Quantifying the Complexity of Dataflow Testing

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

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

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

***Abstract*—It is common belief that dataflow testing criteria are harder to satisfy than statement and branch coverage. As motivations, several researchers indicate the difficulty of finding test suites that exercise many dataflow relations and the increased impact of infeasible program paths on the maximum coverage rates that can be indeed obtained. Yet, although some examples are given in research papers, we lack data on the validity of these hypotheses. This paper presents an experiment with a large sample of object oriented classes and provides solid empirical evidence that dataflow coverage rates are steadily lower than statement and branch coverage rates, and that the uncovered dataflow elements do not generally depend on the feasibility of single statements.**

I. Introduction

Structural testing criteria measure the effectiveness of test suites as the portion of code executed by the test cases, and indicate the parts of the code that have not been exercised yet. Structural criteria differ in the considered code elements: control flow criteria take into consideration control flow elements like statements, branches, decisions, conditions and — to a lesser extent — paths, while dataflow criteria consider relations between definitions and uses of the program variables [1].

An imperfect code coverage that derives from the presence of elements not yet executed may indicate a way to improve the test suite by generating test cases that execute those elements, but may also be the result of the presence of infeasible elements in the code [2].

Unexecuted and infeasible elements weight differently on various criteria. Criteria that involve simple code elements do not suffer much. This is the case for example of simple structural criteria like statement, branch and condition coverage that find interesting industrial applications as proxy measures of the quality of test suites. The wide industrial experience, as well as the data reported in the literature, indicates that it is indeed possible to design test suites that satisfy these coverage criteria up to reasonable thresholds [3]. In particular situations, it is even possible to manually identify and eliminate infeasible elements from the coverage domain. This is required for example by the DO-178B standard regulation that prescribes 100% modified condition decision coverage and manual identification of infeasible elements [4].

The impact of unexecuted and infeasible elements can be high for sophisticated coverage criteria, and in particular for dataflow criteria [5]. As a consequence, executing the many elements not yet exercised as well as fixing reasonable coverage thresholds can become challenging when not impossible. The scientific community agrees in considering dataflow coverage criteria more thorough than simple structural criteria, but commonly acknowledges the difficulty of executing many dataflow elements and the presence of several infeasible dataflow elements [6], [7].

Since the early definitions of dataflow testing by Herman [8] and Rapps and Weyuker [9], dataflow testing criteria have been widely studied. Weyuker studied the complexity of satisfying dataflow testing criteria theoretically, tackling the problem in terms of the number of test cases needed to satisfy those criteria [10]. Several authors experimentally compared dataflow testing with branch and mutation testing [11], [6], [7], [12], [13], [14], [15]. These experiments support an increase of both effectiveness and size between the test suites that satisfy branch, dataflow and mutation testing criteria, respectively. Malevris and Yates empirically estimated to 48% the average rate of dataflow coverage obtainable as collateral effect of pursuing branch testing [16]. All above experiments consider intra-procedural dataflow pairs of procedural programs.

Recent work suggests that such criteria may outclass other structural criteria for object oriented systems due to their ability of coping with the interplay of methods through state variables [17], [18], [19], [20]. The state dependent behaviour of object oriented systems may exacerbate the problem of executing particular code elements and emphasise the impact of infeasible elements [21]. Although these problems are often mentioned in the literature, witnessed by simple examples presented in research papers, and enforced by the lack of success of dataflow criteria in current industrial practice, there are no quantitative data supporting these hypotheses.

In this paper, we provide experimental data to support and quantify this common belief. Our experiment confirms that we can steadily achieve high statement and branch coverage with relative simplicity, but only limited dataflow coverage. We ran the experiment on independently developed open source software systems, we augmented the test suites provided by the developers with test cases generated with state of art tools, and we monitored statement, branch and dataflow coverage. The experiment indicates that the test suites consistently cover more than 85% and 78% of statements and branches, respectively, but less than 44% of definition use pairs. The insight analysis of the examined programs

978-1-4673-6161-3/13/$31.00 © 2013 IEEE

AST 2013, San Francisco, CA, USA

Listing 1: Code excerpt from Apache Commons Lang

```
class MutableFloat {
    public MutableFloat(float value) {this.value = value;}
    public void add(Number operand) {
        this.value += operand.floatValue();
    }
    public boolean isNaN() {return Float.isNaN(this.value);}
    public boolean isInfinite () {return Float. isInfinite (this.value);}
```

indicates that the low dataflow coverage springs from the intrinsic difficulty of both covering complex dataflow relations and recognizing the presence of infeasible dataflow elements, and does not depend on the feasibility of single statements in the programs. These results pinpoint the possibility to improve the thoroughness of testing by considering dataflow coverage, and suggest directions for novel research on this subject.

