Noname manuscript No.
(will be inserted by the editor)

Code Coverage Differences of Java Bytecode and Source Code Instrumentation Tools

Ferenc Horváth · Tamás Gergely · Árpád Beszédes · Dávid Tengeri · Gergő Balogh · Tibor Gyimóthy

Received: date / Accepted: date

Abstract Many software testing fields, like white-box testing, test case generation, test prioritization and fault localization, depend on code coverage measurement. If used as an overall completeness measure, the minor inaccuracies of coverage data reported by a tool do not matter that much; however, in certain situations they can lead to serious confusion. For example, a code element that is falsely reported as covered can introduce false confidence in the test. This work investigates code coverage measurement issues for the Java programming language. For Java, the prevalent approach to code coverage measurement is using bytecode instrumentation due to its various benefits over source code instrumentation. As we have experienced, bytecode instrumentation-based code coverage tools produce different results than source code instrumentation-based ones in terms of the reported items as covered. We report on an empirical study to compare the code coverage results provided by tools using the different instrumentation types for Java coverage measurement on the method level. In particular, we want to find out how much a bytecode instrumentation approach is inaccurate compared to a source code instrumentation method. The differences are systematically investigated both in quantitative (how much the outputs differ) and in qualitative terms (what the causes for the differences are). In addition, the impact on test prioritization and test suite reduction – a possible application of coverage measurement – is investigated in more detail as well.

Keywords Code coverage · white-box testing · Java bytecode instrumentation · source code instrumentation · coverage tools · empirical study

The final publication is available at Springer via http://dx.doi.org/10.1007/s11219-017-9389-z.

Ferenc Horváth · Tamás Gergely · Árpád Beszédes · Dávid Tengeri · Tibor Gyimóthy
Department of Software Engineering, University of Szeged, Szeged, Hungary
E-mail: {hferenc,gertom,beszedes,dtengeri,gyimothy}@inf.u-szeged.hu

Gergő Balogh · Tibor Gyimóthy
MTA-SZTE Research Group on Artifical Intelligence, University of Szeged, Szeged, Hungary
E-mail: {geryxyz,gyimothy}@inf.u-szeged.hu

1 Introduction

In software development and evolution, different processes are used to keep the required quality level of the software, while the requirements and the code are constantly changing. Several activities aiding these processes require reliable measurement of *code coverage*, a test completeness measure. As any other test completeness measure, it does not necessarily have a direct relationship to fault detection rate (Inozemtseva and Holmes, 2014), however code coverage is widely used and relied upon in several applications. This includes white-box test design, regression testing, selective retesting, efficient fault detection, fault localization and debugging, as well as maintaining the efficiency and effectiveness of the test assets on a long term (Pinto et al, 2012). Essentially, code coverage indicates which code parts are exercised during the execution of a set of test cases on the system under test. The knowledge about the (non-)covered elements will underpin various decisions during these testing activities, so any inaccuracies in the measured data might be critical.

Software testers have long established the theory and practice of code coverage measurement: various types of coverage criteria like statement, branch and others (Black et al, 2012), as well as technical solutions including various kinds of instrumentation methods (Yang et al, 2009). This work was motivated by our experience in using code coverage measurement tools for the Java programming language. Even in a relatively simple setting (a method level analysis of medium size software with popular and stable tools), we found significant differences in the outputs of different tools applied for the same task. The differences in the computed coverages might have serious impacts in different applications, such as false confidence in white-box testing, difficulties in coverage-driven test case generation, and inefficient test prioritization, just to name a few.

Various reasons might exist for such differences and surely there are peculiar issues which tool builders have to face, but we have found that in the Java environment, the most notable issue is how *code instrumentation* is done. Code instrumentation technique is used to place "probes" into the program, which will be activated upon runtime to collect the necessary information about code coverage. In Java, there are two fundamentally different instrumentation approaches: *source code* level and *bytecode* level. Both approaches have benefits and drawbacks, but many researchers and practitioners prefer to use bytecode instrumentation due to its various technical benefits (Yang et al, 2009). However, in most cases the application of code coverage is on the source code, hence it is worthwhile to investigate and compare the two approaches. In earlier work (Tengeri et al, 2016), we investigated these two types of code coverage measurement approaches via two representative tools on a set of open source Java programs. We found that there were many deviations in the raw coverage results due to the various technical and conceptual differences of the instrumentation methods. In this work, we have fine-tuned our measurements based on the previous results, examined and described the deviations in the coverage in more detail, and performed experiments and quantitative analysis on the effect of the differences. Similar studies exist in relation to branches and statements (Li et al, 2013).

Extending the earlier experiment, this work reports on an empirical study to compare the code coverage results provided by tools using the different instrumentation types for Java coverage measurement on the method level. We initially

considered a relatively large set of candidate tools referenced in literature and used by practitioners, and then we started the experiments with five popular tools which seemed mature enough and actively used and developed. Overall coverage results are compared using these tools, but eventually we selected one representative for each instrumentation approach to perform the in-depth analysis of the differences (JaCoCo[1] and Clover[2]). The measurements are made on a set of 8 benchmark programs from the open source domain which are actively developed real size systems with large test suites. The differences are systematically investigated both quantitatively (how much the outputs differ) and qualitatively (what the causes for the differences are). Not only do we compare the coverages directly, but investigate the impact on a possible application of coverage measurement in more detail as well. The chosen application is test prioritization/test suite reduction based on code coverage information.

We believe that the two selected tools are good representatives of the two approaches and being the most widely used ones, many would benefit from our results. A big initial question was, however, if we could use the tools as the "ground truth" in the comparison. Since most of the applications of code coverage operate on the *source code*, the source code instrumentation tool Clover was the candidate for this role. Thus, we performed a manual verification of the code coverage results provided by this tool by randomly selecting the outputs for investigation while maintaining a good overall functional coverage of the subject systems. We interpreted the results in terms of the actual test executions and program behavior on the level of source code. During this verification we did not find any issues, which made it possible to use this tool as a ground truth for source code coverage results.

To perform the actual comparison of the tools, various technical modifications had to be done on the tools and the measurement environment; for instance, to be able to perform per-test case measurements and calculate not only overall coverage ratios. This enabled a more detailed investigation of the possible causes for the difference.

Our results indicate that the differences between the coverage measurements can vary in a large range, and that it is difficult to predict in what situations will be the risk of measurement inaccuracy higher for a particular application. In summary, we make the following contributions:

1. The majority of earlier work in the topic dealt with lower level analyses such as statements and branches. Instead, we performed the experiments on the granularity of Java methods in real size Java systems with realistic test suites. We found that – contrary to our preliminary expectations – even at this level there might be significant differences between bytecode instrumentation and source code instrumentation approaches. Method level granularity is often the viable solution due to the large system size. Furthermore, if we can demonstrate the weaknesses of the tools at this level, they are expected to be present at the lower levels of granularity as well.
2. We found that the overall coverage differences between the tools can vary in both directions, and in seven out of the eight subject programs they are at most

[1] http://eclemma.org/jacoco/
[2] https://www.atlassian.com/software/clover/

1.5%. However, for the last program we measured an extremely large difference of 40% (this was then attributed to the different handling of generated code).
3. We looked at more detailed differences as well with respect to individual test cases and program elements. In many applications of code coverage (in debugging, for instance) subtle differences at this level may lead to serious confusion. We measured differences of up to 14% between the individual test cases, and differences of over 20% between the methods. In a different analysis of the results, we found that a substantial portion of the methods in the subjects were affected by this inaccuracy (up to 30% of the methods in one of the subject programs).
4. We systematically investigated the reasons for the differences and found that some of them were tool specific, while the others would be attributed to the instrumentation approach. This list of reasons may be used as a guideline for the users of coverage tools on how to avoid or workaround the issues when bytecode instrumentation based approach is used.
5. We also measured the effect of the differences on the application of code coverage to test prioritization. We found that the prioritized lists produced by the tools differed significantly (with correlations below 0.5), which means that the impact of the inaccuracies might be significant. We think that this low correlation is a great risk: in other words, it is not possible to predict the potential amplification of a given coverage inaccuracy in a particular application. This also affects any related research which is based on bytecode instrumentation coverage measurement to a large extent.

The paper is organized as follows. Section 2 gives the background of code coverage and its usability in different applications. We also list the risks of coverage measurement for Java and the relation to similar works in this section. We state our research aims in more detail in Section 3. Section 4 describes the basic setup for the experiments, the tools, and the subject systems, while Section 5 presents the results of the empirical study. This section is organized according to our research agenda: first, we concentrate on the quantitative and qualitative differences, and then, we investigate the effect on the test case prioritization application. In Section 6, we summarize our findings and provide a more general discussion before concluding in Section 7.

2 Background

The term *code coverage* in software testing denotes the amount of program code which is exercised during the execution of a set of test cases on the system under test. This indicator may simply be used as an overall *coverage percentage*, a proxy for test completeness, but typically more detailed data is also available about individual program elements or test cases. Code coverage measurement is the basis of several software testing and quality assurance practices including white-box testing (Ostrand, 2002), test suite reduction (Rothermel et al, 2002), or fault localization (Harrold et al, 1998).

2.1 Different Types and Levels of Code Coverage

Code coverage *criteria* are often used as goals to be achieved in white-box testing: test cases are to be designed until the required coverage level has been reached according to the selected criteria. However, many possible ways exist to define these criteria. They include various granularity levels of the analysis (such as component, method, or statement) and different types of "exercised parts of program code" (for instance, individual instructions, blocks, control paths, data paths, etc). The term *code coverage* without further specification usually refers to statement level analysis and denotes *statement coverage*. Statement coverage shows which instructions of the program are executed during the tests and which are not touched. Even at this level, there may be differences in what constitutes an instruction, which complicates the uniform interpretation of the results. In Java, for instance, a single source code statement is implemented with a sequence of bytecode instructions, and the mapping between these two levels is not always straightforward due to various reasons such as compiler optimization.

