Search-based Data-flow Test Generation

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

Andre Mis · Alessandra Gorla
Saarland University
Saarbrücken, Germany
{amis,gorla}@cs.uni-saarland.de

Gordon Fraser
University of Sheffield
Sheffield, UK
Gordon.Fraser@sheffield.ac.uk

***Abstract*—Coverage criteria based on data-flow have long been discussed in the literature, yet to date they are still of surprising little practical relevance. This is in part because 1) manually writing a unit test for a data-flow aspect is more challenging than writing a unit test that simply covers a branch or statement, 2) there is a lack of tools to support data-flow testing, and 3) there is a lack of empirical evidence on how well data-flow testing scales in practice. To overcome these problems, we present 1) a search-based technique to automatically generate unit tests for data-flow criteria, 2) an implementation of this technique in the EVOSUITE test generation tool, and 3) a large empirical study applying this tool to the SF100 corpus of 100 open source Java projects. On average, the number of coverage objectives is three times as high as for branch coverage. However, the level of coverage achieved by EVOSUITE is comparable to other criteria, and the increase in size is only 15%, leading to higher mutation scores. These results counter the common assumption that data-flow testing does not scale, and should help to re-establish data-flow testing as a viable alternative in practice.**

***Keywords*-data-flow coverage, search based testing, unit testing**

I. INTRODUCTION

Systematic test generation is often driven by coverage criteria based on structural program entities such as statements or branches. In contrast to such structural criteria, data-flow criteria focus on the data-flow interactions within or across methods. The intuition behind these criteria is that if a value is computed in one statement and used in another, then it is necessary to exercise the path between these statements to reveal potential bad computations. Studies showed that data-flow testing is particularly suitable for object-oriented code [4], [17], [31], as object-oriented methods are usually shorter than functional procedures with complex intra-procedural logic, for which classic structural criteria are intended.

Despite these studies, data-flow criteria are rarely used in practice, and this is the case because of two main reasons: First of all, there is little support for testers to measure the data-flow coverage of their test suites[1], while there is ample choice for structural criteria[2]. Secondly, testers have to put more effort in writing test cases that satisfy data-flow criteria: it is more difficult to come up with a test case that exercises a variable definition as well as a use, rather than just having to reach one statement [8]. This emphasizes the importance of *automated* test generation tools — however, most existing systematic test generation tools target either statement or branch coverage.

A further problem preventing wide-spread adoption of data-flow criteria is a lack of understanding of how well they scale to real world applications. Intuitively, data-flow criteria result in more test objectives to cover, and consequently also more test cases, but the number of infeasible test objectives (i.e., infeasible paths from definitions to uses of the same variable) is also expected to be larger than for simpler structural criteria. However, there simply is not sufficient empirical evidence to decide whether this is a show-stopper in adoption of data-flow testing criteria, or just a minor side effect.

To address these problems, in this paper we present a data-flow test generation technique implemented as an extension of the search-based EVOSUITE [11] tool, which we applied to 100 randomly selected open source Java projects. In detail, the contributions of this paper are as follows:

- We present a search-based technique to generate unit tests for data-flow criteria. This technique uses a genetic algorithm for both, the classical approach of targeting one test objective at a time, as well as the alternative approach of targeting all test objectives at the same time.
- We present an implementation of this technique, extending the EVOSUITE test generation tool to generate test suites targeting all definition-use pairs.
- We present the results of a large empirical study on open source Java applications (the SF100 corpus of classes [12]) in order to shed light on how data-flow testing scales and compares to other criteria in practice.

The results of our experiments indicate that data-flow testing is a viable alternative and does not suffer from scalability problems as feared. Given the same fixed amount of time for test generation, data-flow testing achieves significantly higher mutation scores than test suites targeting branch coverage. The effectiveness of EVOSUITE at producing data-flow oriented test suites is comparable to that of producing structural test suites, and thus in theory there is no good reason why data-flow criteria should not be applied by default in practice.

II. BACKGROUND

Structural testing techniques use the structure of the unit under test (i.e., nodes and branches in the control flow graph) as test objectives, and they consider a test suite to be adequate

[1]To the best of our knowledge, Coverlipse [21], DaTeC [6], [7] and DUAF [33] are the only code coverage tools that support data-flow criteria for Java.

[2]http://java-source.net/open-source/code-coverage

```
1  class CoffeeMachine{
2    Inventory i;
3    int price;
4    int amtPaid;
5
6    makeCoffee(){
7      boolean canMakeCoffee=
           true;
8      if(amtPaid<price)
9        canMakeCoffee=false;
10     if(canMakeCoffee)
11       amtPaid-=price;
12   }
13
14   addCoins(int amt){
15     amtPaid+=amt;
16   }
17   addInventory(int coffee,
           int sugar){
18     canAdd=true;
19     if(coffee<0||sugar<0)
20       canAdd=false;
21     else{
22       i.setSugar(
23         i.getSugar+sugar);
24       i.setCoffee(
25         i.getCoffee+coffee);
26     }
27     return canAdd;
28   }
29
30 }
```

Fig. 1. Sample CoffeeMachine Class

only if it covers each of the feasible objectives at least once. Structural testing criteria operate on a single control flow graph in isolation, and therefore they are ideal to assure that the test suite covers all the cases handled in the logic of a procedure.

Data-flow techniques emerged from the intuition that if in one point of a procedure a value is computed in the wrong way, the effects of such a fault are visible only when that value is used at another point. Thus, data-flow techniques use variable definitions and uses as test objectives, and they consider a test suite to be adequate only if it exercises program points representing definitions and uses on the same variable. Intuitively, data-flow criteria are more complex than structural techniques, since they may also test relations across different control flow graphs.

A. Data-flow Criteria

Herman was the first to identify *def-use pairs* as the relevant elements that a test suite should cover [18]. A def-use is a pair of a *definition* (i.e., program point where a value is assigned to a variable) and a *use* (i.e., program point where the value of a variable is read) of the same variable when there is at least one path in the control flow graph from the definition to the use that does not contain any other definition of the same variable (i.e., it is a *def-clear path*). For instance, method `makeCoffee()` in Figure 1 has two def-use pairs of the boolean value `canMakeCoffee` (lines 7-10 and 9-10).