This paper is organised as follows. Section II illustrates the dataflow reachability and feasibility problems by means of examples excerpted from the code used in the experiment. Section III introduces the main research questions, presents the experimental setting, illustrates the dataset and discusses the threats to validity. Section IV presents the results and summarizes the main conclusions of the experiment.

II. Motivating examples

In this section, we present few examples of the usefulness of dataflow testing for object oriented programs, and the increased inefficiency that derive from infeasible (yet statically identified) testing requirements. All examples refer to dataflow testing requirements that were not exercised in the experiment reported in Section III.

The code reported in Listing 1 illustrates the usefulness of data flow testing for exercising interesting interactions between methods in object oriented code that are not captured with simple control flow criteria. The code presents some straightforward data-dependencies that foster the interaction between the methods of class `MutableFloat`, and thus originate some requirements for dataflow testing. The methods `add` and `isInfinite` can interact through the variable `value` that is defined (assigned) in method `add` and used (read) in method `isInfinite`, and witness a possible interaction between the two methods (a du pair). A typical dataflow testing criterion would require a test case that calls `add` and eventually calls `isInfinite` in such a way that the result of the latter method is computed based on the value assigned in the former method. The rationale behind this requirement is to increase the chances that the effects of a possible faulty value assigned to variable `value` in method `add` propagates to the use in method `isInfinite` and manifests as a failure. It is straightforward to design test cases that satisfy both statement and branch coverage, and yet miss the interactions between the two methods. Indeed in our experiment, neither the test cases provided by the developers nor the ones generated with state of art tools exercise this interaction.

The code reported in Listing 2 provides another example of feasible interactions yet difficult to exercise with state of

Listing 2: Code excerpt from Apache Commons Collections

```
class BinaryHeap {
    public void insert(Object element) {
        ...
        if (m_isMinHeap) percolateUpMinHeap(element, ...);
        else percolateUpMaxHeap(element, ...);
    }
    protected void percolateUpMaxHeap(final Object e, ...) {
        int hole = ...
        ... m_elements[hole] = e;
    }
    protected void percolateUpMinHeap(final Object e, ...) {
        int hole = ...
        ... m_elements[hole] = e;
    }
    public String toString () {
        for (...) {... sb.append(m_elements[i]);}
        ... return sb;
    }
```

Listing 3: Code excerpt from JFreeChart

```
class FastScatterPlot {
    private ValueAxis domainAxis;
    public void zoomDomainAxes(double factor, boolean useA) {
        if (useA) {
            ... domainAxis.resizeRange2(factor, ...);
        }
        ...
    }
    public void panDomainAxes(...) {
        double l = domainAxis.getRange().getLength();
        ...
    }

class ValueAxis {
    private Range range;
    public void resizeRange2(double percent, ...) {
        if (percent > 0.0) {
            range = new Range(...); ...
        }
        ...
    }
    public void getRange() {return range;}
```

the art testing approaches. The branch construct in method `insert` determines two alternative ways to define the state of the array-typed member variable `m_elements` that can be defined by calling either method `percolateUpMaxHeap` or method `percolateUpMinHeap`. Typical dataflow criteria require executing both branches in combination with (subsequent) calls to method `toString`, which uses the values in `m_elements`. Also in this case none of the many test cases generated for the code in our experiment satisfy either of the test requirements, thus providing further examples that thorough dataflow testing can increase the confidence in the thoroughness of the test suites.

The code reported in Listing 3 provides yet another example of method interactions useful but yet difficult to test. The code illustrates a dataflow interaction over variables encapsulated in member instances. For some particular input values, method `zoomDomainAxes` can lead to access the member instance `domainAxis` and define the value of variable `range` within this instance. A subsequent call to method `panDomainAxes` can result in the use of that value.