Another common coverage criterion is *decision coverage*, where the question is whether both outcomes of a decision (such as an *if* statement) are tested, or if a loop is tested with entering and skipping the body. Since this level of analysis deals with not only individual instructions but control flow as well, coverage measurement at this level imposes more issues. For instance, Li *et al.* (Li et al, 2013) showed that decision coverage at statement level for the Java language is prone to differences between the source code and the bytecode measurements; they found that practically the two results were hardly comparable. The main reasons for this besides the different optimizations were the actual shortcuts built in the implementation of the logical expressions: what seems to be a single logical expression in the source code can be a very complex control structure in the bytecode.

These examples show that even within a specific language difficulties might occur in defining and interpreting code coverage criteria. This may be even more emphasized at more sophisticated levels such as control path or data-flow based coverages (Ntafos, 1988).

Coarser granularity level coverage criteria (such as methods, classes, or components) are also common, for instance, when the system size and complexity do not enable a fine-grained analysis. Also, often it is more useful to start the coverage analysis in a "top-down" fashion by starting from the components that are not executed at all, extend the tests to cover that component at least once, and then continue the analysis with lower levels. In particular, *procedure level coverage* is a good compromise between analysis precision and the ability to handle big systems.

In our research, we primarily deal with this granularity, that is, we treat procedures (Java methods in particular) as atomic code elements that can be covered. At this level "covered" means that the method has been executed at least once during the tests but we do not care about what instructions, paths, or data have been exercised in particular. Contrary to what would be expected, this granularity level also involves difficulties in the interpretation of code coverage, which was the main motivation for our research. In particular, we found significant differences between different code coverage measurement tools for Java configured for method level analysis (Tengeri et al, 2016).

2.2 Applications of Code Coverage and Risks

Uncertainties in code coverage measurement may impose various risks. Here, we overview the notable applications of code coverage measurement and how they may be impacted by the uncertainties.

The most important application of code coverage measurement is white-box testing (often referred to as *structure-based testing*). It is a dynamic test design technique relying on code coverage to systematically verify the amount of tests needed to achieve a completeness goal, a coverage criterion. This goal is sometimes expected to be a complete (that is, 100%) coverage, however in practice, this high level is rarely attainable due to various reasons. As white-box testing directly uses coverage data, it is obvious that inaccuracies in the coverage results directly influence the testing activity. On the one hand, a small difference of one or two percentages in the overall coverage value is usually irrelevant if that value is used to assess the completeness. On the other hand, an item inaccurately reported to be covered provides false confidence in the code during a detailed evaluation, and it may result in unnecessary testing costs if an item is falsely reported to be uncovered.

Other applications of code coverage measurement include general software quality assessment[3], automatic test case generation (Rayadurgam and Heimdahl, 2001; Fraser and Arcuri, 2011), code coverage-based fault localization (Jones and Harrold, 2005; Yoo et al, 2013), test selection and prioritization (Offutt et al, 1995; Graves et al, 2001; Vidács et al, 2014), mutation testing (Usaola and Mateo, 2010; Jia and Harman, 2011), and in general, program and test comprehension with traceability analysis (Perez and Abreu, 2014). As in the case of white-box testing, the inaccuracy of code coverage measurement may affect these activities in different ways.

Certain applications do not suffer that much if the coverage data is not precise. This includes overall quality assessment, where the coverage ratio is typically used as part of a more complex set of metrics for software assessment. Here, a difference of a few percentages usually does not affect the overall score. Program comprehension (and general project traceability) is supported by knowing which program code is executed by which test case. Depending on the usage scenario of this information, inaccurate results may lead to either false decisions or simply an increased effort to interpret the data.

The other mentioned applications have high significance in academic research, and the accuracy and validity of the published results may be affected by the issues with the code coverage data. In coverage-driven test case generation, for instance, the generation engine can be confused by an imprecise coverage tool because a falsely reported non-coverage will keep the generation algorithm trying to generate test cases for the program element.

As another example, in code coverage-based fault localization the program elements are ranked according to how suspicious they are to contain the fault based on test case coverage and pass/fail status. Wrong coverage data may influence the fault localization process because if the faulty element is erroneously reported as not covered by a failing test case, the suspicion will move to other (possibly non-defective) program elements.

[3] http://www.sonarqube.org/

2.3 Code Coverage Measurement for Java

This paper deals with the applications of code coverage measurement for Java programs. Java itself is a popular language, and due to its language and runtime design, it can be more easily handled as the subject of code coverage measurement than other languages directly compiled for native code (like C++).

In addition, the increased demand for code coverage measurement in agile projects – where continuous integration requires the constant monitoring of the code quality and regression testing – has led to the appearance of a large set of tools for this purpose, many of which are free of charge and open source. However, it seems that the working principles, benefits, drawbacks, and any associated risks with these tools are not well understood by practitioners and researchers yet.

In Java, two conceptually different approaches are used for coverage measurement. In both approaches, the system under test and/or the runtime engine is *instrumented*, meaning that "measurement probes" are placed within the system at specific points, which enables the collection of runtime data but do not alter the behavior of the system. The first approach is to instrument the *source code*, which means that the original code is modified by inserting the probes, then this version is built and executed during testing. The second method is to instrument the compiled version of the system, *i.e.,* the *bytecode*. Here, two further approaches exist. First, the probes may be inserted right after the build, which effectively produces modified versions of the bytecode files. Second, the instrumentation may take place during runtime upon loading a class for execution. In the following, we will refer to these two approaches as *offline* and *online* bytecode instrumentation, respectively. Some example tools for the three approaches are Clover[2] (source code), Cobertura[4] (offline bytecode) and JaCoCo[1] (online bytecode).

Different possible features are available in tools employing these approaches, and they also have various benefits and drawbacks. In Table 1, we overview the most important differences. Of course, many of these aspects depend on the application context; here, we list our subjective assessment. One benefit of bytecode instrumentation is that it does not require the source code, thus it can be used *e.g.,* on third party code as well. On the other hand, it is dependent on the bytecode version and the Java VM, thus it is not as universal as source code instrumentation. In turn, implementing bytecode instrumentation is usually easier than inserting proper and syntactically correct measurement probe elements in the source code. Source code instrumentation requires a separate build for the instrumented sources, while bytecode instrumentation uses the compiled class files. However, the latter requires the modification of the VM in the online version. Source code instrumentation also allows full control over what is instrumented, while bytecode instrumentation is usually class-based (whole classes are instrumented at once). Online bytecode instrumentation will not affect compile time, but its runtime overhead includes not only the extra code execution time, but (usually a one-time per class load) instrumentation costs too. Finally, the bytecode based results are sometimes difficult to be tracked back to source code, while source code instrumentation results are directly assigned to the parts of the source code (Yang et al, 2009; Lingampally et al, 2007).

[4] http://cobertura.github.io/cobertura/

Table 1: Code Coverage Approaches for Java

Property	Source code	Offline bytecode	Online bytecode
Source code	Needed	Not needed	Not needed
Special runtime	Not needed	Not needed	Needed
Bytecode and VM	Not dependent	Dependent	Dependent
Filtering control	Complete	Partial	Partial
Separate build	Yes	No	No
Results in source	Yes	Partially	Partially
Compile time	Impacted	Impacted	Not impacted
Runtime	Impacted	Impacted	Highly impacted
Implementation	Difficult	Easy	Easy

These numerous benefits of bytecode instrumentation (*e.g.,* easier implementation, no need for source code and separate build) are so attractive that tools employing this technique are far more popular than source code instrumentation-based tools (Yang et al, 2009). Furthermore, most users do not take the trouble to investigate the drawbacks of this approach and the potential impact on their task at hand. Interestingly, scientific literature is also very poor in this respect, namely systematically investigating the negative effects of bytecode instrumentation on the presentation of results in source code (see Section 2.4).

The important benefits of source code instrumentation might overweight bytecode instrumentation in some situations, which are visible from Table 1. The most important benefit is that in the situations when the results are to be investigated on the source code level (in most of the cases!), mapping needs to be done from the computations made on the bytecode level. Due to the fact that perfect one-to-one mapping is generally not possible, this might impose various risks.

2.4 Related Work

There is a large body of literature on comparing various software analysis tools, for instance, code smell detection (Fontana et al, 2011), static analysis (Emanuelsson and Nilsson, 2008), test automation (Raulamo-Jurvanen, 2017), just to name a few.

Most of the works that compare bytecode and source code instrumentation techniques focus on the usability, operability, and the features of certain tools, *e.g.,* (Yang et al, 2009; Lingampally et al, 2007), but the accuracy of the results they provide is less often investigated – despite the importance and the possible risks overviewed above.

Li and Offutt (Li et al, 2013) examined the difference between source code and bytecode instrumentation in relation to branches and statements. Their conclusion was that due to several differences between the two methods, source code instrumentation is more appropriate for branch coverage computation. We verify the differences on coarser granularity (on method level), and how these differences impact the results of the further applications of the coverage.

Kajo-Mece and Tartari (Kajo-Mece and Tartari, 2012) evaluated two coverage tools (source code and bytecode instrumentation based ones) on small programs

and concluded that the source code based one was more reliable for the use in determining the quality of their tests. We also used Java source code and bytecode instrumentation tools in our experiments, but on a much bigger data set in a more comprehensive analysis.

Alemerien and Magel (Alemerien and Magel, 2014) conducted an experiment in order to investigate how the results of code coverage tools are consistent in terms of line, statement, branch, and method coverage. They compared the tools using the overall coverage as the base metric. Their findings show that branch and method coverage metrics are significantly different, but statement and line coverage metrics are only slightly different. They also found that program size significantly affected the effectiveness of code coverage tools with large programs. They did not evaluate the impact of the difference on the applications of code coverage. We investigated only method level coverages, but we did not only use an overall coverage but analyzed detailed coverage information as well. Namely, we computed coverage information for each test case and method individually, and analyzed the differences using this data.

Kessis *et al.* (Kessis et al, 2005) presented a paper in which they investigated the usability of coverage analysis from the practical point of view. They conducted an empirical study on a large Java middleware application, and found that although some of the coverage measurement tools were not mature enough to handle large scale programs properly, using the adequate measurement policies would radically decrease the cost of coverage analysis, and together with different test techniques it could ensure a better software quality. Although we are not examining the coverage tools themselves, we rely on their produced results and cannot exclude all of their individual features from the experiments.

