2015 IEEE International Conference on Software Quality, Reliability and Security

How Effective Are Code Coverage Criteria?

Hadi Hemmati
Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada
hemmati@cs.umanitoba.ca

***Abstract*—Code coverage is one of the main metrics to measure the adequacy of a test case/suite. It has been studied a lot in academia and used even more in industry. However, a test case may cover a piece of code (no matter what coverage metric is being used) but miss its faults. In this paper, we studied several existing and standard control and data flow coverage criteria on a set of developer-written fault-revealing test cases from several releases of five open source projects. We found that a) basic criteria such as statement coverage is very weak (detecting only 10% of the faults), b) combining several control-flow coverage together is better than the strongest criterion alone (28% vs. 19%), c) a basic data-flow coverage can detect many undetected faults (79% of the undetected faults by control-flow coverage can be detected by a basic def/use pair coverage), and d) on average 15% of the faults may not be detected by any of the standard control and data-flow coverage criteria. Classification of the undetected faults showed that they are mostly to do with specification (missing logic).**

I. Introduction

Software testing is one of the most practical approaches for assuring the quality of software systems. One of the challenges in software testing is evaluating the quality of test cases, so that a higher quality test suite detects more faults. Such quality measures are called test adequacy criteria [1]. The commonly used type of test adequacy criterion is called code coverage [2]. Most existing automated test generation tools [3] use code coverage as their quality evaluator. The tools try to generate test cases that cover 100% (or as close as possible to that) of the source code. In software industry, code coverage is also used as a product quality measure. For instance, a team may set up a coverage threshold of 80% as their minimum, which means that they won't release a product unless they have tested it with a minimum of 80% coverage. Given the widespread usage of code coverage in both academia and industry, we set up a study to investigate its effectiveness, in terms of revealing faults.

The idea of the study is to see whether the existing standard and commonly used code coverage criteria are effective enough to identify the real faults. In other words, how much can we rely on code coverage and what type of faults we may miss, even if we have a full coverage? To answer this question, in this paper, we looked at 274 faults of five open source systems and their corresponding failing test cases. We specifically focused on the changes in each test class (modification of a test method or adding a new test method) that resulted in detecting the fault (failing test cases). We then applied four common control flow (statement, branch, MC/DC, and loop) and one data flow (all def-use pairs) coverage criteria on the failing test classes before and after the change, to see whether improving the test class to detect the fault has also improved the code coverage or not. The results showed that 7%–35% of the improvements did not change the coverage of the test class. In other words, even if the test suite satisfies a 100% code coverage, using all of the five mentioned criteria, 7% to 35% of the faults may still be undetected.

We then categorized those undetected faults using a grounded theory approach, card sorting [4], and found that the most common missing faults are the specification-related faults, which could not be detected by code coverage. For example, if a condition is not implemented in the source code, even a 100% coverage of the code can not reveal the fault. Given the relatively large portion of faults that may not be detected by code coverage, and the fact that most test generation tools only focus on maximizing the coverage, the findings of this paper can be a motivation for the testing research community to consider other (or complementary) techniques than (to) code coverage.

II. Background

In the rest of this section, the coverage criteria that are used in this paper have been defined.

A. Control Flow Code coverage

The basic idea of control flow criteria is to examine the execution flow of test cases in the code.In this paper, we use the following common control flow coverage criteria that are implemented in the CodeCover tool [5]:

Statement Coverage Informally speaking, to reach full statement coverage, each statement of the program code has to be executed at least once [5]. One can also measure the extent in which the statements are being covered, i.e., the portion of all statements that are being executed by running the test cases, if the coverage is not 100%. Statement coverage (or block coverage) is usually the weakest adequacy criteria and yet the most used one in practice [1].

Branch Coverage The branch coverage examines all branches of the program (both true and false cases of each decision point in the code). Branch coverage subsumes statement coverage. In other words, it is a stronger adequacy criterion [2].

MC/DC Coverage Each decision point in the source code can be an atomic condition or a boolean expression composed of several atomic conditions. Depending on the concrete values of the involving variables in the the conditions, a final decision is evaluated as $true$ or $false$; which covering both will satisfy the branch coverage. However, covering the $true$ and $false$

978-1-4673-7989-2/15 $31.00 © 2015 IEEE
DOI 10.1109/QRS.2015.30

results does not guarantee covering all true and false cases of all sub-conditions, as well. A refined version of Branch coverage is called Condition-Decision Coverage (C-DC) and it makes sure that not only the branch has been tested for $true$ and $false$, but also each sub-condition is fully covered.