In his seminal paper, Herman defined what was later called *All Def-Use Coverage* criterion, which requires a test suite to cover each def-use pair at least once, meaning that a test suite should cover at least one path from each definition to each one of its uses. Rapps and Weyuker first, and Clarke et al. later, formally defined a set of data-flow criteria and the relation among them [5], [32]. Later Harrold and Rothermel extended such data-flow criteria to test object oriented software at the unit level [17]. Classic criteria focus on the data-flow within methods, and across methods only in presence of explicit method invocations. They do not consider the data-flow interaction through instance variables when the public methods of a class are invoked in arbitrary order. This is, however, particularly important when testing a class in isolation (i.e., for unit testing), since it is often the case that the behavior of a method depends on the instance state, and consequently it is necessary to invoke any other public method that changes the instance state before executing the method that accesses it. For instance, the behavior of `makeCoffee()` depends on the amount that has been paid, which can be changed by executing `addCoins()`. To address this problem, Harrold and Rothermel introduced the concept of intra-class def-use pairs, beside the "classic" intra- and inter-method def-use pairs:

- *Intra-method* def-use pairs: The definition and the use of a variable are within the same method, and the def-use pair is exercised during a single invocation of that method (e.g., definition and use of `canMakeCoffee` at lines 9 and 10).
- *Inter-method* def-use pairs: The definition and the use of the same variable are in different methods, and the path from the definition to the use can be found by following the method invocations in the control flow graph.
- *Intra-class* def-use pairs: The definition and the use of the same instance variable are in two different methods, and the path from the definition to the use can be exercised if the two methods are invoked one after the other (e.g., definition of `amtPaid` at line 15 by invoking `addCoins()` and use of `amtPaid` at line 8 by invoking `makeCoffee`).

In our work we implement the all def-use coverage criterion, and we identify all the def-use pairs defined by Harrold and Rothermel. In Section III we provide more details of the data-flow analysis implemented in EvoSuite.

B. Search-based Testing

Generating test cases by hand is a tedious task, and while it is conceivable that a tester takes information from a coverage tool and tries to add new tests targeting uncovered branches or statements, writing test cases to cover def-use pairs is more difficult. To support users in generating tests, researchers proposed several different automated test generation techniques, such as search-based software testing (SBST). SBST casts the problem of test generation as a search problem, and applies efficient search algorithms such as genetic algorithms (GAs) to generate test cases [26]. This has the big advantage that a given test generation tool can be adapted to other criteria easily by replacing the heuristic applied during the search. In a GA, a population of candidate individuals (i.e., test cases) is evolved using search operators intended to mimic natural evolution. Crossover between two individuals produces two offspring that contain some genetic material from both parents, while mutation of individuals introduces new genetic material. The representation and search operators depend on the test generation problem at hand (e.g., sequences of method calls for testing classes [34]). A fitness function measures how good individuals are with respect to the optimization target, and the better this fitness value is, the higher the probability of an individual for being selected for reproduction, thus gradually improving the fitness of the best individual with each generation, until either an optimal solution is found or some other stopping condition (e.g., a timeout) holds. The

fitness function executes the program under test with an input at a time, and measures how close this input is to reaching a particular structural entity (e.g. node in the control flow graph) chosen as optimization target. The established heuristic to measure the distance of a test towards a node in the control flow graph consists of the *approach level*, which measures the closest point of a given execution to the target node, and the *branch distance*, which estimates how close a particular branch was to evaluating to the desired outcome (see [26] for details):

$$\text{nodeDistance} = \text{approach level} + \nu(\text{branch distance}) \quad (1)$$

This is a minimising fitness function, i.e., the target has been covered if this function evaluates to 0. The branch distance is calculated for the branch where the control flow diverged (i.e., the point of diversion measured by the approach level). As the approach level is an integer number, the branch distance is commonly normalized in the range $[0, 1]$ using a normalization function ν, such that the approach level always dominates the branch distance.

C. EvoSuite and Whole Test Suite Generation

Optimising individual tests for individual coverage goals can be problematic as individual coverage goals are usually not independent, and there is the problem of distributing a fixed search budget among all coverage goals. To overcome these problems, *whole test suite generation* aims to optimise *sets* of test cases towards covering *all* goals described by a coverage criterion. This is the approach implemented in the EvoSuite tool, and studies have confirmed that this is beneficial, since it leads to higher branch coverage, and it is less affected by infeasible coverage goals [13].

In EvoSuite, one individual of the search is a *test suite*, which may contain a variable number of test cases. Each test case in turn is a variable length sequence of calls, i.e., invocations of constructors and methods, assignment of values to fields, generation of arrays or primitive values. When a test suite individual is mutated, then each of its N test cases is mutated with probability $1/N$, and in addition new test cases are added with decreasing probability. When a test case is mutated, then each of its n calls is deleted with probability $1/n$ and changed with probability $1/n$. Finally, new calls are inserted on random objects at random positions with decreasing probability.

The fitness function in whole test suite generation aims to produce a test suite that satisfies all coverage objectives. To optimize for branch coverage it is necessary that *all* branches in a program evaluate to true and to false. For each branch, the following distance is calculated, where $d_{min}(b, T)$ is the minimal branch distance of branch b on all executions in test suite T:

$$d(b,T) = \begin{cases} 0 & \text{if the branch has been covered,} \\ \nu(d_{min}(b,T)) & \text{if the predicate has been executed at least twice,} \\ 1 & \text{otherwise.} \end{cases}$$

This function applies to all branching instructions in the code, rather than edges in the control flow. The reason to require each branch to be covered twice is that if a branch is only covered once then optimizing it to evaluate to the other outcome will not improve the coverage. The resulting fitness function is simply the sum of the individual branch distances as well as requiring all methods M to be called (M_T is the set of methods called by T) in case there are methods without branching instructions [13]:

$$\text{branchFitness}(T) = |M| - |M_T| + \sum_{b_k \in B} d(b_k, T) \quad (2)$$

III. Data-flow Testing with EvoSuite

The data-flow testing process in EvoSuite consists of two main phases: The *data-flow analysis* phase statically identifies the testing objectives, i.e., the def-use pairs, in the class under test. Then, EvoSuite generates the test cases aiming at maximizing the *coverage* of the def-use pairs.

A. Data-flow Analysis

The data-flow criterion implemented in EvoSuite combines the criterion for object-oriented software proposed by Harrold and Rothermel [17], and the all def-use pairs criterion originally proposed by Herman [18]. Thus, the data-flow analysis statically identifies intra- and inter-method def-use pairs according to standard data-flow algorithms, and intra-class def-use pairs according to the algorithm proposed by Harrold and Rothermel.