This work is a followup of our previous work in the topic (Tengeri et al, 2016), in which we investigated bytecode and source code coverage measurement on the same Java systems we used in this work. We found that there were many deviations in the raw coverage results due to the various technical and conceptual differences of the instrumentation methods, but we did not investigate the reasons in detail and how these differences could influence the applications where coverage data was used.

There are many code coverage measurement tools for Java (*e.g.,* Semantic Designs Test Coverage[5], Cobertura[4], EMMA[6], FERRARI (Binder et al, 2007), and others). In Section 4.2, we discuss how we selected the tools for our measurements.

In this work, we consider test suite reduction and test prioritization, as the application of code coverage. Yoo and Harman conducted a survey (Yoo and Harman, 2012) on different test suite reduction and prioritization methods among which coverage-based methods can also be found. The most basic coverage-based prioritization methods, which were studied by Rothermel *et al.* (Rothermel et al, 2001), are the *stmt-total* and *stmt-addtl* coverages. In our experiments, we applied these concepts on the method level, and referred to them as *general*, *additional*, and *additional with resets*. One of the test prioritization algorithms we used in our experiments was optimized for fault localization, which is based on the previous work by Vidács *et al.* (Vidács et al, 2014). Fault localization aware test

[5] http://www.semdesigns.com/Products/TestCoverage
[6] http://emma.sourceforge.net/

suite reduction turned out to produce different results than fault detection aware reduction, which optimizes for code coverage.

3 Research Goals

Following earlier research on the drawbacks of bytecode instrumentation for Java code coverage on lower granularity levels (Li et al, 2013), and addressing challenges listed in the previous sections, the aim of our research is the following. We investigate in quantitative and qualitative terms in what situations and to what extent Java method-level code coverage based on bytecode instrumentation is different than coverage based on source code? We investigate the differences between the actual coverage information on a detailed level and determine the root causes of these differences after a manual investigation of the source code of the affected methods. In addition, we evaluate the impact of the inaccuracies on an application of code coverage measurement, namely test prioritization/reduction. We assume that a certain degree of the differences in the coverages may be reflected in a different degree of inaccuracies of the application.

To achieve our goal, we consider several candidate tools and then conduct an empirical study involving two representative tools, one with source code instrumentation and one with online bytecode instrumentation. We then measure the code coverage results on a set of benchmark programs and elaborate on the possible causes and impacts.

More precisely, our research questions are:

RQ1 How big is the difference between the code coverage obtained by an unmodified bytecode-instrumentation based tool and a source code-instrumentation based tool on the benchmark programs?

RQ2 What are the typical causes for the differences?

RQ3 Can we eliminate tool-specific differences, and if we can, how big the difference, – which can be possibly attributed to the differences in the fundamental approach, that is, bytecode vs. source code instrumentation – remains?

RQ4 How big is the impact of code coverage inaccuracies on the application in test prioritization/test suite reduction?

In this paper, we calculate and analyze coverage results on the *method level*. More precisely, the basic element of a coverage information is whether a specific Java method is invoked by the tests or not, regardless of what statements or branches are taken in that method. At first, this might seem too coarse a granularity, but we believe that the results will be actionable due to the following.

In many realistic scenarios, coverage analysis is done hierarchically starting from the higher level code components like classes and methods. If the coverage result is wrong at this level, it will be wrong at the lower levels too. Also, in the case of different applications, unreliable results at the method level will probably mean similar (if not worse) results at the level of statements or branches as well. Previous works have shown that notable differences exist between the detailed results of bytecode and source code coverage measurements at statement and branch level (Li et al, 2013), and that at method and branch level the overall coverage values show significant differences (Alemerien and Magel, 2014). So, this leaves the question whether there are notable differences in method level coverage results as well open.

4 Description of the Experiment and Initial Measurements

To answer the research questions set forth in the previous section, we conducted an empirical study on eight open source systems (introduced in Section 4.1) with code coverage tools for Java employing both instrumentation approaches. Initially, we involved more tools, but as Section 4.2 discusses, we continued the measurements with two representative tools. In Section 4.3, we overview the measurement process and discuss some technical adjustments we performed on the tools and subjects.

Apart from the coverage measurement tools, our measurement framework consisted of some additional utility tools. The main tool we relied on was the SoDA[7] framework (Tengeri et al, 2014). For the representation of the coverage data in SoDA, the data generated in different forms by the coverage tools were converted into the common SoDA representation, the coverage matrix. Later, this representation was used to perform the additional analyses. This framework also contains tools to calculate statistics, produce graphical results, etc. SoDA includes the implementation of the test case prioritization and the test suite reduction algorithms, which we used for our Research Question 4. Apart from this, only general helping shell scripts and spreadsheet editors have been used.

4.1 Benchmark Programs

For setting up our set of benchmark programs, we followed these criteria. As we wanted to compare bytecode and source code instrumentation, the source code had to be available. Hence, we used open source projects, which also enables the replication of our experiments. We used the Maven infrastructure in which the code coverage measurement tools easily integrate, so the projects needed to be compilable with this framework. Finally, it was important that the subject programs had a usable set of test cases of realistic size, which are based on the JUnit framework[8] (preferably version 4). The reason for the last restriction was that the use of this framework was the most straightforward for measuring per-test case method coverage.

We searched for candidate projects on GitHub[9] preferring those that had been used in the experiments of previous works. We ended up with eight subject programs which belonged to different domains and were non-trivial in size (see Table 2). The proportion of the tests in these systems as well as their overall coverage is varying, which makes our benchmark even more diversified. Columns "All tests" and "Ex. tests" show the size of the test suites and the number of testcases that were excluded (we discuss the technical modifications that we performed in Section 4.3 in detail).

4.2 Selection of Coverage Tools

Our goal in this research was to compare the code coverage results produced by tools employing the two instrumentation approaches. Hence, we wanted to make

[7] http://soda.sed.hu
[8] http://junit.org/
[9] https://github.com/

Table 2: Subject programs. Metrics were calculated from the source code (generated code was excluded).

Program	version	LOC	Methods	All tests	Ex. tests	Domain
checkstyle	6.11.1	114K	2 655	1 589	104	static analysis
commons-lang	#00fafe77	69K	2 796	3 683	358	java library
commons-math	#2aa4681c	177K	7 167	5 842	902	java library
joda-time	2.9	85K	3 898	4 177	162	java library
mapdb	1.0.8	53K	1 608	1 786	68	database
netty	4.0.29	140K	8 230	4 066	247	networking
orientdb	2.0.10	229K	13 118	950	153	database
oryx	1.1.0	31K	1 562	208	0	mach. learning

sure that the tools selected for the analysis are good representatives of the instrumentation methods and that our results are less sensitive to tool specificities. The comprehensive list of tools we initially found as candidates for our experimentation is presented in Table 3.

We ended up with this initial list after reading the related works (some of them are mentioned in Section 2), and searching for code coverage tools on the internet. We learned that the area of code coverage maeasurement for Java was most intensively investigated in the early 2000's. At that time there were several different tools available, but the support for most of these tools has long ended. There were tools referred by related literature and some webpages which we could not even found, so we did not include them in the table.

In the next step, this list was reduced to five tools, which are shown in the first five rows of the table and marked boldface. For making this shortlist, we established the following criteria. First, we aimed at actively developed and maintained tools that were popular among users. We measured the popularity of the tool candidates by reviewing technical papers, open source projects, and utilizing our experiences from previous projects. The tools had to handle older and current Java versions including new language constructs (support for at least Java 1.7 but preferably 1.8 was needed). Finally, we wanted the tool to easily integrate into the Maven build system[10], as today this seems to be a popular build system used in many open source projects. In addition, the ability of smooth integration reduces the chances of unwanted changes in the behavior of the system and in the tests used in the experiments. Finally, among the more technical requirements for the tools was the ability to perform coverage measurement on a per test basis.

We ended up with five tools to be used for our measurements meeting these criteria. Three of them use bytecode instrumentation, and two are based on source code instrumentation. In Section 2.3, we discussed three fundamental code coverage calculation approaches for Java. However, in the case of bytecode instrumentation, there are no fundamental differences in how and which program elements are instrumented, only the "timing" of the instrumentation is different. Hence, we include source code instrumentation as one category, but we do not consider both types of bytecode instrumentation separately in the following.

In the following, we discuss briefly the selected tools.

[10] https://maven.apache.org/

Table 3: Tools for Java code coverage measurement

Tool	Approach	Supported Java/JRE version	Active	Licence
Clover	source	1.3+	present	commercial/free
Cobertura	bytecode	1.5–1.7	2015	free
JaCoCo	bytecode	1.5+	present	free
Jcov	bytecode	1.0+	present	free
SD Test Coverage tools	source	1.1+	present	commercial
Agitar(One)	bytecode	1.6+	present	commercial
CodeCover	source	1.5–1.7	2014	free
Coverlipse	bytecode	1.5	2009	free
EclEmma (JaCoCo-based)	bytecode	1.5+	present	free
Ecobertura (Cobertura-based)	bytecode	1.5–1.7	2010	free
Emma	bytecode	1.5	2005	free
Gretel (by Univ. of Oregon)	bytecode	1.3+	2003	free
GroboUtils	bytecode	1.4	2004	free
Hansel (Gretel-based)	bytecode	1.5	2006	free
InsECTJ	bytecode	1.5	2003	free
Jcover	both	1.2 – 1.4	2009	commercial
Jtest (by Parasoft)	bytecode	?	present	commercial
JVMDI	bytecode	1.4+	2002	free
Koalog	bytecode	?	2004	commercial
NetBeans Code Coverage Plugin	bytecode	1.6	2010	free
NoUnit	bytecode	1.5	2003	free
PurifyPlus	bytecode	1.5+	present	commercial
Quilt	bytecode	1.4	2003	free
TestWorks	bytecode	1.2+	present	commercial

4.2.1 Source code-based instrumentation tools

As mentioned earlier, there are comparably much less coverage tools employing this method. Essentially, we could find only two active tools that are mature enough and meet our other criteria to serve the purposes of our experiment. The tools selected for the source code instrumentation approach were Clover by Attlassian[2] (version 4.0.6), and Test Coverage[11] by Semantic Designs (version 1.1.32).