The code in Listing 4 is an example of a feasible definition and a feasible use that comprise an infeasible pair. It shows

Listing 4: Code excerpt from Apache Commons Collections

```
class SequencedHashMap {
    public void clear() {
        sentinel.next = sentinel;
        sentinel.prev = sentinel;
        ...
    }
     public Map.Entry getLast() {
         return (isEmpty()) ? null : sentinel.prev;
    }
    public boolean isEmpty() {return sentinel.next == sentinel;}
```

that the presence of infeasible dataflow elements goes beyond the infeasibility of single code elements, and amplifies the impact of infeasibility on dataflow criteria that we discuss in Section III. Methods `clear` and `getLast` define and use variable `sentinel.prev`, respectively, and the (static) dataflow analysis of the code identifies the interaction between the two methods through variable `sentinel.prev` as a testing requirement. However, this interaction is infeasible: For any possible input, calling method `clear` determines an empty collection, which guards executing the target use in method `getLast`, while any attempt to populate the collection by invoking other methods (not included in the listing) of the class would result to overwriting variable `sentinel.prev` with a new value, thus determining the infeasibility of the requirement regardless of the feasibility of both the definition and the use.

The experimental results in the next section provide empirical data about the impact of the phenomena illustrated in this section on dataflow testing criteria.

III. Experiment

In this section we present the results of the experiment that we conducted to evaluate dataflow testing criteria. We introduce the main research questions, illustrate the experimental setting and discuss the results as well as the threats to the validity of the experiment.

A. Research Questions

We design our study around three main research questions:

- Q1 To what extent can we satisfy dataflow testing criteria, to the best of manually designed (functional) test cases and state-of-the-art test case generation tools?
- Q2 Is there significant difference between the extents to which we can satisfy dataflow and controlflow (statement and branch) coverage criteria?
- Q3 To what extent does the difficulty of satisfying dataflow testing criteria depends on infeasible statements or branches, and to what extent does it depend on the nature of dataflow criteria?

Question Q1 addresses the conjecture that dataflow testing entails requirements that are difficult to satisfy. Questions Q2 and Q3 deal with the nature of the difficulties of covering dataflow testing requirements. In particular question Q2 focuses on the relation between statement/branch and dataflow coverage, while question Q3 can indicate whether dataflow testing criteria call for dedicated solutions.

TABLE I: Subjects and test suites

Application	ELOC	#Classes	Number of test cases: Bundled	Random	EvoSuite
JFreeChart	55 k	619	2022	11999	12463
Collections	13 k	444	12836	2903	4865
JGAP	15 k	419	1398	2281	3130
JTOPAS	3 k	63	209	1200	383
LANG	11 k	150	2051	5013	4212
XMLSEC	10 k	180	89	6458	1062
Totals	107 k	1875	18605	29854	26115

ELOC is the number of executable lines of code computed with Cobertura.
JFreeChart is a Java chart drawing package, http://www.jfree.org/jfreechart
Collections is the Apache Commons Collections, http://commons.apache.org
JGAP is a Java genetic programming framework, http://jgap.sourceforge.net
JTOPAS is Java text parsing library, http://jtopas.sourceforge.net
LANG is the Apache Commons Lang, http://commons.apache.org/lang
XMLSEC is the Apache Xml Security for digital signature and encryption, http://santuario.apache.org/javaapi.html

B. Variables and Treatment

The dependent variable of our experiment is the extent to which we are able to satisfy the dataflow testing requirements for a class under test. We measure this variable according to the dataflow class testing criterion proposed by Harrold and Rothermel that aims to capture the dataflow interactions that arise when users of a class invoke sequences of methods in arbitrary order [17]. The criterion formalises the notion of test adequacy for a class as executing all the *du-pairs* related to that class. A du-pair of a class is a pair of definition and use program locations such that:

1) They are both reachable when executing some method of the class;
2) Executing the definition (resp. the use) location results to writing (resp. reading) a datum in and from the same memory place;
3) There exist a program that calls the methods such that it executes the definition and then the use location, without modifying the shared datum in between.