Each class under test (CUT) is considered in isolation. For each CUT, the first step consists of creating the graphs needed for the analysis: one control flow graph per each method, and a class control flow graph (CCFG) that connects them. The CCFG is the graph used in Harrold and Rothermel's algorithm to compute the intra-class def-use pairs. It simply connects all the public methods through a frame node that allows the data-flow of instance variables across these methods.

The analysis starts by visiting the methods that do not invoke any other method of the same class. To compute the intra- and inter-method pairs, we implemented a standard forward may analysis to identify which variable definitions may reach a node in the control flow graph [28]. For each node we compute which definitions may reach it from the entry node, and we create a def-use pair for each use of the same variable within the node. Finally, we add the pair to the objectives that the test suite should cover. The data-flow analysis considers local, instance and static variables. We consider each formal parameter to be defined at the beginning of the method to capture intra-method def-use pairs, and we later substitute the variable name of the formal parameter definitions and uses with the respective actual parameter to capture inter-method pairs. We call those intra-method pairs involving parameters *parameter def-use pairs*, in order to distinguish them from other intra-method pairs.

Our current implementation does not consider reference aliasing, and consequently the analysis may miss some def-use pairs. On the other hand, a sound alias analysis would

potentially increase the overall number of def-use pairs and the number of infeasible def-use pairs significantly, and may thus have a negative impact on the scalability of the technique. We plan to investigate the impact of reference aliasing on the scalability of the analysis and on the infeasibility of the test objectives as future work.

The analysis handles method invocations within a control flow graph depending on what class the invoked method belongs to. In case the invoked method belongs to the current CUT, the data-flow analysis follows the method invocation, and gets the data-flow analysis results of the invoked method with a standard inter-procedural data-flow analysis. If the invoked method does not belong to the current CUT, we approximate the analysis. Similarly to what Martena et al. proposed to test interclass relations in object oriented software [24], we classify the invoked methods according to the potential effects that they might have on the instance. If the method is an *inspector*, i.e., if it does not change the instance state, then we consider the method invocation as a *use* of the object reference on which the method is invoked. Similarly, if the invoked method is a *modifier*, i.e., if it may mutate the instance state, we consider the method invocation as a *definition* of the object reference on which the method is invoked. For instance, at line 22 the invocation `i.setSugar()` would be considered as a definition of reference `i`, and the invocation `i.getSugar()`, at line 23, as a use of the same reference. To minimize the time required by the analysis, we computed the set of inspectors and modifiers in the Java standard library once, and we stored this information such that it can be reused.

Another approximation in our analysis regards arrays. We approximate operations on arrays by considering the array as a single whole element. Consequently, we consider any operation (i.e., definition or use) on any element of the array as an operation on the whole array.

B. Covering Def-Use Pairs

Def-use pairs can be represented as node-node fitness functions according to the categorization by Wegener et al. [35]. This means that the optimization first aims to reach the first node (the definition), and then consequently the second node (the use), and standard fitness metrics such as Equation 1 are used to represent the distance to a node.

However, the coverage of the use-node is not as straight forward as simply applying Equation 1: First, we need to ensure that there is no killing definition (i.e., another definition of the same variable) between the source definition and the target use. Second, in unit testing there can be several instances of each class at any time, and we need to ensure that the definition and the use are covered on the same instance. To achieve this, we instrument the target class such that we collect an execution trace during execution of test t:

$$trace(t) = \langle (id_1, o_1, bd_1), (id_2, o_2, bd_2), (id_3, o_3, bd_3), \ldots \rangle \tag{3}$$

Each id represents a basic block in the control flow graph of the CUT, and o is a unique ID that identifies the instance on which this basic block was executed. In addition, bd represents the branch distance to executing the alternative branch.[3] In practice, we use Java's bytecode instrumentation facilities to add tracing calls that keep track which basic block (and definition or use) was executed, and we use Java's `identityHashCode` function to uniquely identify objects. Each definition and each use corresponds to a unique basic block id. The branch distance is determined by applying standard rules [26]. The projection of an execution trace on an object O is a sequence of basic block ids consisting of all ids where the corresponding object equals O:

$$p(\langle (id_1, o_1, bd_1), (id_2, o_2, bd_2), \ldots \rangle, O) = \langle (id, o, bd) \mid o = O \rangle \tag{4}$$

We denote a sub-trace of trace t from position i to position j (inclusive) as $t_{i,j}$. Let $def(d, t)$ be the set of positions in sequence t for which the basic block includes definition d, and $use(u, t)$ be the set of positions in sequence t for which the basic block includes use u.

The distance to reaching a definition is calculated using Equation 1 on the execution trace: in this case the approach level is defined as the minimal distance between a basic block in the execution trace $t = \langle (id_1, o_1, bd_1), (id_2, o_2, bd_2), (id_3, o_3, bd_3), \ldots \rangle$, and the basic block containing the definition def in the control dependence graph; the branch distance, instead, is the value bd for the corresponding position in the execution trace.

If the definition has been reached, then the approach level and branch distance are 0 by definition, and there is at least one occurrence of (def, o_i, bd_i) for that definition in the execution trace. To calculate the distance towards reaching the use, we require a sub-trace of the execution trace for which this definition is active. We thus calculate for each occurrence (def, o_i, bd_i) of the target definition in the execution trace the sub-trace $t_{i,j}$, where i is the position of (def, o_i, bd_i) and j is the position of the next occurrence of a definition def' of the same variable on the same object (def', o_i, bd_i), or else the end of the execution trace if there is no further definition. Then $t'_{i,j} = p(t_{i,j}, o_i)$ is the projection of $t_{i,j}$ on object o_i. The use-distance for this definition is now calculated using Equation 1 on $t'_{i,j}$ as described above for definitions. The overall distance to reaching the use-node is the minimum distance for each of the definitions.

The overall fitness of a test case with respect to def-use pair (d, u) on trace t is thus (assuming all l_i and l_j are adjacent definitions):

$$duFitness(d, u, t) = \begin{cases} 1 + \text{nodeDistance}(d, t) \\ \qquad \text{if } d \text{ is not covered by } t, \\ \nu(\min(\{\text{nodeDistance}(u, t_{l_i, l_j}) \\ \qquad \mid l_i, l_j \in def(d, t)\})) \\ \qquad \text{if } d \text{ is covered by } t \end{cases}$$

[3] For simplicity we assume there is always one alternative branch. However, this can be generalized to any number of alternative branches without loss of generality.