Clover is the product of Atlassian, and it was a commerical product for a long time but, recently, it became open source. It handles Java 8 constructs, easily integrates with the Maven build system, and can measure per-test coverage.

Test Coverage is a commercial coverage tool from Semantic Designs. Native, it works on Windows, handles most Java 8 code and can be integrated into the Maven build process as an external tool. Per-test coverage measurement is not feasible by this tool, because it could only be solved by the individual execution of test cases.

We performed some initial experiments to compare these two tools. The details and results of this investigation can be found in Appendix A. Results showed that there were only minimal differences in the outputs produced by the two tools, their accuracy is almost the same.

Finally, we chose Clover to be used in our detailed bytecode-source code measurements because it has better Maven and per-test coverage measurement sup-

[11] http://www.semdesigns.com/Products/TestCoverage/JavaTestCoverage.html

port, which made it easier to integrate it into our experiments. Also, Test Coverage had difficulties in handling some parts of our code base, which would have required their exclusion from the experimentaion.

To be able to use the source code instrumentation results as the baseline in our experiments, we did a manual verification of the results of Clover by performing a selective manual instrumentation. A subset of the methods was selected for each of our subject systems, up to 300 methods per system. Then, we manually instrumented these methods and ran the test suite. We interpreted the results in terms of actual test executions and program behavior on the level of source code. When the results were checked, we found no deviations between the covered methods reported by the manual instrumentation and by Clover. Thus, we treat Clover as a "ground truth" for source code coverage measurement from this point onward.

4.2.2 Bytecode-based instrumentation tools

We found three candidate tools in this category that met the mentioned criteria: JaCoCo[1] (version 0.7.5.201505241946), Cobertura[4] (version 2.1.1) and JCov[12] (version c7a7c279c3a6). Contrary to the two previous ones, all three tools in this category are open source. We performed similar experiments to compare the results of these three tools, and investigated their differences. These experiments and the results are detailed in Appendix B. Our conclusion was that the main cause of the differences was mostly due to the slightly different handling of compiler generated methods in the bytecode by the three tools (such as for nested classes). Since the quantitative differences were at most 4% and they were concerned mostly generated methods, which are less important for code coverage analysis, we concluded that one representative tool of the three should be sufficient for further experiments.

We ended up using JaCoCo[1] for the bytecode instrumentation approach thanks to its popularity and slightly higher visibility and easier integration for use in our experiments than the other two. This is a free Java code coverage library developed by the EclEmma team, which can easily be integrated into a Maven-based build system. JaCoCo has plug-ins for most of the popular IDEs *i.e.,* Eclipse[13], NetBeans[14], IntelliJ[15], for CI- and build systems *e.g.,* Jenkins[16], Maven[10], Gradle[17] and also for quality assessment tools *e.g.,* SonarQube[3]. These plug-ins have about 20k installations/downloads per month in total. In addition, several popular projects, *e.g.,* Eclipse Collections[18], Spring Framework[19], and Checkstyle[20] are utilizing JaCoCo actively. JaCoCo has up-to-date releases and an active community.

[12] https://wiki.openjdk.java.net/display/CodeTools/jcov/
[13] https://www.eclipse.org/
[14] https://netbeans.org/
[15] https://www.jetbrains.com/idea/
[16] https://jenkins.io/
[17] https://gradle.org/
[18] https://www.eclipse.org/collections/
[19] http://projects.spring.io/spring-framework/
[20] http://checkstyle.sourceforge.net/

4.3 Measurement Process

In order to be able to compare the code coverage results and investigate the differences in detail, we had to calculate the coverages with the different settings and variations of the tools. In particular, we wanted the data from the two tools to be comparable to each other, and we wanted to eliminate tool-specific differences. Hence, we essentially calculated different sets of coverage data, which we will denote by JaCoCo^{glob}, JaCoCo, JaCoCo^{sync}, Clover^{glob}, Clover, and Clover^{sync}, with explanations following shortly.

The experiment itself was conducted as follows. First, we modified the build and test systems of each subject program to integrate the necessary tasks for collecting the coverage data using the two coverage tools. This task included a small modification to ignore the test failures of a module that would normally prevent the compilation of the dependent modules and the whole project. This was necessary when some tests of the project failed on the measured version, and in a few cases when the instrumentation itself caused some tests to fail. Furthermore, to avoid any bias induced by "random" tests, we executed each test case three times and excluded from further analysis the ones that did not yield the same coverage consistently every time. Eventually, we managed to arrive at a set of filtered test cases that was common for both tools. Column "Excluded tests" of Table 2 gives the number of excluded tests for each subject.

Since we planned a detailed study on the differences between the tools, we wanted to make sure that we could gather *per-test case* and *per-method* coverage results from the tools as well (*i.e.,* which test cases covered each method and the opposite). Clover could be easily integrated in the Maven build process and there were no problems in producing the per-test case coverage information we needed. JaCoCo measurements could also be integrated into a Maven-based build system, but originally it could not perform coverage measurements for individual test cases. So, to be able to gather the per-test case coverage information, we implemented a special listener at first. Then, we configured the test execution environment of each program to communicate with this listener. As a result, we were able to detect the start and the end of the execution of a test case (tools and examples are available at[21]). From the per-test case coverage, we then produced a *coverage matrix* for each program, which is essentially a binary matrix with test cases in its rows, methods in the columns and 1s in the cells if the given method is reached when executing the given test case. From this matrix, we could easily compute different kinds of coverage statistics including per-test case and per-method coverage.

Due to the mentioned extension of the JaCoCo measurements, we essentially started with two different kinds of JaCoCo results: the original one without test case separation, which we will denote by JaCoCo^{glob}, and the one with the special listener denoted by JaCoCo. Theoretically, there should be no differences between the two types of measurements, but since we noticed some, we investigated their amount and causes. Table 4 shows the two overall coverage values for each program in columns two and three, with the differences shown in the fourth column. It can be observed that JaCoCo results are always somewhat smaller than the JaCoCo^{glob} measurements. The difference is caused by executing and covering some general utility functions (such as the preparation of the test execution) in the unseparated

[21] https://github.com/sed-szeged/soda-jacoco-maven-plugin

version during the overall testing, but these cannot be associated to any of the test cases. Since these methods have no covering test cases assigned, when we summarize the coverage of all test cases, the methods remain uncovered. Note, that Clover does not suffer from this issue as it originally produces per-test case results.

Table 4: Effect of technical setup on overall coverage values.

Program	JaCoCo^{glob}	JaCoCo	difference	Clover^{glob}	Clover	difference
checkstyle	53.85%	53.77%	-0.08%	93.82%	93.82%	0.00%
commons-lang	93.29%	92.92%	-0.37%	93.28%	93.28%	0.00%
commons-math	85.59%	84.92%	-0.67%	84.65%	84.65%	0.00%
joda-time	91.36%	89.52%	-1.84%	89.94%	89.94%	0.00%
mapdb	79.65%	74.64%	-5.01%	76.06%	76.06%	0.00%
netty	47.41%	40.92%	-6.49%	46.66%	40.18%	-6.48%
orientdb	38.40%	27.01%	-11.39%	39.84%	28.01%	-11.83%
oryx	29.62%	29.51%	-0.11%	27.51%	28.75%	+1.24%

Another technicality with the Clover tool had to be addressed before moving to the experiments themselves. Namely, for handling multiple modules in projects we had two choices with this tool: either to integrate the measurement on a global level for the whole project, or to integrate it individually in the separate sub-modules (this configuration can be performed in the Maven build system). Since JaCoCo follows the second approach, we decided to configure Clover individually for the sub-modules as well. These measurements will be denoted simply by Clover, and will be used subsequently.

Three of the eight subject systems (netty, orientdb and oryx) include more than one sub-module, so this decision affected the measurement in these systems. To assess how such a handling of sub-modules differs from the other approach, we performed global measurements as well (denoted by Clover^{glob}), whose results can be seen in the last three columns of Table 4. Clover^{glob} measurements typically include a smaller number of covered elements than Clover, but it may happen that the coverage itself is bigger, which is due to the different number of overall recognized methods. We will elaborate on the differences caused by sub-module handling in detail in Section 5.2.

5 Results

The experiment results presented in this section follow our RQs from Section 3. As discussed in Sections 2.3 and 4.2, we treat source code-based instrumentation as more suitable for source code applications and Clover results as the ground truth, hence JaCoCo results will be compared to Clover, serving as the reference.

5.1 Differences in Unmodified Coverage Values – **RQ1**

Our first research question dealt with the amount of differences we can observe in the overall coverage values calculated by the two tools. In this phase, we wanted

to compare the raw, unmodified data from the tools "off the shelf," because this could reflect the situations users would experience in reality as well. However, as explained in Section 4.3, we needed to perform a modification of the tool execution environment to enable per-test case measurements, which caused slight changes in the overall coverages. In this section, we rely on this modified set of measurements, which is denoted simply by JaCoCo and Clover.

5.1.1 Total coverage

First, we compared the overall method-level coverage values obtained for our subject programs, which are shown in Table 5. JaCoCo and Clover results are shown for each program, along with the difference of the coverage percentages. Coverage ratios are given in percentages of the number of covered methods from all methods recognized by the corresponding tool.

Table 5: Overall coverage values for the unmodified tools

Program	JaCoCo	Clover	difference
checkstyle	53.77%	93.82%	-40.05%
commons-lang	92.92%	93.28%	-0.36%
commons-math	84.92%	84.65%	+0.27%
joda-time	89.52%	89.94%	-0.42%
mapdb	74.64%	76.06%	-1.42%
netty	40.92%	40.18%	+0.74%
orientdb	27.01%	28.01%	-1.00%
oryx	29.51%	28.75%	+0.76%
average	**61.65%**	**66.84%**	**-5.19%**

Excluding the outlier program checkstyle, the differences between the tools range in a relatively small interval, from -1.42% to 0.76%. In the following sections, we seek for the reasons of the differences, and we will explain the outlier as well (in Section 5.2.4).