Specifically we measure the extent of satisfaction of this criterion as the portion of du-pairs executed by a test suite. We refer to this measure as the *du-pair coverage rate*. We measure the du-pair coverage rate by means of the tool DaTeC developed in previous work [19]. DaTeC embodies a static analyzer that computes the set of possible du-pairs related to the field members of the classes under test, and a runtime monitor that detects the du-pairs executed during testing. For control purposes, in our experiment we measures also the statement and branch coverage rate, defined as the executed portion of the lines of code and the branches of a class, using Cobertura [http://cobertura.sourceforge.net].

The independent variables are the subject classes and the test suites run against those classes. Table I outlines statistics of the subjects and the test suites. As subjects, we considered all class modules from 6 open-source Java applications, ranging between 3,000 and 55,000 executable lines of code per application, resulting to a sample of 1,875 classes.

For these subjects we ran a total of 74,574 test cases. We designed the test suites trying to maximise the thoroughness of

the test suites according to the state of the art. First, we considered all test cases bundled with the subject applications. Open source development processes foster development of test cases along all the life cycle of an application, and sharing of those test cases among the communities of developers. Typically, the test cases are maintained within the application packages. Second, we generated test cases with a set of automated test generation approaches, selecting available tools and research prototypes that could handle generating test cases for Java classes at the state of the art. We generated test cases with tools based on random testing (Randoop [22] and CodePro AnalytiX [23]), and search-based testing (EvoSuite [24]). We configured Randoop and EvoSuite with time limits of 120 seconds per application and 180 seconds per class, respectively, and run CodePro in heuristic-based mode. In Randoop we set a maximum length of 300 lines of code per generated test case. Randoop and Evosuite completed successfully the test generation task for all applications, while CodePro worked only for Jtopas and XmlSecurity. These configurations correspond to test suites that produce high statement and branch coverage figures across all our subjects, as discussed in details in Section IV. To evaluate the effects of the variability associated with the random-nature of the techniques, we ran Randoop and EvoSuite several times, without seeing significant variations in the coverage measures.

The test suites were generated and executed on a OSX 10.7 MacBook Pro equipped with 2.2 GHz Intel Core i7 and 8GB of RAM memory. We did not plan precise measures of the time spent in each phase of the experiment, since it is not relevant for our experiment. Large applications took some hours for us to go through the whole process.

C. Preliminary Analysis of the Subjects

Table II summarises the statistics related to the subject classes. The sample includes a total of 1601 non-zero-size classes. The remaining 274 zero-size classes are due to entirely abstract modules (for example, interfaces) that contain declarations of method signatures but no executable code. The analysis of the distribution of the executable lines of code across the classes reveals that most classes are small sized, as expected for well architected object-oriented designs. (Hereafter we use the term statement as synonym of executable line of code.)

The table also reports the statistics on the distribution of the du-pairs per class, as computed by DaTeC. There can be classes that do not define any member field, and thus do not originate any du-pair. These classes do not originate requirements for dataflow testing, and we do not further consider these classes in our experiment. In the end, our sample includes 1169 non-zero du-pairs classes, which we regard as a statistically significant number of observations. The analysis of the distribution of the number of du-pairs across the classes reveals that most classes have a manageable number of du-pairs; 75% of the classes in the sample have less than 62 du-pairs.

TABLE II: SIZE STATISTICS OF THE SAMPLE CLASSES.

	#Non-zero		eloc			branch			DUP		
Application	eloc	DUP	q1	m	q3	q1	m	q3	q1	m	q3
JFreeChart	513	420	24	55	120	10	24	56	10	29	142
Collections	417	283	8	16	31	2	6	21	3	11	50
JGAP	322	256	11	22	53	0	6	25	4	9	33
JTOPAS	43	37	20	29	54	4	8	28	5	10	25
LANG	143	99	10	27	57	2	8	37	3	6	38
XMLSEC	163	74	10	30	63	6	14	34	7	20	68
All classes	1601	1169	11	27	69	4	12	36	5	15	61

Eloc is the number of executable lines of code and branch the number of branches, computed with Cobertura.
DUP is the number of du-pairs computed with DaTeC. q1 represents the first quartile of the sample, m the median and q3 the third quartile.