C. Evolving Test Suites for Data-flow Coverage

When targeting individual coverage goals with a limited search budget one faces the issue of having to distribute the available resources on all coverage goals, or to select some that are more important. Furthermore, if some of the coverage goals are infeasible, then any resources invested to cover these are wasted by definition. As data-flow criteria are expected to result in more coverage goals and also more infeasible coverage goals, this potentially is an issue, which can be countered by applying whole test suite generation.

A first observation is that in order to cover all def-use pairs, a test suite should first of all reach each definition. Every definition is guaranteed to be reached if all statements or all branches of a program are covered, therefore we can simply reuse the branch coverage fitness function (Equation 2) to achieve that all definitions are reached.

A second observation is that each definition and use may be part of several def-use pairs. Consequently, covering a definition or a use a single time may not be sufficient to achieve full coverage, and optimizing towards one def-use pair may lead to loss of coverage of another def-use pair involving the same definition or use. Therefore, the fitness function needs to drive test suites towards an adequate size where each def-use pair can be covered without impeding optimization towards other def-use pairs. The exact number of times a definition needs to be executed is difficult to determine as one execution of a definition may lead to coverage of several def-use pairs. Therefore, a conservative assumption is that each definition needs to be covered as many times as there are def-use pairs it is part of. If there are redundant executions in the end, then this is not problematic as typically test suites and test cases are minimized before they are presented to the developer.

Finally, for each definition that is covered by a test suite we need to determine the minimal distance to a use of that definition as described above. To simplify the notation, we use the following shorthand to denote the minimal use-distance for a def-use pair on a given test suite T, normalized to $[0, 1]$:

$$d_{use}(def, use, T) = \begin{cases} 1 & \\ \text{if } def \text{ is not covered by } T, & \\ \nu(\text{nodeDistance}(u_i, t)) \text{ for } u_i \text{ with} & \\ \text{minimal distance after } def & \end{cases}$$

Assuming a function $exec(n, T)$ that returns the number of times a node n has been executed in a test suite T, this results in the following overall fitness function:

$$\begin{aligned} \text{defuseFitness}(T) &= \text{branchFitness}(T) \\ &+ \textstyle\sum_{def_i}(|\{(d,u) \mid d = def_i\}| - |\text{exec}(def_i, T)|) \\ &+ \textstyle\sum_{(def_i, use_i)} d_{use}(def_i, use_i, T) \end{aligned} \tag{5}$$

IV. Evaluation

To gain insights into how well data-flow testing works in practice, we have implemented the described technique as part of the EvoSuite test generation tool and performed a set of experiments on the SF100 corpus of classes [12]. This is a statistically representative sample of 100 open source Java applications, which increases the confidence that the results generalize to open source software in general. In detail, we aim to answer the following research questions:

- **RQ1**: How many def-use pairs does EvoSuite identify in open source software?
- **RQ2**: How many of the def-use pairs can EvoSuite cover?
- **RQ3**: Is the whole test suite generation approach beneficial for data-flow testing?
- **RQ4**: How does the computational complexity of the fitness function affect the search?
- **RQ5**: How do def-use test suites compare to other criteria in terms of the resulting size and length?
- **RQ6**: What is the fault detection ability of the generated def-use test suites, and how do they compare to test suites generated with other criteria?

A. Experimental Setup

We used the SF100 corpus as case study for our experiments. SF100 consists of 100 projects that were randomly sampled from all Java projects on the SourceForge open source development platform. The current version of EvoSuite reports $9,268$ testable classes. For each of these classes we applied EvoSuite to generate def-use coverage, branch coverage, and weak mutation test suites, and measured each of these coverage criteria and the mutation score of the resulting test suites. Weak mutation testing differs from the mutation score in that mutation testing requires that a mutant can be detected with an assertion, whereas weak mutation only requires state infection [14]. In addition we also ran EvoSuite targeting individual def-use pairs rather than optimizing whole test suites. Each run had a timeout of 2 minutes, and used the default parameters of EvoSuite. After test generation each test suite was minimized with respect to the target criterion. For each class/configuration, experiments were repeated 10 times with different random seeds to take into account the stochastic nature of EvoSuite [2]. In total, we had $9,268 \times 4 \times 10 = 370,720$ runs of EvoSuite.

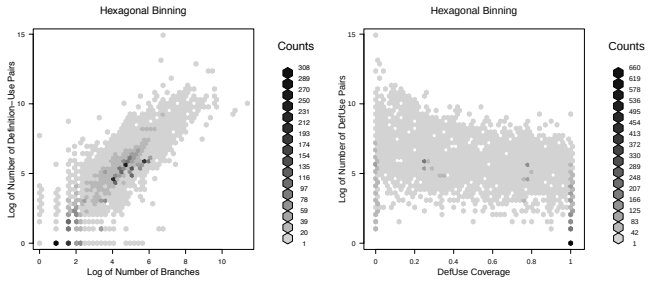
Following the guidelines in [2], all data was analyzed statistically using the Vargha-Delaney $\hat{A}_{12}$ effect size and Wilcox-Mann-Whitney U-test. Given a performance measure K (e.g., def-use coverage), $\hat{A}_{xy}$ measures the probability that running algorithm x yields higher K values than running algorithm y. If the two algorithms are equivalent, then $\hat{A}_{xy} = 0.5$. This effect size is independent of the raw values of K, and it becomes a necessity when analyzing the data of large case studies involving artifacts with different difficulty and different orders of magnitude for K. E.g., $\hat{A}_{xy} = 0.7$ entails one would obtain better results 70% of the time with x.

B. Data-flow Analysis Results

Table I lists the statistics on the def-use pairs identified in the classes of SF100. In total, there are 819,997 def-use pairs, which is almost three times as many as there are

TABLE I
STATISTICS ON THE DEF-USE PAIRS AND OTHER COVERAGE ENTITIES IN THE SF100 CORPUS OF CLASSES.