5.1.2 Per-test case coverage

While Table 5 presents the overall coverage values produced by the whole test suite, the coverage ratios attained by the individual test cases might show another range of specific differences. Table 6a contains statistics about the coverages for the individual test cases for JaCoCo, and Table 6b shows similar results for Clover. (Coverage is again the number of covered methods relative to all methods). This includes minimum, maximum, median, and average values with standard deviation. In Table 6c the difference in the average values between the two tools is shown (positive values denote bigger average coverage values for Clover). It can be observed that checkstyle reflects the high global difference of Clover and JaCoCo in the per-test case results too, although not as emphasized as in the global case. Interestingly, in the case of mapdb, netty, and oryx the average individual differences between Clover and JaCoCo have the opposite sign than the global differences.

Table 6: Per-test case coverages

(a) JaCoCo results

Program	min	max	med	avg	dev
checkstyle	0.00%	15.87%	4.11%	3.02%	2.36%
commons-lang	0.00%	3.10%	0.20%	0.61%	0.74%
commons-math	0.00%	4.34%	0.26%	0.47%	0.54%
joda-time	0.00%	8.84%	1.24%	1.62%	1.37%
mapdb	0.00%	20.86%	6.28%	7.67%	5.39%
netty	0.00%	3.43%	0.24%	0.32%	0.29%
orientdb	0.00%	9.40%	0.22%	0.58%	1.20%
oryx	0.00%	2.05%	0.33%	0.45%	0.40%

(b) Clover results

Program	min	max	med	avg	dev
checkstyle	0.00%	30.13%	6.21%	4.66%	3.65%
commons-lang	0.00%	2.93%	0.21%	0.64%	0.76%
commons-math	0.00%	4.34%	0.25%	0.47%	0.55%
joda-time	0.00%	9.93%	1.23%	1.64%	1.39%
mapdb	0.00%	22.08%	6.09%	7.19%	6.08%
netty	0.00%	3.69%	0.26%	0.37%	0.33%
orientdb	0.00%	9.88%	0.24%	0.62%	1.28%
oryx	0.00%	1.79%	0.38%	0.48%	0.40%

(c) JaCoCo to Clover average difference

Program	avg diff
checkstyle	+1.64%
commons-lang	+0.03%
commons-math	0.00%
joda-time	+0.02%
mapdb	-0.48%
netty	+0.05%
orientdb	+0.04%
oryx	+0.03%

Note, that it is not obvious how individual coverage differences imply global coverage difference and vice versa. It might happen that individual coverages differ greatly but the overall coverage is not changed. For example, one test case is enough for a method to be reported as covered, and if one instrumentation technique reports a hundred covering test cases while the other technique reports only one, the global coverage will not change only the individual ones. Similarly, low average individual differences might result in a high global difference; if many test cases have only one method which is reported differently, and these methods are uniquely covered by those test cases, the small individual differences will sum up in a high global coverage difference.

An even more detailed way to compare the per-test case coverages is investigating not only the overall coverage ratios but the whole *coverage vector* (it is the row vector of binary values from the coverage matrix for the corresponding test case). Figure 1 shows the analysis of the difference between the corresponding

coverage vectors produced by the two tools. The difference was computed as the Hamming distance normalized to the vector lengths. Note, that the two tools may recognize a different number of methods (more on this in the next section), so in these cases the vectors were padded with no-coverage marks for the missing methods. Then, the distribution of the obtained differences was calculated and shown as a histogram. The X axis of the graphs shows the ranges of differences (size ranges are 1%) and the Y axis the number of cases (relative to all cases) for the given difference range. As expected, a lot of small differences occurred. In particular, a significant portion of the vectors had 0 difference. On the other end, none of the programs had vectors with Hamming distance values larger than 20%. Hence, to ease readability, we omit the values that were 0 or larger than 20% from the diagrams and show the corresponding numbers instead in the top right corner of the graphs.

There are two interesting results here. The Hamming distances of mapdb have a different distribution than that of the other programs (the differences go up to 14% with this program), and not surprisingly, it is also reflected in the higher average per-test case differences. This shows that while differences would occur in either direction, in most of the cases the JaCoCo coverage turns out to be higher than the Clover result (in contrast to the others, where on average Clover reports higher coverage). The second interesting observation is that checkstyle behaves differently than the other programs: the high average per-test case difference measured for this program (1.64%) is not observable from the Hamming distances (more than 90% of the test cases show no difference and the others are below 1%). This seems to be inconsistent at the first sight. However, as we will explain it in Section 5.2.4, checkstyle shows a significant difference in the number of methods detected by the two instrumentation techniques. Thus, the average coverage values for JaCoCo and Clover used significantly different denominators, while during Hamming distance computation a common denominator was used and this caused the observed difference.

5.1.3 Per-method coverage

In the previous experiment, we investigated the coverage from the test case dimension. In the next one, we did the same from the method dimension. The distributions of the Hamming distances were calculated similarly to the per-test case analysis. The results in Figure 2 show a similar overall picture to the per test-case analysis. Therefore, we used the same method to exclude and emphasize the differences larger than 20% or equal to 0. The distribution of the distances and the average per-test case coverage values seem to be unrelated. However, checkstyle and mapdb behave differently than the other programs in this case, too. The high average per-test case difference measured for checkstyle is not observable from the Hamming distances, while the high distances in the case of mapdb result in a relatively high average difference.

In this case, we performed another, slightly different analysis. For each method, we recorded how many of the test cases cover that method according to the two tools. Then, we counted the number of the methods for which the number of the covering test cases was equal, and how many times one or the other tool reported this differently. This kind of analysis is useful because it helps to find out the

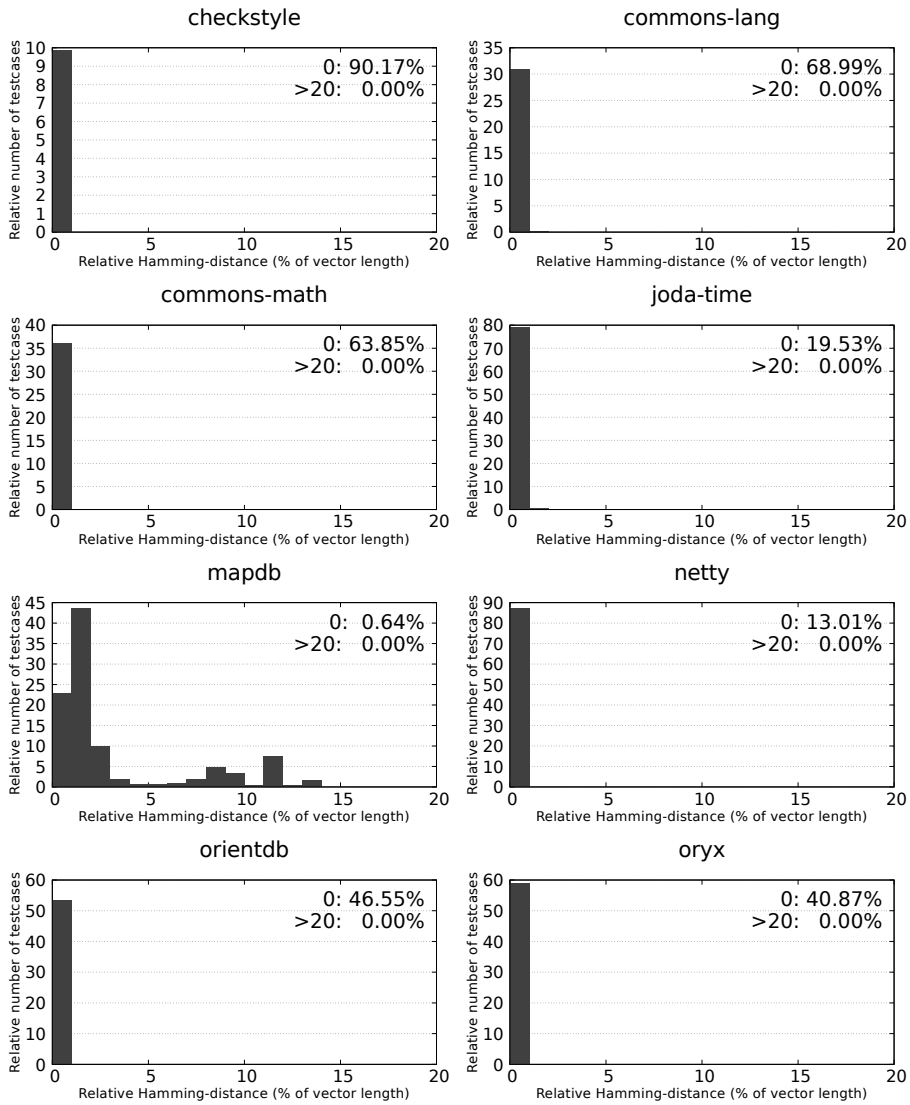


Fig. 1: Relative Hamming distances of test case vectors (JaCoCo vs. Clover)

number of situations when the methods are found falsely (not) covered, which may lead to confusion in certain applications.