Another practical criterion is called Modified Condition-Decision Coverage (MC/DC) [1]. This criterion covers only the combinations of atomic conditions where the value of the atomic condition can affect the overall decision, independently. In other words, the outcome of a compound decision changes as a result of changing each single condition. CodeCover provides a coverage metric called "Term Coverage", which implements MC/DC.

Loop Coverage Finally, we also use the loop coverage metric implemented in CodeCover. Loops are error-prone, e.g. off-by-one errors in *for* loops. The loop coverage helps carefully testing of loops, and shows, whether each loop has been repeated more than once [5]. Note that not covering a loop body can be detected by statement/branch coverage. Running the loop body at least once can also be covered by branch coverage, but making sure that the loop has been executed more than once requires loop coverage.

B. Data Flow code coverage

Data-flow coverage criteria are based on the investigation of the ways in which values are associated with variables and how these associations can effect the execution of the program [2]. This analysis focuses on the occurrences of variables within the program. Each variable occurrence is classified as either a definition occurrence (def) or a use occurrence (use). A definition occurrence of a variable is where a value is bound to the variable. A use occurrence of a variable is where the value of the variable is referred.

def-use pair coverage In short, def-use pair coverage criteria examine whether a use of a variable has been properly tested with respect to its different possible definition points. The most common approach for monitoring the def-use pairs per variable is a technique known as "last definitions". Under this approach, each definition *d* of a variable *v* that is executed is recorded. Then, when a use *u* of that variable is executed, the last *d* recorded is the definition that reaches *u*, and the $DUA(d, u, v)$ pair is marked as being satisfied [6]. We use an implementation of this algorithm in DUA-FORENSICS tool [7].

Note that, in general, data flow coverage are expected to be stronger than the control flow coverage criteria, but they are also more expensive to run [6].

III. Empirical Study

In this section we describe the design and results of our empirical study.

A. Research questions

The main objective of this study is to assess the ability of typical code coverage measures in revealing faults. To achieve this goal, we have designed an experiment to answer the following research questions:

TABLE I. Systems under test

	Program	KLOC	Test KLOC	Tests
Chart	JFreeChart	96	50	2,205
Closure	Closure Compiler	90	83	7,927
Math	Commons Math	85	19	3,602
Time	Joda-Time	28	53	4,130
Lang	Commons Lang	22	6	2,245
Total		321	211	20,109

RQ1: How effective are the typical code coverage criteria for evaluating test cases?

To answer this question, we first look at four typical control-flow coverage criteria (statement, branch, MC/DC, and loop coverage). For each criterion individually, and for all criteria combined, we answer the following questions:

- RQ1.1. How effective are control-flow criteria, for evaluating test cases?
- RQ1.2. How much a typical data-flow coverage can add to the effectiveness of control flow criteria?

Finally, for the remaining undetected faults we are interested in knowing:

RQ2: What types of faults may remain undetected, after applying the given control and data-flow coverage criteria?

B. Subjects of study

In this experiment, we study test cases from five open source Java projects: JFreeChart (Chart), Closure Compiler (Closure), Math Commons (Math), Joda-Time (Time), and Lang Commons (Lang). The projects data are extracted from a database called defects4j [8]. defects4j has already extracted JUnit test cases of these projects and has isolated the faults per project as well. Table I summarizes the information about the systems under study.

C. Data extraction procedure

The defects4j database provides a set of bug-fixing revisions, per project. Each revision comes with two sets of test cases. The first set is the passing test suite. All test cases of this set pass on the faulty code. The second set of test cases is called the failing test suite. The failing test suite is a modified version of the passing test suite, after some improvements in the form of adding new test cases or modifying some existing tests. When the test cases are improved and the fault is detected the source code will be fixed. Therefore, each revision includes a faulty and a fixed version of the source code. There is also a passing and failing test suite for the faulty version, per revision (the failing test suite will be passed on the fixed version of the code).