Name	Min	Avg	Median	Max	Total
Branches	0	30.27	17.00	2,478	277,757
Mutants	0	147.60	56.00	27,828	1,349,349
Def-Use Pairs	0	89.27	33.00	29,727	819,997
Intra-Method Pairs	0	34.32	13.00	1,620	315,293
Parameter Pairs	0	14.85	8.00	1,008	136,445
Inter-Method Pairs	0	1.42	0.00	466	13,007
Intra-Class Pairs	0	38.67	2.00	29,053	355,252



(a) Def-Use pairs vs. branches (correlation 0.87).

(b) Def-Use pairs vs. coverage (correlation -0.47)

Fig. 2. Comparison of achieved coverage per class

branches to cover (277,757), but significantly less than mutants produced by EVOSUITE (1,349,349). The largest share of def-use pairs are intra-class pairs, followed by intra-method pairs. Interestingly, there are only comparatively few inter-method pairs; in part this is because we only consider inter-method pairs within the same class. Figure 2(a) shows that there is a strong correlation between the number of branches and the number of def-use pairs (Pearson's correlation of 0.866 with 95 percent confidence interval of $[0.8606, 0.8708]$).

RQ1: *On average, classes have three times as many def-use pairs as branches, but significantly more mutants.*

C. Achieved Coverage

To illustrate EVOSUITE's performance at covering these def-use pairs, Table II summarizes the coverage achieved when targeting branch coverage, def-use coverage, and weak mutation testing. As expected, targeting each of the criteria achieved the highest average coverage for that particular criterion. On average, EVOSUITE achieved 54% def-use pairs coverage. At first sight this may seem low, but considering that targeting branch coverage leads to only 56% branch coverage reveals that there are many factors in SF100 that affect the achievable coverage (e.g., environmental dependencies on files and network sockets etc [12]). The branch coverage achieved by the def-use test suites is the lowest of all (49%), suggesting that the exploration of the search space did not have as much time to advance as in the case of simpler criteria. Given more time, EVOSUITE would therefore likely achieve higher coverage values. Branch coverage achieves the lowest coverage of def-use pairs and weak mutation testing. However,

TABLE II
AVERAGE COVERAGE VALUES ACHIEVED BY TARGETING DIFFERENT COVERAGE CRITERIA.

Criterion	Branch	Def-Use	Weak Mutation
Def-Use Coverage	0.49	0.54	0.46
Branch Coverage	0.56	0.48	0.45
Weak Mutation	0.50	0.51	0.50

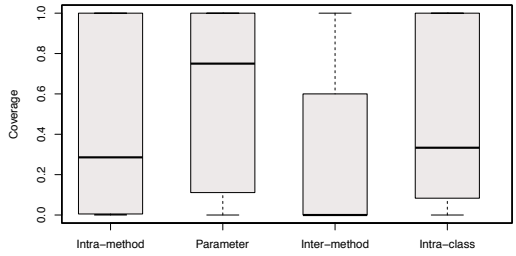


Fig. 3. Boxplot of achieved coverage per types of def-use pairs.

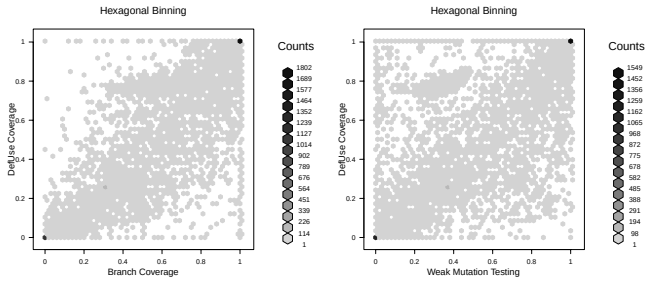
weak mutation testing covers 51% of the def-use pairs, which is significantly more than branch coverage.

RQ2: *On average, test suites produced by* EVOSUITE *cover 54% of the identified def-use pairs.*

To see how the complexity of a class influences the achieved coverage, Figure 2(b) plots the achieved def-use coverage against the number of def-use pairs (on a logarithmic scale). This illustrates that, although the average is as low as 54%, the range of coverage values is spread all across the spectrum up to 100%. There is a clear tendency that classes with more def-use pairs lead to lower coverage; this is confirmed by the correlation between number of achieved coverage and number of def-use pairs, which is -0.47 (Pearson's correlation with 95 percent confidence interval of $[-0.4828 - 0.4509]$). There is also a large cluster of trivial classes with none or a very low number of def-use pairs on which 100% coverage is (trivially) achieved.

Figure 3 illustrates the achieved coverage per type of def-use pair. Parameter-pairs have the highest average coverage, whereas inter-method pairs are the most difficult to cover. Indeed, EVOSUITE seems to have problems in covering inter-method pairs, which will merit further investigations to improve test generation. However, as the number of inter-method pairs is the smallest (cf. Table I) this does not affect the overall coverage very much. Intra-method and intra-class pairs seem to be very similar in their complexity, but more difficult than parameter pairs.

Figure 4 shows how the coverage of def-use pairs related to the other target criteria, i.e., it compares for each class the def-use coverage when targeting def-use coverage with the branch coverage when targeting branch coverage, respectively the same for weak mutation testing. The correlation between coverage of def-use pairs and branch coverage is 0.818 (Pearson's correlation with 95 percent confidence interval of $[0.8108, 0.8244]$), and the correlation between coverage of def-



(a) Def-Use vs Branch Coverage (b) Def-Use vs Weak Mutation

Fig. 4. Comparison of achieved coverage per class

use pairs and weak mutation score is 0.719 (Pearson's correlation with 95 percent confidence interval of $[0.7087, 0.7285]$). This correlation is very high, and this suggests that if a class is well testable using any of the criteria it will also be testable by the other criteria.

D. Effects of Whole Test Suite Generation

In previous work [13], [14] we have shown that whole test suite generation is beneficial when generating test suites targeting branch coverage and weak mutation testing. In both these cases the complexity of the fitness function was linear in the size of the traces of a test suite; our def-use fitness function is not linear. Therefore, we compare the results of using the whole test suite generation approach to the traditional approach targeting one def-use pair at a time. Table III summarizes the results in terms of the achieved def-use coverage. The improvement of the whole test suite generation approach is very clear with $\hat{A}_{12} = 0.74$.