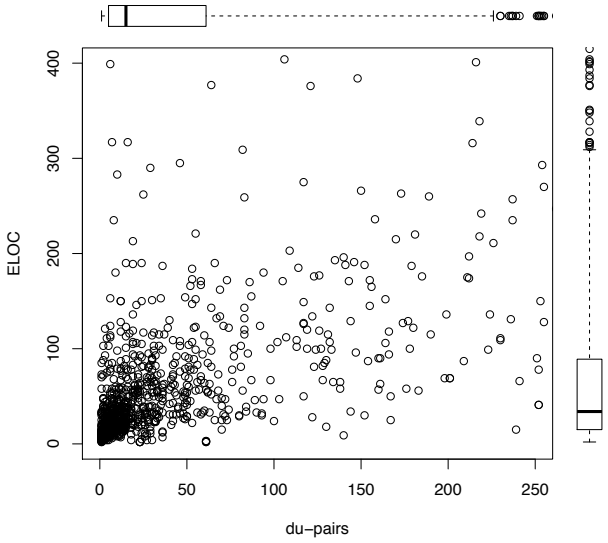
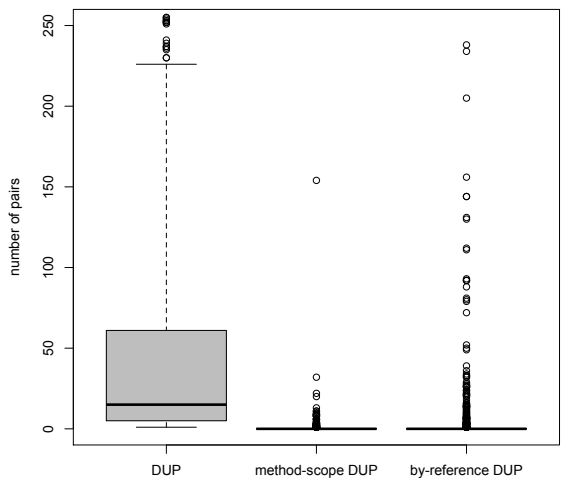


Fig. 1: Relation between du-pairs and lines of code

We investigated the relationship between the number of du-pairs and the number of statements. Figure 1 presents the scattergraph of the number of du-pairs and statements per class, which does not highlight particular trends between the two values. In fact, a correlation test could not find statistical evidence of correlation.

Finally, to better understand the nature of the du-pairs in the sample, we measured the number of du-pairs by distinguishing between: 1) du-pairs with definition and use within different (possibly different invocations of the same) methods, referred to as *class-scope du-pairs*, and du-pairs that fully belong to the static inter-procedural control-flow of single methods, referred to as *method-scope du pairs*; 2) du-pairs that directly relate the primitive attributes of the class under test, referred to as *by-value du-pairs*, and du-pairs that relate variables accessed through class attributes of reference types, referred as *by-reference du-pairs*. Intuitively, class-scope and by-reference du-pairs shall be more challenging for dataflow testing than method-scope and by-value ones, respectively.

Figure 2 shows that method-scope and by-reference du-pairs have a scarce impact across the classes in our sample. The datum on method-scope du-pairs was somehow expected, since methods that access class attributes either assign or read their values, but rarely do both operations. Our sample is well



Distribution values are as follows. DUP: median = 15, 3rd quartile = 61, maximum = 60421. Method-scope DUP: Maximum = 154, all other statistics = 0. By-reference DUP: maximum = 2316, all other statistics = 0.

Fig. 2: Distributions of du-pairs scoped in methods or accessed by reference

representative of class-scope du-pairs.

The datum on by-reference du-pairs reveals a limitation of the static analysis to deal with polymorphic calls through references to abstract (or interface) types. When statically analyzing these calls, DaTeC cannot identify the actual target of the invocations (potentially any subclass of the type of the references) and resolves them as no-effect calls. Inspecting the code of the considered applications, we have found many class attributes of reference types that are impacted by this limitation. In fact, several common object-oriented design patterns foster class attributes of abstract types for reuse purposes. In the general case, we shall expect more remarkable figures for the number of by-reference du-pairs than across our sample. Our experiment confirms with preliminary data that by-reference du-pairs challenge dataflow testing even more than by-value ones (see next section). On this basis, we believe that our experiment misses most by-reference du-pairs and thus induces an optimistic bias on the conclusions about the extent to which dataflow testing requirements can be actually satisfied.