When we compared the "number of covering test cases," we identified three kinds of differences. First, JaCoCo and Clover recognized different sets of methods, for which the reasons will be explained in Section 5.2.4. Second, for some of the methods recognized by both approaches, Clover reported at least one covering test case but JaCoCo did not, and vice versa. The third kind of difference is when both tools reported that a method was covered, but by a different number of test cases. Figure 3 shows the associated results. In particular, the percentage of the methods

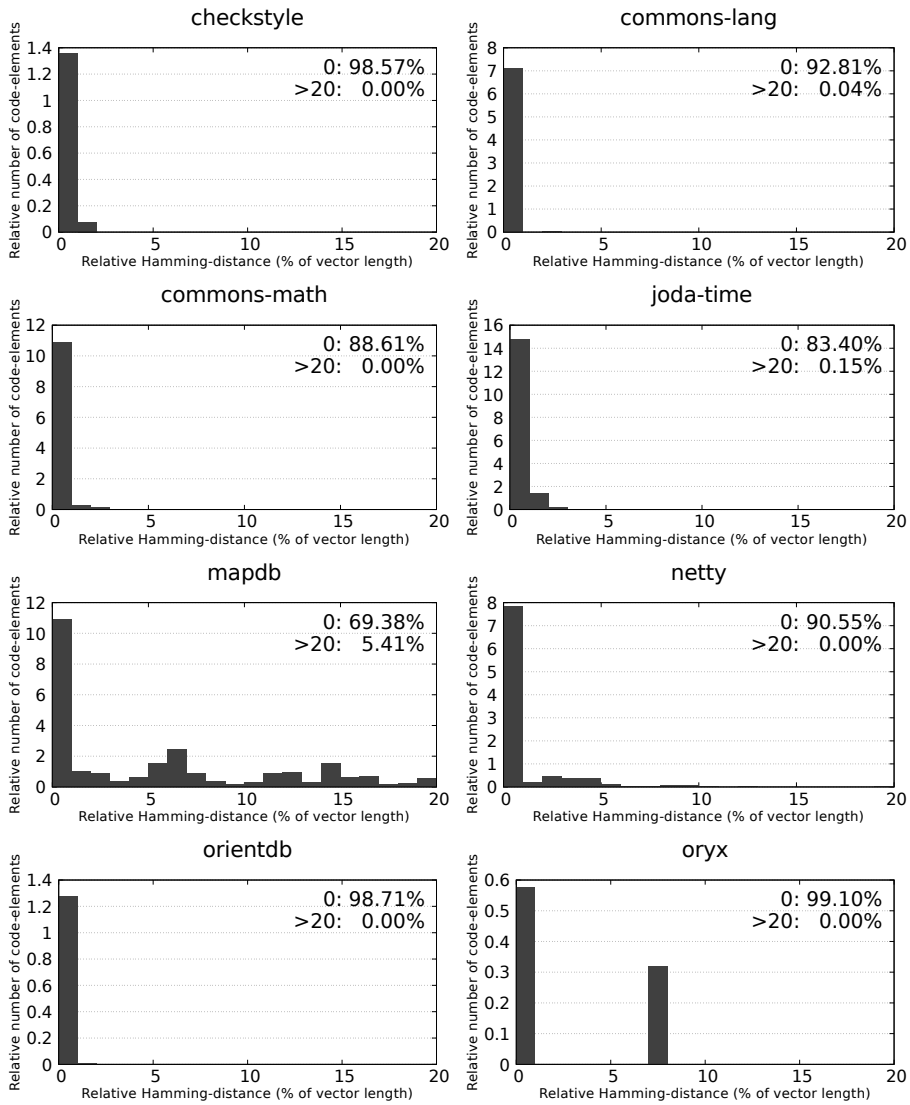
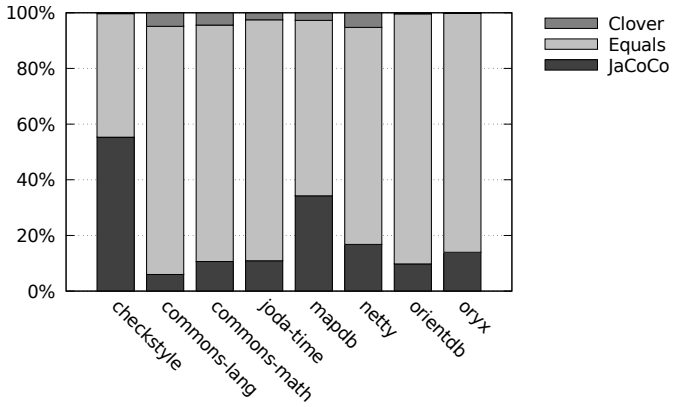


Fig. 2: Relative Hamming distances of code-element vectors (JaCoCo vs. Clover)

is shown for each program (with respect to the total number of methods recognized by any of the tools) for the following cases: there is no difference in the covering sets of test cases, and either Clover or JaCoCo reports more covering test cases. In the latter category, all three kinds of differences from above are counted together.

An ideal case would be if only *Equals* is present, which would mean that the two tools completely agree in the coverages. However, we can observe that the situation is quite different. First, many methods are not recognized by the Clover tool, which can be attributed to various reasons but mostly to the generated code. A notable outlier is checkstyle with 55% of such methods, the others are below 15%.



Clover: Clover reports more covering TCs; Equals: both tools report the same covering TCs; JaCoCo: JaCoCo reports more covering TCs.

Fig. 3: Summary of differences in the per-method coverage

Next, as can be seen in Table 7, there are only a few methods for which Clover and JaCoCo do not agree in the coverage fact (covered by at least one test case) while both recognize the method (*Czero* and *Jzero* columns). We investigated all these 220 methods manually to find out the reasons for the difference (see Section 5.2.4). The other two columns report on the cases when the number of covering test cases was not zero but different. Column *CltJ* means "Clover reports less than JaCoCo," while *JltC* is "JaCoCo reports less than Clover". A significant portion of the methods in the subjects were affected by the inaccuracy to some extent (nearly 30% for mapdb and over 11% for joda-time, for instance).

Table 7: Differences in per-method coverages of code elements of JaCoCo and Clover

Program	Czero	CltJ	JltC	Jzero
checkstyle	1	9	16	0
commons-lang	0	21	131	5
commons-math	19	297	239	7
joda-time	0	358	86	2
mapdb	7	450	25	2
netty	91	300	466	76
orientdb	1	104	32	5
oryx	4	8	1	0

As a summary answer to RQ1, the detailed per-test case and per-method measurements, when compared to the overall coverage ratios, may show quite different trends. In some cases, the overall ratios are reflected in the detailed data, but not necessarily: a high overall difference is often caused by a little difference on a detailed level, and the opposite. In other words, by observing a certain overall level of inaccuracies, we cannot predict the differences on the more detailed levels, and consequently, the effect on possible applications.

5.2 Causes of Differences – **RQ2**

In this section, we address the possible causes for the differences we observed and presented in the previous section. We used manual inspection, and carefully examined the differences between the coverage results reported by JaCoCo and Clover. Due to their large number, we could not look into each individual difference, instead we manually selected the typical cases making sure that each system and module was sufficiently covered by our investigation. We also made sure to investigate all of the most problematic cases shown in columns *Czero* and *Jzero* of Table 7. We involved the original and instrumented versions of the source code and the bytecode as well. In addition, we examined other artifacts like build configuration files to reveal additional factors that could be the cause of differences. The work was performed by three authors of the paper, first by dividing the difference cases equally and performing the inspection individually. Then, each result was cross-validated by at least one of the other authors to ensure consistency of the results. Altogether, we manually investigated several hundred individual methods and test cases one by one during this work. Finally, we were able to identify a set of common reasons, which we overview in the following.

5.2.1 Cross-submodule coverage

In the case of the projects consisting of multiple sub-modules, Clover and JaCoCo work differently. Clover first instruments the whole source, thus, it is able to report a cross-module coverage. On the other hand, JaCoCo concentrates on the tested module and does not instrument other modules when it is tested, thus it cannot report a cross-module coverage. Consider Figure 4 for illustration. Let the system have three sub-modules A, B, and C, which define their own dependencies and build processes including unit tests. In the example, modules A and C include test cases, while module B does not. The arrows on the figure indicate the possible calls from tests to non-test methods and between non-test methods. During the build (and test phase) of module C, module A is treated as an "external" dependency, which prevents JaCoCo from instrumenting and measuring the coverage of the methods of A (along the gray edges starting from module C). Thus, it only considers a method of module A covered if the method is invoked from the tests of module A (along the black edges). On the other hand, Clover aggregates the coverage among all modules, so if a method from A is used in a test in C (through some gray edges), Clover considers the method as covered. These different behaviors can lead to differences in the global coverage of the projects. Our subjects netty, orientdb, and oryx are examples of multiple module projects. The other five programs are single module projects.

Note, that although we investigated only Maven based projects in our experiments, we think that similar problems may occur in other build configuration systems as well.

5.2.2 Untested sub-modules

In the case of JaCoCo, if a module does not have any tests its methods will not be recognized. Consider again Figure 4, where module B does not have any tests, thus JaCoCo will not be executed for it (grayed in the figure). Consequently, the

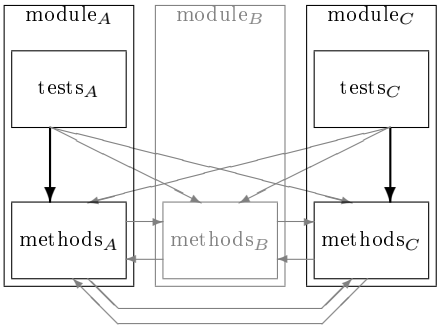


Fig. 4: Illustration of problems with sub-modules

methods of B will not be recognized and they will be missed from the set of all methods of the project. Clover, on the other hand, correctly determines the set of all methods across all sub-modules, and will include methods of modules A, B, and C.

5.2.3 Test case preparation and cleanup

Some test cases might need a preparation or cleanup, and this is common in some of the programs. Technically, this is usually implemented as *setup* and *teardown* methods (annotated by `@Before`, `@BeforeClass`, `@After` or `@AfterClass` in JUnit) associated to a test class or a set of test methods. These are executed before/after a set of test cases or before/after each test case that requires them. In the JaCoCo measurement architecture, these are counted as part of the test cases, *i.e.,* all the methods executed during these setup/teardown phases are reported as covered by the corresponding test cases. On the contrary, Clover does not treat setup and teardown as an integral part of the test case, and as a consequence, if a method is covered only during the setup or teardown phase of a test case, it will not be assigned to the test case.