The exact details of extracting such revisions' data are outside of the scope of this paper, since we mainly reuse the defects4j dataset and do not extract the revision information by ourselves. However, the interested readers are referred to [8] for more details about the revisions' data extraction process. It worth mentioning that the bug-fixing revisions are not necessarily matched with the products' releases. These are internal working versions after each bug-fix. The reason

that we use these revisions is that we are interested in all individual faults in an isolated way. If we would extract the actual releases' data there could be multiple faults per revision, which would make the analysis of individual faults difficult.

To study the code coverage effectiveness, we looked at the entire 357 bug-fixing revisions and 20,109 test classes available in defects4j. All test classes are written by the projects' developers and all faults are real bugs reported and fixed in the projects. These bugs are reproducible and isolated (i.e., there is only one fault per revision). There might be several modifications in the test suite from a revision to the next one and among them there is at least one test method in a test class that fails in the failing test suite and passes in the passing suite.

D. Design

To answer our first research question, we calculate code coverage, per revision. We call a coverage criterion effective or sufficient for detecting a fault if any test suite fulfilling the criterion guarantees the detection of the fault. To assess this sufficiency, for a given coverage criterion A, we could create a test suite with 100% A coverage and see whether it detects the fault or not. However, we opted not to create artificial test cases and use only developer written tests, which might not satisfy 100% A coverage. Therefore, in this experiment, we look at the relative code coverage changes in a test suite before and after improvement. Recall from the data extraction procedure that each revision contains exactly one fault, which is first undetected by the passing test suite and then (after the test suite improvement) is detected by the failing test suite.

This way, the coverage criterion A is not suitable for identifying the given fault, if the A coverage is the same when using the passing and failing test suite, on the faulty code. For example, assume we have a faulty statement in our code and the passing test suite already executes the statement without failing. The improved test suite, however, runs the very same statements (probably with different variables values) and detects the fault. In this situation, the test suite's statement coverage has not changed but the fault can only be detected by the failing test suite. This means that statement coverage does not guarantee finding this fault and is not sufficient/effective.

It is important to notice that this is an optimistic approach. In other words, if the coverage does not change we are sure that it is not effective, but if it changes we still can not guarantee that this was not by chance. This is a natural characteristic of a coverage-based metric. So all our analysis and statistics, provided in the results, are upper-bound of effectiveness per coverage criteria.

We applied the same process on the 357 revisions (357 isolated real faults) of the five java projects. We used CodeCover tool [5] to instrument the source codes to capture the test cases' control flow coverage. The tool supports several criteria among which we selected the most common ones that also match with the basic testing adequacy criteria from textbooks, namely Statement, Branch, MC/DC (called Term coverage in the tool), and Loop coverage criteria.

To get the data-flow coverage information, we used an academic tool called DUA-Forensics [7], which is an open source Java analysis and instrumentation framework and is built on top of the Soot Analysis Framework [9]. DUA-Forensics works on the bytecode level, thus we provided .class files of the projects to it for instrumentation. We then ran the test suites on the instrumented .class files to produce data flow measurements. The tool provides several code analysis features among which we used its data-flow analysis feature. The DU measure that DUA-Forensics tool provides matches with the basic def-use pair data-flow coverage. We only applied data-flow analysis on revisions where no control-flow criteria were effective in detecting the faults.

To answer the second research question, we focused on the revisions where neither control nor data-flow coverage criteria were effective. For these revisions, we first extracted the bug-fix related to the current fault, by comparing the source code before and after the fix. Then we applied a grounded theory approach called card sorting [4] on these faults (we would first understand the fault by looking at the fix and then categorize the fault). First the author together with an undergraduate student, who was working on this project, did all the categorizations, independently. Then, together, in several meetings, we merged the two into a final categorization.

E. Results

As discussed, we ran the two versions of test suites (passing and failing) on the faulty code of each revision and calculate the control/data-flow coverage info using the CodeCover/DUA-Forensics tools. However, running these tools on our projects did not always go smoothly. There was a small set of cases (14 revisions of Lang project), where CodeCover crashed when trying to get coverage results. Lack of concise error messages and developer support made debugging efforts in this case futile.