D. Threats to Validity

The du-pair coverage rates that we measured in our experiment depend on both the test cases that we specifically executed and the coverage domain computed with DaTeC. We have executed both the test cases bundled with the subject applications and additional ones generated according to either random or search-based testing. There is no indication that the bundled test cases have been built with du-pairs in mind, and neither the random nor the search-based tools used in the experiment address dataflow criteria directly. This could partially explain the low du-pair coverage rates that we observed in the experiment. Our results suggest that there is large room for improvement and almost no supporting tools for dataflow coverage metrics.

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. Conservative choices produce infeasible elements of the coverage domain. Approximations produce incomplete coverage domains. We have already commented that our dataset likely misses several *by-reference* du-pairs that cannot be identified with DaTeC, and that this lack likely induces an optimistic bias on the rates of du-pair coverage measured in the experiment.

Our experiment measures dataflow testing adequacy according to the du-pair coverage criterion. For dataflow testing of object oriented software, du-pair coverage is a well acknowledged approach in the scientific literature. There exist other criteria, for instance the ones proposed by Alexander et al. [20]) that we plan to investigate as future work. The main barrier to extend our experiment over other dataflow coverage criteria is the lack of publicly available tools to automatically measure test adequacy accordingly.

Our subjects are limited to open source applications, and we shall thus restrict the scope of our conclusions to open-source software applications. In general, we are aware that the results of a single scientific experiment cannot be directly generalized, but we are confident that the data reported in this paper can shed new light on dataflow testing criteria.

IV. Results

We executed all bundled and generated test cases for the subject applications, and collected coverage data as described above. Figure 3 plots the cumulative frequency distribution of the classes for which we achieved less than a given coverage rate after executing all available and generated test cases, according to statement, branch and du-pair coverage, respectively. We observed significantly large portions of classes with low rates of du-pair coverage (highest curve), while the portions of classes with low rates of branch and statement coverage (middle and lowest curve, respectively) are consistently small across the sample.

To answer the research question Q1, we aim to establish the maximum rate of du-pair coverage that can be achieved with statistically significant evidence. Thus, we examined our dataset to find the minimum coverage rate X ($0 \leq X \leq 100\%$) for which we can support the alternative hypothesis ($H1_A^X$) with respect to the null-hypothesis

$$H1_0^X \equiv \mu_{dur} \geq X\%$$

where μ_{dur} denotes the mean rate of du-pair coverage per class achieved in our experiment. Being able to reject this null-hypothesis means that there is statistically significant evidence that on the average the rate of du-pair coverage per class is at most X%.

Operatively, we start with testing the null hypothesis $H1_0^{100}$, and then test the hypothesis for incrementally smaller values

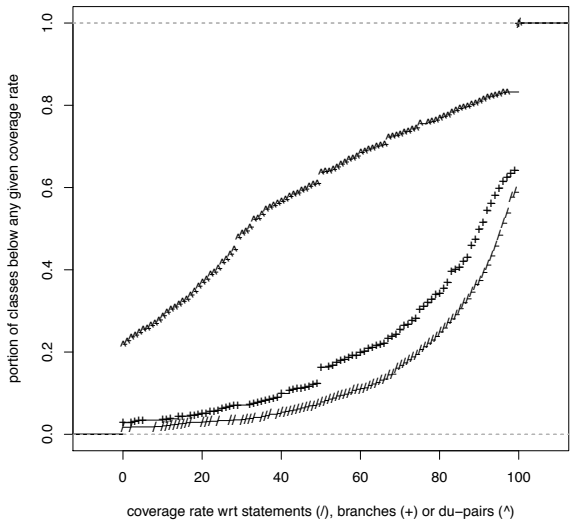


Fig. 3: Cumulative frequency distribution of the classes with less than the given (statement, branch, du-pair) coverage rate

