]. More recently, the potential of data flow testing for enhancing the testing of object oriented systems has increasingly attracted the attention of researchers [10], [11], [12], [13], [14], [15], [23], [16].

The advantages of data flow testing have been investigated and questioned in several experiments. In their classic papers, Frankl and Weiss compared the ”all-uses” data flow testing criterion with the ”all-edges” criterion, and found no evidence that ”the probability that a test set exposes an error increases as the percentage of definition-use associations or edges covered by it increases” [6]. Hutchins et al. reported empirical evidence that both data flow and branch testing can be more effective than random testing, but the two criteria detect complementary faults and none clearly outperforms the other. Other researchers tackled the evaluation of the effectiveness of data flow testing from a different angle, comparing between data flow and mutation testing [17], [8], [9]. These experiments indicate that satisfying mutation testing generally requires more test cases than satisfying data flow testing, but data flow testing is (almost) as much effective and thus preferable because easier to satisfy. Recently, Hassan and Andrews provided experimental data that indicate that ”MPSC is comparable in usefulness to def-use in predicting test suite effectiveness”, where MPSC generalizes branch coverage to coverage of tuples of branches [19]. Conversely, our as well as other experiments indicate that data flow testing is indeed more effective than classic control flow criteria when dealing with inter procedural aspects [20], [32].

The potentiality of data flow testing in the presence of inter procedural aspects is confirmed by the applications of data flow testing to object oriented software, where the emphasis is on object interactions both at intra- and inter-class levels. The class control flow graphs proposed by Harrold and Rothermel [10] to capture intra-class relations of object oriented programs, have been exploited by Buy et al. [12] and Martena et al. [13] to define data flow testing approaches for intra- and inter-class testing, respectively. The contextual def-use associations introduced by Souter and Pollock [14] led to data flow testing approaches that capture the structural characteristics of object oriented designs [15], [16]. The need to account for aliases and pointers may affect the precision of these analyses [25].

In summary, the usefulness of data flow testing for object oriented software is still debated, and the experimental data on its effectiveness are yet inconclusive.

This paper contributes empirical evidence that the current approaches to data flow testing of object oriented programs (based on static techniques) are not well-grounded, in that they overlook large amounts of the test objectives that shall be pursued according to the theoretical definition of data flow testing. On this basis, this paper questions the internal validity of several previous experiments on the effectiveness of data flow testing, and can explain to some extent why the reported data have been contradictory and inconclusive so far.

The argument of this paper is grounded in the concrete evidence of data flow relations that, though not identified statically, occur when executing the programs. The paper purposely introduces a dynamic analysis that pinpoints the data flow relations that occur at runtime. The dynamic technique DReaDs introduced in this paper relates to previous work that applies dynamic analysis for detecting memory anomalies, taint checking, program slicing, and back-in-time debugging. Since the seminal work of Huang, several dynamic techniques trace the accesses to the program variables to pinpoint anomalous uses of the memory, such as, never used values, reads from undefined locations, out of bounds array accesses and memory leaks [33], [34], [35]. Dynamic taint analysis identifies the statements that access unchecked inputs, and is exploited by automatic test generators to focus security vulnerabilities in the code [36], [37], [38]. Existing approaches to program slicing leverage data dependencies observed at runtime [39], [40], [41]. Memory graphs similar to the ones used in DReaDs have been used to support program comprehension tasks and advanced functionalities of debuggers [42], [43]. To the best of our knowledge, DReaDs is the first attempt to trace data flow relations at the same level of abstraction of static techniques, thus enabling comparability between static and dynamic data.

VI. Conclusions

This paper investigates the effectiveness of static data flow analysis for data flow testing, focusing the ability of classic data flow approaches to identify the proper set of data flow relations in an object oriented program. Our underlying assumption is that the usefulness of data flow testing depends on the the ability of approximating well the set of data flow relations that can be observed in the program execution, and we show that static data flow techniques miss many data flow relations that actually occur. Thus, we shall find a way to better approximate the data flow relations entailed by programs to properly evaluate data flow testing approaches.

In this paper, we focus on object oriented programs, and in particular on the data flow relations that involve reaching definitions of class state variables, and we compare the data flow relations identified with classic static data flow techniques with the data flow relations dynamically revealed during the execution. We introduce a dynamic technique to collect the reaching definitions of the class state variables, and we compare the set of dynamically identified definitions with the corresponding set of definitions computed with a state-of-the art approach based on static data flow analysis.

Surprisingly, the experimental results discussed in the paper indicate that the amount of actual data flow relations missed with the static data flow approach (false negatives) largely overruns the amount of possibly infeasible data flow relations identified by the same approach (false positives). This results indicate that research on data flow testing shall focus more on techniques to (dynamically) identify data flow relations than on the techniques to rule out false negatives.

Acknowledgment

This work is partially supported by the Swiss National Science Foundation DyStaCCo project— SNF grant nr. 146831.

References

[1] P. M. Herman, "A data flow analysis approach to program testing," *Australian Computer Journal*, vol. 8, no. 3, pp. 92–96, 1976.

[2] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Transactions on Software Engineering*, vol. 11, pp. 367–375, 1985.

[3] 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.

[4] 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.

[5] 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.

[6] 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, no. 8, pp. 774–787, 1993.

[7] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proc. of the 16th International Conference on Software Engineering*. IEEE, 1994, pp. 191–200.

[8] 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.

[9] 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.

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

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

[12] U. Buy, A. Orso, and M. Pezzè, "Automated testing of classes," in *Proc. of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*. ACM, 2000, pp. 39–48.

[13] V. Martena, A. Orso, and M. Pezzé, "Interclass testing of object oriented software," in *Proc. of the 8th International Conference on Engineering of Complex Computer Systems*. IEEE, 2002, pp. 135–144.

[14] 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.

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

[16] 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.

[17] 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.

[18] J. Andrews, L. Briand, Y. Labiche, and A. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608–624, 2006.

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

[20] G. Denaro, M. Pezzè, and M. Vivanti, "Quantifying the complexity of dataflow testing," in *Proc. of the International Workshop on Automation of Software Test*. IEEE, 2013, pp. 132–138.

[21] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[22] M. J. Harrold and M. L. Soffa, "Interprocedual data flow testing," in *Proc. of the Third Symposium on Software testing, analysis, and verification*. ACM, 1989, pp. 158–167.

[23] G. Denaro, A. Gorla, and M. Pezzè, "Datec: Contextual data flow testing of java classes," in *31st International Conference on Software Engineering - Companion Volume, 2009*, 2009, pp. 421–422.

[24] B. G. Ryder, "Dimensions of precision in reference analysis of object-oriented programming languages," in *Proc. of the 12th international conference on Compiler construction*. Springer, 2003, pp. 126–137.

[25] T. J. Ostrand and E. J. Weyuker, "Data flow-based test adequacy analysis for languages with pointers," in *Proc. of the Symposium on Testing, Analysis, and Verification*. ACM, 1991, pp. 74–86.

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

[27] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for Java," in *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*. ACM, 2007, pp. 815–816.

[28] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proc. of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ACM, 2011, pp. 416–419.

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

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

[31] 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.

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

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

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

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

[36] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proc. of the 31st International Conference on Software Engineering*. IEEE, 2009, pp. 474–484.

[37] 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.

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

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

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

[41] 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.

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

[43] A. Lienhard, T. Gîrba, and O. Nierstrasz, "Practical object-oriented back-in-time debugging," in *Proc. of the 22nd European conference on Object-Oriented Programming*. Springer, 2008, pp. 592–615.