We experienced more problematic cases when applying DUA-Forensics. DUA-Forensics deals with .class files directly, which in some cases may produce faulty .class files. For example, there were cases where class methods became too big after instrumentation. However, JVM limits the size of generated Java bytecode for each method in a class to the maximum of 64K bytes. This limitation will cause the JVM to throw $java.lang.VerifyError$ at runtime when the method size exceeds this limit [10]. While corruption of .class files might be fixable by modifying the source to eliminate the resulting problem, we rejected this idea because a) we wanted to keep the source as the developer wrote it, and b) modification of source files might compromise the data we are trying to collect. Revisions of a project in which these problems occur were omitted and not used in our experiments. As a result, we have omitted one revision from Chart, 23 revisions from Closure, and 45 revisions from Math, all due to problems related to DUA-Forensics.

Therefore, the total number of accepted revisions in our experiment was 274 revisions. Table II summarizes the coverage results of these 274 revisions, per project. First, for each control-flow coverage, we report the number/percentages of cases (out of the total working revisions per project) where the passing and failing test suites had different coverage values using the given criterion. We then combine this knowledge and report the total number/percentages of revisions, per project, where at least one of the control-flow coverage criteria is effective in detecting the fault.

TABLE II. SUMMARY OF THE COVERAGE CRITERIA EFFECTIVENESS, IN FIVE SUBJECTS OF THE STUDY.

	Chart	Closure	Math	Time	Lang	**Total**
Total # of faults	26	133	106	27	65	357
# of CodeCover/DUAF problematic cases	0/1	0/23	0/45	0/0	14/0	14/69
Total # of studied faults	**25**	**110**	**61**	**27**	**51**	**274**
	% of detected faults					
Statement	32%	5%	10%	0%	14%	10%
Branch	32%	18%	18%	11%	18%	19%
MC/DC	24%	18%	18%	11%	25%	19%
Loop	12%	5%	18%	0%	8%	8%
All Control-flow	**44%**	**24%**	**33%**	**15%**	**29%**	**28%**
# of undetected faults by control flow criteria	14	84	41	23	36	198
	% of detected faults					
def-use (DUA)	86%	87%	80%	91%	50%	79%
Data & control-flow	**92%**	**90%**	**87%**	**92%**	**65%**	**85%**

TABLE III. NUMBER OF THE UNDETECTED FAULTS, PER CATEGORY/PROJECT.

	Chart	Closure	Math	Time	Lang	Total
Environment	0	1	0	1	0	2
Integration	0	0	1	0	1	2
Combinatorial	1	1	0	0	0	2
Specification	1	9	7	1	17	35
Total	2	11	8	2	18	41

RQ1: Answering RQ1.1, the first observation is the low number of cases where control-flow coverage (all four together) is effective. There are only 76 faults out of the total 274 faults (28%) where at least one of the control-flow coverage criteria is effective in detecting the fault. This translates into 56% to 85% of the faults remaining undetected per project. Therefore, an immediate follow up study would be examining the potential improvements that data-flow coverage may bring in, which is discussed in RQ1.2.

Looking at the individual control-flow coverage values, we can see that the MC/DC and branch coverage criteria perform better than statement and loop coverage (19 and 19% vs 10 and 8%), which is somewhat expected. However, the more interesting observation is that nevertheless a combination of control-flow coverage will always perform better than the best individual one (28% vs. 19%). The improvements may vary between 15% (in Lang where the best individual coverage, MC/DC, had detected 13 faults and the combined control-flow coverage has added 3 more to that) to 82% (in Math where the best individual coverage (there are several of them) had detected only 11 faults and the combined control-flow coverage has added 9 more faults to that). This finding suggests the use of multiple coverage criteria rather than focusing on maximizing just one.

Based on the results of RQ1.1, we realize that control-flow coverage criteria are not as effective as one would think (72% undetected faults after applying four common control-flow coverage criteria – and remember this is the upper-bound of their effectiveness). So in RQ1.2, we looked at the potential improvements that data-flow analysis can bring in. Table II also shows the number/percentages of faults that are detected by DU-pair coverage, but not with any of the studied control-flow coverage criteria, per project (Note that we only apply data-flow analysis on cases where control-flow is not effective).