This result is particularly interesting because previous studies [9], [36] reported large numbers of infeasible def-use pairs, i.e., pairs for which there exist no test cases that would cover them. Although we cannot report numbers of infeasible pairs without manual investigation, one of the main advantages of the whole test suite generation approach is that it is not adversely affected by infeasible test objectives. Since in our fitness function every def-use pair incurs some computational overhead once the definition is covered, it is not the case that the approach is completely oblivious to the number of infeasible pairs. However, the results clearly show that the problem of infeasible pairs is attenuated by the whole test suite generation approach.

RQ3: *Whole test suite generation leads to significantly higher def-use coverage compared to the traditional approach of targeting individual def-use pairs.*

E. Computational Overhead

The complexity of the def-use fitness function is higher than that of other coverage criteria, and intuitively one would expect def-use coverage to tend towards longer test cases (as will be analyzed in Section IV-F). These factors may adversely affect the search, as within a fixed search budget (e.g., time) less exploration of the search space can be performed. To quantify the effect on the search, we compare the number of executed statements per technique and class. The reason for counting statements is that the number of tests in a test suite as well as the number of statements in a test case can vary, therefore the only comparable measurement is the number of statements executed.

[4]Values in brackets are for the comparisons that are statistically significant at $\alpha = 0.05$ level; p-values are of the test for $\hat{A}_{12}$ symmetry around 0.5.

TABLE III
AVERAGE DEF-USE PAIR COVERAGE.

For each class, we counted how often the whole test suite generation approach led to worse ($\hat{A}_{12} < 0.5$), equivalent ($\hat{A}_{12} = 0.5$) and better ($\hat{A}_{12} > 0.5$) def-use coverage compared to the traditional approach of targeting individual def-use pairs.[4]

Name	Coverage	Worse	Equal	Better	$\hat{A}_{12}$	p-value
Def-Use Cov. (whole)	0.54	-	-	-	-	-
Def-Use Cov. (individual)	0.52	995 (840)	2900	5306 (1865)	0.74	0.000

TABLE IV
AVERAGE NUMBER OF EXECUTED STATEMENTS FOR BRANCH COVERAGE TEST SUITES, DEF-USE PAIR COVERAGE, AND WEAK MUTATION TESTING.

For each class, we counted how often the def-use coverage test suite led to fewer ($\hat{A}_{12} < 0.5$), equivalent ($\hat{A}_{12} = 0.5$) and more ($\hat{A}_{12} > 0.5$) executed statements.[4]

Criterion	Statements Executed	Worse	Equal	Better	$\hat{A}_{12}$	p-value
Def-Use Coverage	54728.49	-	-	-	-	-
Branch Coverage	83760.28	5716 (3277)	1086	2398 (767)	0.36	0.000
Weak Mutation	1e+05	5584 (3453)	1496	2120 (415)	0.34	0.000

Table IV compares the number of executed statements for runs with def-use coverage with runs using branch coverage and weak mutation testing. In both cases the number of executed statements is significantly lower for def-use coverage, and the effect sizes are 0.36 and 0.34. Consequently, we can conclude that the overhead of the def-use calculation affects the search, and def-use coverage would require more time to achieve the same level of exploration.

RQ4: *Def-use coverage leads to significantly fewer executed statements in a fixed time compared to branch coverage and weak mutation testing.*

F. Test Suite Size

Considering that there are typically more def-use pairs than branches, it is to be expected that there are more tests in def-use test suites than in branch coverage test suites. However, the question also is how the criterion influences the individual tests. In particular, given that intra-class pairs require more than one call, one would expect an increase in the average length of test cases (i.e. the number of statements in each test case). Table V compares the coverage criteria in terms of the number of tests generated, their total length (the sum of the lengths of the constituent tests), and the average length of the individual tests. In addition, we compare for each criterion whether the overall length of the test suites is larger than those produced by def-use coverage. Def-use coverage leads to larger test suites than branch coverage, but smaller test suites

TABLE V
AVERAGE SIZES FOR DEF-USE PAIR COVERAGE TEST SUITES, BRANCH COVERAGE TEST SUITES, AND WEAK MUTATION TESTING.
For each class, we counted how often the def-use coverage test suite led to smaller ($\hat{A}_{12} < 0.5$), equivalent ($\hat{A}_{12} = 0.5$) and larger ($\hat{A}_{12} > 0.5$) test suites.[4]

Criterion	Tests	Average Length	Total Length	Shorter	Equal	Longer	$\hat{A}_{12}$	p-value
Def-Use Coverage	6.84	3.00	25.98	-	-	-	-	-
Branch Coverage	6.69	2.86	22.54	2634 (969)	2111	4455 (2207)	0.57	7.78e-123
Weak Mutation	6.97	2.98	26.18	3757 (1760)	2563	2880 (1357)	0.48	4.6e-18

TABLE VI
AVERAGE MUTATION SCORE RESULTS FOR BRANCH COVERAGE TEST SUITES, DEF-USE PAIR COVERAGE, AND WEAK MUTATION TESTING.
For each class, we counted how often the def-use coverage test suite led to worse ($\hat{A}_{12} < 0.5$), equivalent ($\hat{A}_{12} = 0.5$) and better ($\hat{A}_{12} > 0.5$) mutation score. [4]

Name	MS	Worse	Equal	Better	$\hat{A}_{12}$	p-value
Def-Use Coverage	0.16	-	-	-	-	-
Branch Coverage	0.14	793 (180)	5742	2665 (1650)	0.57	1.29e-278
Weak Mutation	0.18	2714 (1525)	5625	861 (302)	0.44	1.11e-213

than mutation testing. On average, test cases produced for def-use coverage are longer than those produced for branch coverage and those produced for weak mutation testing.

RQ5*: Def-use coverage leads to longer tests than branch coverage and weak mutation testing, but weak mutation testing leads to more tests.*

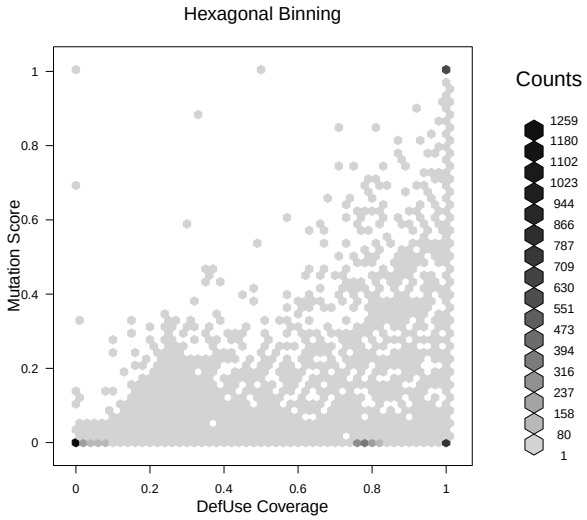


Fig. 5. Scatterplot of covered def-use pairs vs mutation score.