5.2.4 Recognized method sets

In addition to the previous three cases, a further inaccuracy exists between the JaCoCo and Clover results regarding the method sets, which is due to the fact that the set of methods detected from the source code and the bytecode can be different. There are many reasons for this; some of them are the inherent problems of the measurement and some of them are tool specific. Table 8 introduces our measurements in this regard (also see Figure 3). The second column shows how many methods are recognized by both tools, and how many are recognized only by Clover or JaCoCo which are given in the third and fourth columns, respectively. The last column contains the sum of these three values, *i.e.,* the total number of methods recognized by Clover and JaCoCo together.

Observe that several methods are recognized only by Clover or JaCoCo. The second group is not really surprising because we expected in advance a relatively

Table 8: Number of all methods

Program	Both	Clover only	JaCoCo only	Total
checkstyle	2 653	2	3 263	5 918
commons-lang	2 783	13	154	2 950
commons-math	7 080	87	221	7 388
joda-time	3 884	14	76	3 974
mapdb	1 585	23	150	1 758
netty	8 195	35	1 297	9 527
orientdb	13 097	21	1 306	14 424
oryx	1 560	2	244	1 806

large number of generated methods in the bytecode (due to the necessary mechanisms of the Java language, which will be elaborated shortly). However, we were somewhat surprised to see that some methods were recognized only by Clover. This section also investigates the reasons for this difference. In any case, the impact of the difference in the recognized method sets can be significant. The results from section 5.1 were all produced for the two tools which were based on a different total number of methods. This makes difficult, for instance, the comparison of the overall coverage ratios because they involve different denominators for the two tools.

The actual causes of the different method sets are overviewed below.

Test methods Unit tests themselves should not be investigated for coverage, hence all methods of unit test classes needed to be excluded from further analysis. JaCoCo relies on the project description to determine the test methods. On the other hand, Clover tries to determine test methods by checking the class and method names, and in most cases this is reliable. However, in some cases when the test class names did not follow the naming conventions, Clover misclassified the tests as regular methods.

Compiler generated code The difference in favor of JaCoCo consisted of various methods generated by the Java compiler, *e.g.,* default constructors (if they were not given in the source), "`<clinit>`" methods, and access methods in the case of some nested class operations. Generated methods are considered for the coverage analysis by most bytecode instrumentation tools – including JaCoCo, however a source code-based tool like Clover may not include them. This issue results in additional methods appearing in bytecode coverage results, which can increase or decrease the overall coverage value.

Generated code All programs we investigated included code constructs that result in compiler generated methods, which are not visible in source code, only in bytecode (*e.g.,* default constructors and initializers). On the other hand, some projects generate a portion of the *source code* of the application on-the-fly using some external tools like ANTLR, or a configuration setting. In particular, most of the big differences between JaCoCo and Clover results of checkstyle were caused by this reason (see Table 5). The two tools handle this kind of code differently: while JaCoCo includes them in the same manner as any other regular code, Clover excludes them from the analysis. Since the tests of checkstyle do not cover any of

the generated code, the result is that JaCoCo uses a larger denominator than the other tool with a similar amount of covered elements in the nominators. In general, instrumentation tools may handle this situation differently, but usually they can be configured to consider the generated section of the source code as part of the code base.

5.2.5 Instrumentation

We found that in some cases the instrumentation itself modified the behavior of the tests, which might have influenced the list of executed methods. An example is in the joda-time program, where two specific test cases failed after being instrumented by Clover. This is because the tests utilize Java reflection to query the number of subclasses of the tested class, and – as Clover implements coverage measurements and test case detection by inserting subclasses into the examined class – these two tests failed on assertions right at the beginning of the test case. Similar failures occurred in the checkstyle project as well, where two of the test cases check if the classes they test have a fixed number of fields. However, with the additional fields that Clover inserts in the classes, these assertions fail.

5.2.6 Exception handling during coverage measurement

When JaCoCo instruments the bytecode, it inserts probes into strategic locations by analyzing the control flow of all methods of a class. If the control flow is interrupted by an exception between two probes, JaCoCo will not consider the instructions between the probes to be covered. The reason is that if a method throws an exception at the beginning of the caller method, JaCoCo marks the caller method as not covered because it misses the instrumentation probe on the exit point of the method. However, the instrumentation strategy of Clover is able to handle this situation and it will mark the caller method as covered because it simply considers the probe at the entry point of the method. Another reason for this issue was that JaCoCo computes lower coverage for tests that are expected to throw some exceptions (*i.e.,* annotated as `@Test(expected=SomeException)`. It is related to the above mentioned exception handling, and it is a known issue of JaCoCo.

5.2.7 Name encoding

A common reason for the differences was related to enums, anonymous and nested classes. The problem is that in some cases a method of such a class may get additional parameters when compiled to bytecode to access the members of its enclosing class. In other cases, the methods even lost some of their source code parameters. This resulted in different signatures of the source code and bytecode instance of the same method.

For example, a constructor like `MyEnum(String name)` of an `enum` type in the `pack` package will have the signature `pack/MyEnum/MyEnum(LString;)V` in the source code, while – due to technical requirements – the bytecode-based tools will see it as `pack/MyEnum/MyEnum(LString;ILString;)V`. Another example is when there is a private static class named `Bar` with a private constructor `Bar(final Foo f)` nested in a final class named `Foo`. The source code based tools recognize the constructor as `Foo$Bar(LFoo;)V`, while bytecode-based ones will see `Foo$Bar()V`.

Such missing or extra parameters in the bytecode make the signatures of these methods different in JaCoCo and Clover measurements. This difference prevented the automatic assignment of the methods of the two measurements and caused the reduction of JaCoCo coverage counts in our experiments.

5.2.8 Other

We also found some other, occasional reasons for the deviations. The first one was the different handling of some built-in methods of the `Object` class (for example, `equals`, `finalize` or `hashcode`). If these were redefined through multiple inheritance levels, both tools occasionally produced incorrect results for these methods. Due to this difference, both JaCoCo and Clover could report lower coverage on the same project. Another reason was that Clover had issues detecting the test cases that were called from test cases (see, for example, the class `c` in commons-math), which resulted in incorrect elements in the coverage results. Although it is possible to avoid calling test cases from test cases (even transitively), if this happens for any reason the resulting detailed coverage data might be unreliable. Note that the overall coverage of the test suite will not be influenced by this issue because the coverage will not be missed just recorded at a different program point.

5.2.9 Summary of difference causes

During our investigations of the differences listed above, we used the following approach. We tried to eliminate or fix the issues one by one in the hope to reach a state when the measurements produced by the two tools were synchronized. This way, we would have been able to categorize each difference as tool-specific or approach-specific. Finally, we were not able to uniquely classify all difference causes to one of the two categories, as we detail below.

First, we excluded the test cases that were failing because of the instrumentation, but this was rather a workaround than a solution. Second, we eliminated cross-module related issues by measuring these sub-modules individually, and we filtered out those methods from the covered set of a test case that were executed only during the setup/teardown phase. This was appropriate to to eliminate certain kinds of differences in our experiments, but in real applications it might eliminate important coverage information (depending on the definition and implementation of a test case and whether module or whole system coverage is needed). In addition, we relied on the Maven project hierarchy and examined the source path information of the classes, and filtered out those methods that were located in the test source directories, *e.g.*, `src/test`. This was required to filter out methods that were incorrectly treated as non-test methods.

To mitigate the remaining inaccuracies, we tried to synchronize the method sets (to make the individual test case and method coverage result comparable), for which we defined a set of criteria. We thought that, as software engineers usually work with source code, the synchronized set should be the set of methods actually appearing in the source code. We wanted to verify if Clover produced this list accurately. For that we used the SourceMeter static analysis tool[22], and found that there were no differences between the two lists in any of the programs. Thus,

[22] https://www.sourcemeter.com/

for each subject program, we created a list of methods based on the source code (excluding *e.g.,* compiler-generated methods). We also made an assignment between the JaCoCo and Clover methods by hand. With this workaround – although the two tools recognized the same methods with different names – we could compare their coverage values for the individual methods.

We denote the results using these sets as JaCoCo^{sync} and Clover^{sync} results. The important property of the synchronized sets is that they are based on the same set of methods, hence in this step we eliminated inaccuracies in the method sets regardless of their reason.

To summarize, Table 9 shows how the issues we found persist in the different measurements. The first column names the issue, the second one states whether we considered the issue being clearly tool specific, approach specific, or something in between. We did not categorize any of the issues as purely "approach" specific because we think that any of the potential differences could be theoretically aligned in source code and bytecode instrumentation. However, a number of such issues are not expected to be handled equivalently in a realistic tool or this would be impractical. For example, the standard name encoding in bytecode would be unusual to identically follow in source code (see the enum example above).

The third column in the table shows whether the corresponding issue was present in the JaCoCo^{glob} measurements, and the last two columns show whether the issue caused differences in the two kinds of comparisons we performed. We found that although there were some tool specific issues, most of them are generalizable, and will probably be applicable to other bytecode and source code instrumentation based tools. Indeed, we found that most of the specific issues of JaCoCo are present in Cobertura and JCov, the other two bytecode instrumentation tool we considered, as well.

Table 9: Presence of issues with different levels of filtering

	Issue	tool spec.	J^{glob}	J/C	J/C^{sync}
1.	cross-submodule coverage	yes	–	–	–
2.	untested sub-modules	yes	–	–	–
3.	test case preparation and cleanup	yes	●	●	–
4.	recognized method sets	partially	●	●	–
5.	instrumentation	yes	–	○	○
6.	exception handling during cov. measurement	yes	●	●	●
7.	name encoding	partially	●	●	○
8.	other	partially	●	●	●

Measurement (Column 3) –: issue is not present in measurement; ●: issue is present in measurement. **Comparisons** (Columns 4–5) –: caused differences can be and are automatically eliminated; ○: caused differences are manually eliminated; ●: differences are present.