Not very surprisingly, given its cost and complexity of calculation, DU-pair coverage fairs well. The percentages are anything between 50% to 91%. Recall that these are percentages of faults from the pool of undetected faults (after applying control-flow coverage) that are detected by our data-flow coverage criterion. This means that adding data-flow analysis to common control-flow analysis hugely benefits the overall effectiveness of coverage-based test evaluation strategies. However, there are still 41 faults out of 274 faults (15%), where neither control-flow nor data-flow coverage criteria are effective. These undetected faults range from 8% (Time and Chart) to 35% (Lang), in different projects. Therefore, in RQ2 we are interested in identifying and categorizing these special faults.

RQ2: After applying the card sorting approach, we managed to identify four high level categories of faults that are remained undetected by both control and data-flow coverage criteria:

a) Environment faults: The faults in this category are to do with external systems that interact with the software under test. We could also call them integration faults in the system level, but we kept the keyword "integration" for lower-level integrations, which will be discussed in the next category. We had exactly two cases of these types in Time and Closure projects. The first (Time) fault was about wrong data in a data file that the code was using to get IDs for time zones. In other words, the code was correct but the overall system was faulty. Code coverage was not effective because the wrong output would be supplied by the external data file and only using a specific time zone ID would reveal that.

The second fault was to do with the choice of operating system. The faulty method would take a file path string as an argument, convert the file name into a javascript module name and return the string. Because of the special way that Windows handles forward and backward slashes, this could cause a mistake when replacing slashes in a string. So again this fault cannot be detected even if the code is covered 100% by the test cases, using any studied criterion, if the OS is not Windows.

b) Integration faults: The faults in this category are related to the wrong usage of external libraries. We encountered two cases of these faults that were not detected by any coverage criteria. First fault was in Lang, where comparing *BigDecimal* objects with different scale but the same value would return false when it should return true (e.g., 2.0 and 2.00). This fault was to do with the wrong usage of the *equals* method of *BigDecimal* class, where *compareTo* method should have been used. Therefore, covering the program under test would not help finding the fault unless a specific input (such as 2.0 and 2.00) is tested, so that the external library returns the unexpected value. The second fault was on the Math project and it was quite similar to the previous example (a mistake in the usage of the external library).

c) Combinatorial faults: The faults in this category are a bit different than the previous two, since they could have

been detected with a stronger code coverage criterion (e.g., pair-wise coverage). We experienced two cases of these faults. The first fault was in Chart where an object called *XYSeries* had a list of *X* and *Y* values. In the *addOrUpdate* method depending on the existence of values of *X* and *Y* or both the algorithm would be different. The faulty code in this method would return -1, as index, only if both atomic conditions of the following compound *if* statement were false: *if(index* $\geq$*0 && !allowDuplicates)*

Note that our existing criteria can guarantee the execution of statement 4 (statement coverage) and the false branch of the if condition (branch coverage). It can also guarantee to test ($index \geq 0$) with True and False. And the same for $!allowDuplicates$ (MC/DC). However, it does not guarantee having both conditions false at the same time. This is a typical fault and could be detected with several more demanding coverage criteria such as all conditions coverage or pairwise coverage in this case (note that MC/DC is not sufficient here either). The second fault was in the Closure project and had a very similar nature (a complex condition was not tested with a specific faulty combination).

d) Specification faults: The last category is also the biggest one with 35 faults. These are faults that are also called "faults of omission" in the literature [11]. We had two sub-categories of these faults: "missing condition" and "wrong logic". In the first sub-category, all faults are fixed by adding an extra condition in the code. In other words, the faulty code was not differentiating the partitions of input space that are created by the extra condition. We call these types of faults specification faults because if the developers would follow the specification of the software they should have known about the expected behaviour differences in the two categories and thus would have the extra condition to create separate code for the different cases.

For example, a faulty revision of Math contains a method that converts an array of Objects to an array of Classes. This method did not account for $null$ objects. The fix added a $null$ check to the code. Notice that if the condition was there, MC/DC could have detected the fault but since there was no condition, any arbitrary object value was enough to pass the tests, without examining the $null$ case.