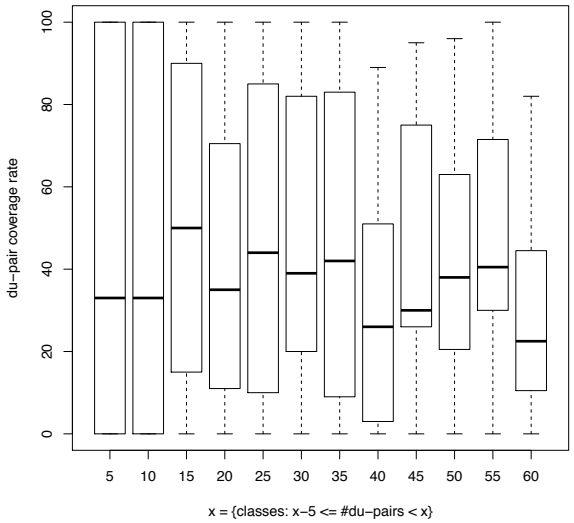


Fig. 4: Boxplots of the du-pair coverage rates for groups of classes with step-5 increasing number of du-pairs

of the coverage rate. We found support (p-value 0.018)[1] for H_A^{44} (while we could not reject H_0^{43}).[2] In summary:

> Our experiment supports with statistically significant evidence that, using the testing techniques encompassed in this paper, we achieve (on average) du-pair coverage rates per class less than 44%.

To crosscheck whether this conclusion can significantly depend on the number of du-pairs in the sampled classes, we made a further test after grouping the classes in the sample according to the number of du-pairs. We formed class groups with step-5 by increasing the number of du-pairs, that is, classes with 1 up to 4 du-pairs in the first group, classes with 5 up to 9 du-pairs in the second group, and so forth. We deliberately considered only classes with less than 60 du-pairs (up to the 3rd quartile of the sample) to avoid outliers and singular groups. Figure 4 summarizes the distribution of the du-pair coverage rates across the groups, which does not highlight special trends. In fact, a one-way analysis of variance (ANOVA) test did not reject the null hypothesis that the mean coverage rate is the same across all groups.

We controlled the impact of by-reference du-pairs on the current coverage figures. Our dataset includes 180 classes with at least 1 by-reference du-pair, and in 101 cases, our experiment does not cover any by-reference du-pairs in those classes. Overall we covered only the 10% of the by-reference du-pairs in the whole dataset. Although this datum is not statistically significant per se, it can indicate a negative impact of by-reference du-pairs on the coverage of the classes that include these du-pairs in their coverage domain, and that probably the du-pair coverage rates of the classes in our sample are worse than the ones measured in this experiment.

To investigate research question Q2, we study the relation between du-pair coverage and statement or branch coverage across our sample. Figure 5 shows the scattergraph of the considered classes with respect to the covered du-pairs and statements; the scattergraph with respect to the covered du-pairs and branches shows similar trends and is not reported for space reasons. We observe the intuitive relation that higher du-pair coverage rates entail higher statement coverage rates, while the vice-versa is not necessarily true. Statistically, we tested the null- and alternative hypothesis that

$$H2_0 \equiv \mu_{dur} \geq [\mu_{sr} | \mu_{br}]$$
$$H2_A \equiv \mu_{dur} < [\mu_{sr} | \mu_{br}]$$

where μ_{dur}, μ_{sr} and μ_{br} are mean rates of du-pair, statement and branch coverage per class, respectively.

We found several sources of evidence for the validity of the alternative hypothesis. With the procedure already described to test the rate of du-pair coverage, we found statistical evidence that statement coverage is on average above 85% and branch coverage is on average above 78% across our sample. A paired Student's t-test supports that the mean of the du-pair coverage rates significantly differs from the mean of the statement or the branch coverage rates (p-values $< 2.210^{-16}$). A correlation test supports that across the classes in the sample the du-pair coverage rate only slightly correlates with either the statement or the branch coverage rate (estimated correlation indexes are 0.35 and 0.36 respectively, while statement and branch coverage rates exhibit 0.88 correlation between them). In summary:

> Our experiment supports with statistically significant evidence that the testing procedure encompassed in this paper achieves (on average) consistently lower du-pair coverage than statement/branch coverage per class, with low correlation between the respective trends.

[1] In this paper we use 5% as level of statistical significance, that is, we require p-value < 0.05; p-values are according to the Student's t-test distribution, which applies for normally distributed data. The Shapiro-Wilk test supports the normality of the du-pair coverage data with W-value = 0.87 (p-value $< 10^{-15}$).