G. Fault Detection Ability

To determine how good the resulting test suites are at detecting faults, we use mutation analysis as a proxy measurement [1]. EVOSUITE can measure the mutation score (i.e., the ratio of detected to undetected mutants) in terms of the mutants it produces. A mutant is considered to be killed by a test case if there exists an assertion that can distinguish execution on the original version of the class and the mutant (i.e., it fails on one version and passes on the other).

Table VI summarizes how the def-use test suites compare to branch coverage and weak mutation test suites. Branch coverage achieves worse mutation scores in $2,665$ cases, and better mutation scores in 793, and thus is statistically worse with an effect size of $\hat{A}_{12} = 0.57$. Consequently, we can conclude that def-use test suites are better at detecting faults than branch coverage test suites. This is an interesting result as we have seen that def-use coverage results in significantly more coverage goals, and the overall target coverage in the fixed search budget (2 minutes) is lower than that achieved when targeting branch coverage. In addition, the branch coverage achieved by the data-flow test suites is lower than that achieved when targeting branch coverage, yet the mutation score is higher. This suggests that branch coverage is not a good proxy measurement for fault detection ability.

Compared to the weak mutation test suites the situation is the other way round: Def-use test suites are only better in 861 cases but worse in $2,714$ cases, thus def-use test suites are worse with an effect size of 0.44. This is not surprising, considering that the weak mutation test suites were optimized towards the same mutants that are used to measure the mutation score, making this a biased measurement.

RQ6*: Def-use coverage results in higher mutation scores than branch coverage, but lower mutation scores than weak mutation testing.*

One striking result is that mutation scores are generally very low. SF100 is a very large and diverse corpus of classes, and obviously one needs to be careful when just looking at average values. Indeed, Figure 5 reveals that the mutation scores are very diverse, although it seems easier to achieve high def-use coverage than it is to achieve a high mutation score. The correlation between coverage of def-use pairs and achieved mutation score is 0.435 (Pearson's correlation with 95 percent confidence interval of $[0.4185, 0.4516]$). This suggests that achieving a high coverage of def-use pairs is likely to also lead to a high mutation score.

The average mutation score is generally low for all criteria, and this is mainly due to two reasons: First, classes in SF100 represent many problems that yet need to be addressed by test generation tools, e.g., handling file accesses and network sockets. Second, classes in SF100 generally tend to be less "testable" in the sense that the public API of these classes exhibits fewer possibilities to add assertions (i.e., not enough observer methods or public fields). As a side-note, the mutation scores are also a bit lower than in our previous experiments on mutation analysis [14]. This is because in previous experiments EVOSUITE crashed on a larger number of classes which in the current version of EVOSUITE do not

lead to a crash, but still are not testable and do not lead to test suites.

H. Threats to Validity

Threats to *internal validity* may result from how the empirical study was carried out. EvoSuite and our experimental setup have been carefully tested, although testing can of course not prove the absence of defects. To accommodate for the randomness of the underlying techniques, each experiment was run 10 times and results were statistically evaluated.

Threats to *construct validity* are on how we measured the performance of a testing technique. We mainly focused on coverage, but in practice a small increase in code coverage might not be desirable if it implies a large increase of the test suite size. Furthermore, using only coverage does not take the human oracle costs into account, i.e., how difficult it will be to manually evaluate the test cases and to add assert statements.

Threats to *external validity* were minimized by using the SF100 corpus was employed as case study, which is a collection of 100 Java projects randomly selected from SourceForge [12]. This provides high confidence in the possibility to generalize our results to other open source software. We only used EvoSuite for experiments and did not compare with other tools. The reason is that we are aware of no other tool that can be *automatically* and *safely* (EvoSuite uses a sandbox to prevent potentially unsafe operations, e.g., accesses to the filesystem) be applied to SF100.

V. Related Work

A. Comparison of Data-flow with Other Criteria

Frankl and Weiss measured the ability of detecting faults in several test suites of different size and with different all-uses and statement coverage values [9], and they show that test suites with high all-uses coverage are more effective, although they are usually longer. Their study, though, involved only 9 small-sized Pascal programs with seeded errors. Moreover, their data does not show that the probability of detecting a failure increases as the percentage of def-use pairs or edges covered increases, nor give conclusive results in the case of equal-size test suites. Similarly, Hutchins et al. studied the effectiveness of all c-use/some p-use criterion and all-def-use pairs coverage criterion compared to statement coverage criterion and random testing [20]. They used 130 faulty program versions derived from seven small-sized (141 to 512 LOC) C programs. They observed that both all-def-use pairs and all-statements test suites are more effective than random testing, but they have similar detection rate, although they frequently detect different faults, and are therefore complementary.

Other studies investigated the effectiveness of data-flow testing with respect to mutation testing [10], [25], [30]. They all compared the mutation score and the fault detection ratio of test suites with high data-flow coverage and of test suites generated using mutation testing. The results show that mutation testing requires a higher number of test cases than data-flow testing (up to 200% more test cases), and is slightly better in terms of fault detection capability. All the mentioned studies were conducted on limited sets of procedural programs, and therefore they cannot be generalized to large object oriented applications.

B. Data-flow Test Generation

Hong et al. [19] applied model checking to derive test cases for data-flow criteria. They model the data-flow graph of a program as a Kripke structure, and then express the coverage criterion as a set of CTL properties, such that a counterexample to such a property represents a test case that covers a def-use pair encoded by the property. This approach can be used for different data-flow criteria by simply changing the CTL properties produced. However, this approach was only applied to a single flow graph, and does not support inter procedural analysis.

Search-based testing has been considered for data-flow testing at several points. Wegener et al. [35] defined different types of fitness functions for structural testing, and data-flow criteria are classified as "node-node" fitness functions, where the search is first guided towards reaching the first node (the definition), and then from there on towards reaching the second node (the use). Some experimental results on small example classes were presented later by Liaskos and Roper [22], [23], and a variation of this approach was also independently proposed and evaluated on small examples by Ghiduk et al. [15] and in the Testful tool [3], [27]. Although genetic algorithms are the most commonly applied search algorithm, there have also been attempts at using ant colony optimization [16] or particle swarm optimization [29], although none of these approaches has been demonstrated on non-toy problems.