The second sub-category is "wrong logic" faults, where all projects, except Time, had at least one of them. The fixes of these faults all modify the code in a way that it represents different specification (calling different methods, modifying conditions, changing calculations, or completely modifying the implemented logic). The corresponding passing test suite could not detect the faults because the modifications added new code that, in the previous version, did not have a test for (these faults are somewhat similar to missing condition category but with usually a larger missing code). As an example of modifying the implemented logic, *getShortClassName* method in *ClassUtils* in the Lang project accepts a string with a class name and returns the class name minus the package name. The results were incorrect when supplied with array types. The fix simply added the logic to extract the short class name from arrays. The tests initially could not detect the fault because there was no test for arrays, even though they were covering the existing code properly.

Table III summarizes the number of faults per category. The full list of the 41 faults including the faulty code, the fixed code, the fix on the old and the new test class and their diffs, together with a short description of the fault, per revision/project is available online at the following address: http://sealab.cs.umanitoba.ca/?p=810

IV. Discussion on the implications of the results

As the results show, the effectiveness of the criteria increases from an average of 28% for four control-flow criteria to 85% when DUA coverage is also added to the list. This is a huge improvement, which should attract the automated test generation tool builders to accommodate data analysis into their objectives.

It is also important for practitioners to know the limitations of the control-flow coverage criteria, which are typically used as a quality indicator. For instance, statement coverage, which is the most common measure in most coverage tools is only capable of detecting 10% of the faults in our five systems (0% to 32%). This very low effectiveness should alarm companies that rely a lot on statement coverage.

The second important finding of this paper is the high percentages (85%) of specification-related faults among the total missing faults. What this tells us, in short, is that code coverage, of any kind, is going to fail on detecting this types of faults. In other words, to achieve 100% fault detection rate, a combination of specification-based (e.g., model-based testing) and coverage-based test generation is required. On the other hand, it also tells us that the coverage-based approaches, if properly used (using multiple criteria including data-flow criteria), are quite effective.

Note that we did not include fault-based adequacy criteria (e.g., mutation score) in this study. It is interesting to extend this study with fault-based criteria and see what portion of the undetected faults can be detected by mutation analysis.

V. Threats to validity

We have used commonly used metrics and employed exiting well-known tools to extract the metrics values per test suite execution, to eliminate internal threats with respect to implementation or selection of metrics. The raw data (test cases, bugs, and fixes) is also coming from a publicly available dataset, which already have other publications about.

In RQ2, however, the categorization is manual. There is some subjectivity that goes into such a process. We reduce that by having two people going through the categories individually to reduce the risk. Also we share our raw data to provide a means for replication and verification.

The main threat to the conclusions is the upper-bound analysis strategy used in this paper. Basically, the statistics provided per coverage criteria are all the upper-bound of effectiveness, since the coverage of an improved test case could change by chance (nothing to do with the fault), as explained in Section III-D, as well. The alternative to this strategy would be repeating each individual test case coverage analysis several times with different improved tests, which all detect the fault, and calculate the coverage of all those alternative improved tests. This would give us a better (still no guarantee) picture

of an average scenario, rather than the optimistic one. However, given our study design (only using existing developers-written tests) we did not have several fault revealing improved test cases per fault. Nevertheless, this only affects the relative evaluations between criteria (*e.g.*, the amount of improvements when adding data flow analysis in) and the general message of the paper (the ineffectiveness of basic control-flow coverage criteria and the missing faults categories) is still valid.

In terms of the external validity, we have used five commonly used Java open source programs, which are actively updated and have a live open source community around. The bugs are all real and our fault set is in a reasonable size (274). All test cases are developer-written and they are also in good size (20,109). Therefore, our results are to some extent generalizable. Of course, to get more confidence, one needs to replicate the study in many more programs including those from commercial systems. That's why we share all our raw data and results online. In addition, our process including the used tools are all well-documented, which makes the replication feasible.

VI. Related work

There have been many studies over the past 40 years that compare the effectiveness of different coverage criteria in terms of detecting faults. Coverage criteria have also been evaluated in the context of automated test generation and prioritization/selection.For example, in the context of model-based test generation, Briand *et. al.* [12] compared several UML statechart-based criteria. There are, however, fewer studies [13] that compare the effectiveness of different code coverage criteria in the context of coverage-based test generation strategies, where the coverage of the test may not be 100%. Gligoric, *et. al.* [14] called this non-adequate test suites and compared several control flow coverage criteria in this context.