[2] We also repeated the test after excluding the classes that could be identified as outliers with respect to the number of du-pairs in our dataset, and still fond support for H_A^{46}.

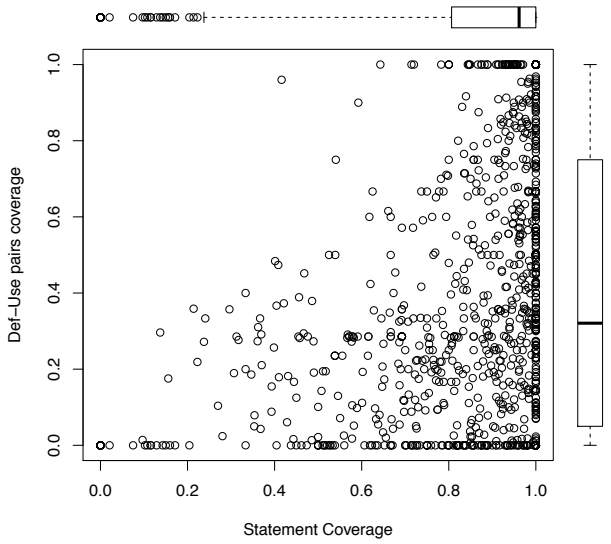


Fig. 5: Scattergraph of the classes with respect to their du-pair and statement coverage rates

To investigate the research question Q3, we further analyzed whether the du-pairs that were not covered across our sample relate to the uncovered statements. We classified all du-pairs such that the associated definition or use either 1) corresponds to an uncovered statement or 2) involves a never executed inter-procedural call. The result proportion over the whole number of uncovered du-pairs is between 16% (if we consider the entire dataset) and 22% (if we consider the dataset after removing the classes with outliers with respect to the number of du-pairs). In summary:

> The reachability of the missed du-pairs can be directly related to the reachability of some missed statement in a very limited number of cases.

V. Conclusions

This paper has described an experiment to study the difficulty of satisfying dataflow testing requirements. The reported data provide some quantitative empirical evidence beyond anecdotical cases that it is indeed difficult to achieve high dataflow coverage, and that dataflow coverage is largely independent and inherently more complex than the simple control flow coverage metrics commonly used in industry. This suggests room to complement existing testing approaches by extending the thoroughness of the test suites on the side of dataflow elements.

Overall the results of this paper motivate a renewed interest in dataflow testing, which is in fact the main focus of our ongoing research. In particular, we are currently investigating new approaches and tools to automatically generate test suites that achieve consistently high dataflow coverage, possibly with increased focus on by-reference du-pairs and ability to handle infeasible du-pairs, to enable software engineers to exploit this challenging yet promising facet of the thoroughness of testing.

References

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

[2] M. Baluda, P. Braione, G. Denaro, and M. Pezzè, "Structural coverage of feasible code," in *Proceedings of the 5th Workshop on Automation of Software Test*. ACM, 2010, pp. 59–66.

[3] C. Cadar, D. Dunbar, and D. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2008, pp. 209–224.

[4] United States, *RTCA, Inc., Document RTCA/DO-178B*. U.S. Department of Transportation, Federal Aviation Administration, Washington, D.C., 1993.

[5] 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.

[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, 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 *Proceedings of the 16th International Conference on Software Engineering*. IEEE Computer Society, 1994, pp. 191–200.

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

[9] 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.

[10] E. J. Weyuker, "The complexity of data flow criteria for test data selection," *Information Processing Letters*, vol. 19, no. 2, pp. 103–109, 1984.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] N. Malevris and D. Yates, "The collateral coverage of data flow criteria when branch testing," *Information and Software Technology*, vol. 48, no. 8, pp. 676–686, 2006.

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

[18] 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.

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

[20] 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.

[21] M. N. Ngo and H. B. K. Tan, "Detecting large number of infeasible paths through recognizing their patterns," in *Proceedings of the 11th European Software Engineering Conference joint with the 15th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2007, pp. 215–224.

[22] 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.

[23] "Google codepro analytix," https://developers.google.com/java-dev-tools/codepro/doc/.

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