Besides these search-based attempts, Buy et al. [4] combined data-flow analysis, symbolic execution and automated reasoning to generate test cases. Symbolic execution is exploited to obtain the method pre-conditions that must be satisfied in order to traverse a feasible, definition-clear path for each def-use pair, and automated deduction is used to determine the ordering of method invocations that allows satisfying the preconditions of interest. However, there is no evidence of how well this approach works in practice.

VI. Conclusions

In this paper we have presented a search-based approach to generate test suites for data-flow coverage criteria, and evaluated it on the SF100 corpus of classes. The experiments confirm that data-flow testing produces more coverage goals and, given the same search budget, more test cases than simpler criteria such as branch coverage. Mutation analysis suggests that the resulting test suites have a better fault detection ability as branch coverage test suites. On the other hand, weak mutation testing can outperform data-flow testing in terms of the mutation score, but therefore produces more test objectives and test cases. As these results were achieved on a representative sample of open source software, we can expect them to generalize to all open source software.

The most important insight of our experiments is that def-use coverage is practically applicable, and represents a viable

alternative or supplement to simpler structural criteria such as branch coverage. This counters the common belief that data-flow testing does not scale, and the use of an automated tool to produce the tests overcomes the difficulty of manually covering def-use pairs. This is an encouraging result that will hopefully foster the use of data-flow criteria in practice.

There are limitations to our approach that need to be addressed in future work. As we do not consider aliasing or contextual def-use pairs, the overall number of def-use pairs in our analysis is smaller than theoretically possible. We will extend EVOSUITE and continue experimentation with other data-flow testing variants and criteria. Finally, the use of SF100 has shown that, besides data-flow testing, there are significant open challenges in automated test generation [12], such as handling file and network dependencies.

To learn more about EVOSUITE and SF100, visit our Web site:

http://www.evosuite.org

Acknowledgments. This project has been funded by a Google Focused Research Award on "Test Amplification", the ERC grant SPECMATE and by the SNF project AVATAR (n. 200021 132666). We thank J.P. Galeotti, Andreas Zeller and Mauro Pezzè for comments on an earlier version of the paper.

REFERENCES

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

[2] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2011, pp. 1–10.

[3] L. Baresi, P. L. Lanzi, and M. Miraz, "TestFul: an evolutionary test approach for java," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2010, pp. 185–194.

[4] U. Buy, A. Orso, and M. Pezzè, "Automated testing of classes," in *ACM International Symposium on Software Testing and Analysis (ISSTA)*, 2000, pp. 39–48.

[5] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil, "A formal evaluation of data flow path selection criteria," *IEEE Transaction on Software Engineering*, vol. 15, pp. 1318–1332, November 1989.

[6] G. Denaro, A. Gorla, and M. Pezzè, "Contextual integration testing of classes," in *International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2008, pp. 246–260.

[7] ——, "DaTeC: Dataflow testing of java classes," in *International Conference on Software Engineering (ICSE Tool Demo)*. ACM, 2009, pp. 421–422.

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

[9] 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 (TSE)*, vol. 19, pp. 774–787, 1993.

[10] 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, September 1997.

[11] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software." in *ACM Symposium on the Foundations of Software Engineering (FSE)*, 2011, pp. 416–419.

[12] ——, "Sound empirical evidence in software testing," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2012, pp. 178–188.

[13] ——, "Whole test suite generation," *IEEE Transactions on Software Engineering (TSE)*, 2012.

[14] ——, "Efficient mutation testing using whole test suite generation," University of Sheffield, Technical Report CS-12-02, 2012.

[15] A. S. Ghiduk, M. J. Harrold, and M. R. Girgis, "Using genetic algorithms to aid test-data generation for data-flow coverage," in *Asia-Pacific Software Engineering Conference (APSEC)*. IEEE Computer Society, 2007, pp. 41–48.

[16] A. Ghiduk, "A new software data-flow testing approach via ant colony algorithms," *Universal Journal of Computer science and engineering Technology*, vol. 1, no. 1, pp. 64–72, 2010.

[17] M. J. Harrold and G. Rothermel, "Performing data flow testing on classes," in *ACM Symposium on the Foundations of Software Engineering (FSE)*. ACM, 1994, pp. 154–163.

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

[19] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural, "Data flow testing as model checking," in *ACM/IEEE International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2003, pp. 232–242.

[20] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *ACM/IEEE International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 1994, pp. 191–200.

[21] M. Kempka, "Coverlipse: Eclipse plugin that visualizes the code coverage of JUnit tests," http://coverlipse.sourceforge.net.

[22] K. Liaskos and M. Roper, "Hybridizing evolutionary testing with artificial immune systems and local search," in *Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conference on*, april 2008, pp. 211 –220.

[23] K. Liaskos, M. Roper, and M. Wood, "Investigating data-flow coverage of classes using evolutionary algorithms," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation (GECCO)*. ACM, 2007, pp. 1140–1140.

[24] V. Martena, A. Orso, and M. Pezzé, "Interclass testing of object oriented software," in *International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE Computer Society, 2002.

[25] 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.

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

[27] M. Miraz, "Evolutionary testing of stateful systems: a holistic approach," Ph.D. dissertation, Politecnico di Milano, 2010.

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

[29] N. Nayak and D. Mohapatra, "Automatic test data generation for data flow testing using particle swarm optimization," *Contemporary Computing*, pp. 1–12, 2010.

[30] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang, "An experimental evaluation of data flow and mutation testing," *Softw. Pract. Exper.*, vol. 26, pp. 165–176, February 1996.

[31] M. Pezzè and M. Young, *Software Test and Analysis: Process, Principles and Techniques*. John Wiley and Sons, 2008.

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

[33] R. Santelices and M. J. Harrold, "Efficiently monitoring data-flow test coverage," in *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*. ACM, 2007, pp. 343–352.

[34] P. Tonella, "Evolutionary testing of classes," in *ACM International Symposium on Software Testing and Analysis (ISSTA)*, 2004, pp. 119–128.

[35] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and Software Technology*, vol. 43, no. 14, pp. 841–854, 2001.

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