Another set of related work to this study is about categorization of faults, as we do in RQ2. There are several studies of this type that categorize bugs [15]–[17]. For example, "persistent software errors" by Robert Glass [11] has the same type of findings, in terms of "omitted logic" being one of the most common difficult faults, as ours. In a more recent study Li *et. al.* [18], looked at two large scale open source system with 29,000 real bugs and automatically categorized them. They also found that semantic bugs are the dominant category of the difficult faults.

Overall, unlike the existing work, the first RQ of this paper investigates the effectiveness of five well-known control and data flow coverage criteria on detecting real bugs of five real-world open source systems. Our focus on the individual fault revealing test cases, differentiates our study from the related work. In terms of the RQ2, we only categorize the faults that are not detected by the common code coverage criteria. This is important because it gives us an overview of the difficult faults. Thus the practitioners are advised on looking into other options such as mutation score, historical data, or specification-based approaches to evaluate existing tests or generate new ones, if they wish to not miss such difficult faults.

VII. Conclusion and future work

Code coverage criteria are very popular metrics to evaluate the quality/adequacy of test cases. However, most practitioners and academics only use basic criteria, such as statement and branch coverage. In this study, we looked at the effectiveness of such criteria compared to other more advanced control and data-flow criteria such as MC/DC and def-use pair coverage, respectively. The results showed that the effectiveness of the metrics in identifying bug revealing test cases increases from an average of 10% for statement coverage to 19% for MC/DC and to 28% for all control-flow criteria together (the four that are used in this paper) and finally to 85%, when they are combined with data-flow coverage. We also found that the difficult faults (the remaining 15%) that remained undetected using all criteria are mainly missing logic faults, where a specification-based testing approach could be beneficial.

References

[1] A. P. Mathur, *Foundations of Software Testing, second edition*. Pearson Education India, 2008.

[2] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, 1997.

[3] M. Harman, "The current state and future of search based software engineering," in *2007 Future of Software Engineering*. IEEE, 2007, pp. 342–357.

[4] M. Miles and A. Huberman, *Qualitative Data Analysis: An Expanded Sourcebook*. SAGE Publications, 1994.

[5] "Codecover tool," http://codecover.org, last accessed: 2015-03-25.

[6] R. Santelices and M. J. Harrold, "Efficiently monitoring data-flow test coverage," in *IEEE/ACM International Conference on Automated Software Engineering*, 2007.

[7] R. Santelices, Y. Zhang, H. Cai, and S. Jiang, "Dua-forensics: A fine-grained dependence analysis and instrumentation framework based on soot," in *ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis*, 2013.

[8] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *International Symposium on Software Testing and Analysis*, 2014.

[9] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot-a java bytecode optimization framework," in *IBM conference of the Centre for Advanced Studies on Collaborative research*, 1999.

[10] "Method size limit in java," http://chrononsystems.com/blog/method-size-limit-in-java, last accessed: 2015-03-25.

[11] R. L. Glass, "Persistent software errors," *IEEE Transactions on Software Engineering*, no. 2, pp. 162–168, 1981.

[12] L. Briand, Y. Labiche, and Y. Wang, "Using simulation to empirically investigate test coverage criteria based on statechart," in *IEEE International Conference on Software Engineering*, 2004.

[13] M. Staats, G. Gay, M. Whalen, and M. Heimdahl, "On the danger of coverage directed test case generation," in *Fundamental Approaches to Software Engineering*, 2012.

[14] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov, "Comparing non-adequate test suites using coverage criteria," in *ACM International Symposium on Software Testing and Analysis*, 2013.

[15] R. Chillarege, W.-L. Kao, and R. G. Condit, "Defect type and its impact on the growth curve," in *IEEE international conference on Software engineering*, 1991.

[16] M. Sullivan and R. Chillarege, "Software defects and their impact on system availability: A study of field failures in operating systems," in *FTCS*, 1991, pp. 2–9.

[17] T. J. Ostrand and E. J. Weyuker, "The distribution of faults in a large industrial software system," in *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 4. ACM, 2002, pp. 55–64.

[18] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now?: an empirical study of bug characteristics in modern open source software," in *ACM Workshop on Architectural and system support for improving software dependability*, 2006, pp. 25–33.