5.3 Differences Due to the Instrumentation Approach – **RQ3**

In the previous section, we listed the causes of differences in the coverages produced by the two measurement tools. Some of them turned out to be due to tool-

specific design decisions, while others seemed to be inherent due to the differences in the fundamental approach, namely bytecode vs. source code instrumentation. Finally, in some cases we could not determine if a specific difference belonged to the "tool-specific" or "approach" category. By using the synchronized method sets and eliminating other tool-specific differences that were possible, we arrive at the JaCoCo^{sync} and Clover^{sync} sets of measurements. The differences in these we attribute to most probably the fundamental differences in bytecode vs. source code instrumentation, however we cannot be sure that there are no more tool-specific issues present. In this section, we quantitatively compare these two coverages. We take a look again at the total coverage ratios, as well as the per-test case and per-method details.

Table 10 shows the comparison of all three aspects at a general level, in which the final results of Section 5.1 are repeated for convenience, and the corresponding data are presented for the synchronized versions.

The differences of the overall coverages are shown in the columns 2–7 of the table. As expected, in the synchronized set of results there are fewer differences between the two measurements (the largest difference is 0.64%, in contrast to 40.05% of the outlier in the previous set). These results also indicate that the coverage of JaCoCo is never greater than that of Clover with the synchronized set of methods. This suggests that bytecode instrumentation typically demonstrates the *safe but imprecise* case, because a smaller coverage may lead to wasted effort but not to false confidence.

Table 10: Differences in overall coverage with the original and synchronized versions of the tools

Program	JaCoCo	Clover	orig.diff.	JaCoCo^{sync}	Clover^{sync}	sync diff.
checkstyle	53.77%	93.82%	-40.05%	93.81%	93.81%	0.00%
commons-lang	92.92%	93.28%	-0.36%	93.12%	93.30%	-0.18%
commons-math	84.92%	84.65%	+0.27%	85.23%	85.35%	-0.12%
joda-time	89.52%	89.94%	-0.42%	90.17%	90.19%	-0.02%
mapdb	74.64%	76.06%	-1.42%	76.03%	76.11%	-0.08%
netty	40.92%	40.18%	+0.74%	39.79%	40.43%	-0.64%
orientdb	27.01%	28.01%	-1.00%	28.02%	28.05%	-0.03%
oryx	29.51%	28.75%	+0.76%	28.67%	28.67%	0.00%
average	**61.65%**	**66.84%**	**-5.19%**	**66.86%**	**66.99%**	**-0.13%**

The comparison of the per-test case results is contained in the columns 2–3 of Table 11. Here, the overall Hamming distances can be compared, which have been computed jointly for all test cases from the respective coverage matrices. It can be observed that the average differences are reduced in different degrees: while in the case of commons-lang the reduction was minimal, the difference almost disappeared in the case of oryx. The reduction is dependent on the internal structure and relations of the programs' methods and tests, and cannot be directly predicted from the different properties we measured in other experiments.

Table 11: Differences in per-test case and per-method coverages with the original and synchronized versions of the tools

Program	orig. H.	sync. H.	orig. strict	orig. nostrict	sync. strict	sync. nostrict
checkstyle	0.005%	0.002%	25	1	16	0
commons-lang	0.014%	0.012%	152	5	132	5
commons-math	0.034%	0.005%	536	26	251	7
joda-time	0.157%	0.031%	444	2	308	2
mapdb	3.062%	1.165%	475	9	81	3
netty	0.155%	0.013%	766	167	484	98
orientdb	0.013%	0.008%	136	6	61	6
oryx	0.187%	0.004%	9	4	1	0
average	**0.450%**	**0.160%**	**318**	**28**	**167**	**15**

The per-method coverage differences are also presented for the final results for the respective measurement levels in columns 4–7 of Table 11. They show the numbers of test cases when there is a disagreement between the two tools according to the two levels of strictness, as explained in Section 5.1.3. In particular, columns "orig. strict" and "orig. nostrict" correspond to the sums $CltJ+JltC$, and $Czero+Jzero$ from Table 7, respectively, while the other two are the same for the synchronized measurements. As can be observed, the synchronization improves this measurement as well, and the improvement rates are again very different for the individual subjects. The counts roughly halved both in the strict and non-strict cases, however there are notable cases when this was more significant (*e.g.,* mapdb) and also where it was much smaller (*e.g.,* commons-lang).

Answering RQ3 is not easy: while we have seen that there might be notable differences in the "synchronized" data sets showing differences due to the instrumentation approach, tool specific ones cannot always be sorted out reliably.

5.4 Impact on Test Case Prioritization and Test Suite Reduction – **RQ4**

In Section 2.2, we listed leading applications of code coverage measurement, and how they are possibly impacted by the inaccuracies of the tools. The results presented earlier in this section showed that the inaccuracies may directly impact some of the applications, most notably white-box testing, and that this can be directly measured/predicted. However, it does not directly follow if the inaccuracies would have a similar effect in applications where code coverage is indirectly used to achieve a different purpose. We selected code coverage-based test case prioritization and the related test suite reduction (Rothermel et al, 2001; Yoo and Harman, 2012) to quantify the impact of code coverage inaccuracies.

Informally, *test case prioritization* takes the list of test cases of a test suite and produces a specific order of their execution, which is believed to maximize the chances of early defect detection, localization and correction. Typically, defect detection is the primary concern, but in this work, we concentrate on both detection

and localization. The goal of the former is to have failing test cases because they indicate that there are faults *somewhere* in the system. On the other hand, in fault localization we aim to find the causes of the faults, in other words pinpoint to the *location* of actual defects in the code. Both activities may be aided by the use of code coverage information, but coverage needs to be used differently:

- For fault detection, the usual approach is to maximize coverage at the beginning of the prioritized list because it is naturally expected that elements that are not covered by the test cases may not exhibit faults.
- On the other hand, successful localization highly depends on how much the test cases are able to exhibit different program behavior; *i.e.,* if the test cases show similar behavior on different program elements, these elements may be indistinguishable from this respect. Consequently, for fault localization, those test cases should be chosen high in the priority that distinguish between different program elements This is often quite different than simply highest coverage. Many fault localization algorithms exploit this fact, such as Raptor (Gonzalez-Sanchez et al, 2011), FLINT (Yoo et al, 2011), and Partition-based (Vidács et al, 2014).

A practical use of the prioritized list of test cases is that not all of them are executed, but only the first N elements of the prioritized list are selected. This can happen in various settings. First, if faults are detected or localized the testing may be terminated. Second, test selection may be terminated at the first point where maximum coverage (or a suitable fault localization metric) is reached. Finally, if there are resources to execute only a fixed number N of tests, this is a suitable approach because the chances of successful testing are maximized by using a suitable prioritization algorithm. The test suites are then either minimized by permanently discarding the remaining test cases or limited only for the execution (Yoo and Harman, 2012). Section 5.4.2 deals with test suite reduction, which is based on the prioritized list of test cases investigated in Section 5.4.1.

Our rationale for selecting these applications is that they have solid algorithmic background and the outcome of the algorithms may significantly influence test effectiveness and efficiency.

In this section, we rely on our first set of coverage data used in Section 5.1 for our Research Question 1. These are the "raw" coverages produced by the tools, and are not influenced by our "synchronization" efforts for Research Questions 2 and 3. We do this because, in most cases, the coverage tools and their results are used "as is": the users do not make an effort to additionally process the data. Thus, we use coverage data denoted by JaCoCo and Clover for these comparisons.

5.4.1 Test case prioritization

In this experiment, we used four test case prioritization strategies: three optimized for maximal fault detection, and one for fault localization. There are many strategies for coverage-based fault detection prioritization, but the so-called *general* and *additional* are probably the most widely used ones (Rothermel et al, 2001). We will also consider a variation of the second one called *additional with resets*.

The *general* strategy greedily assigns a higher rank to those test cases that produce highest absolute coverage (that is, the test cases are simply ordered by their coverage value). The *additional* algorithm is a bit more clever in that it looks

for test cases that contribute the most to the not yet covered elements (that is, it starts with the highest covering test case and then it greedily selects the test cases based on their additional coverage). A common issue with these approaches is that once a high coverage is attained, the algorithm cannot select but randomly from the remaining test cases. Therefore, an extension to the second algorithm (*additional with resets*) restarts the greedy selection once no improvement in the additional coverage can be obtained. In particular, it resets the coverage counter to zero whenever the maximum coverage is reached, and continues to append test cases to the prioritized list as if it was creating a new list from the remaining test cases (Rothermel et al, 2001).

Our selected algorithm for fault localization aware test case prioritization was the *partition-based* algorithm (Vidács et al, 2014). The basic idea behind this algorithm is that the code elements in the coverage matrix are *partitioned* according to their coverage patterns produced by the test cases (that is, equal matrix columns constitute a partition). Since it is advisable that the code elements are distinguishable from each other during fault localization, the finest partitioning is sought in this prioritization. The algorithm greedily selects the test cases that best divide the existing partitions that were obtained with the earlier test cases. To do this, it divides in each step the code elements into covered and uncovered subsets, and then it is recursively invoked to these subsets to choose the new test cases.

These four algorithms typically produce quite different rankings in the prioritized lists, but all of them are highly sensitive to the coverage data. Some of them mostly depend on the overall coverage (such as *general*), while fine details in the coverage patterns may have big influence on the others, for example *partition-based*. Hence, in the first experiment, we wanted to compare for each algorithm how their prioritized lists differ when computed by the two coverage tools.

Note that certain test suite prioritization algorithms (and reduction algorithms as well) make arbitrary choices in the cases when more than one item has the same priority value. Therefore, due to their non-deterministic nature they could produce different results even on the same coverage data. Hence, we designed our algorithms and their inner data structures to be deterministic.

To compare the prioritizations, we used Kendall's τ_B rank correlation coefficient, which is known to be suitable for handling ties.

Table 12: Prioritization: Kendall's τ correlation between JaCoCo and Clover results

Program	General	Additional	Additional with resets	Partition-based
checkstyle	0.801	0.274	0.666	0.049
commons-lang	0.890	0.373	0.751	0.043
commons-math	0.886	0.267	0.682	0.055
joda-time	0.940	0.245	0.672	0.018
mapdb	0.695	0.124	0.815	0.064
netty	0.567	0.129	0.476	0.266
orientdb	0.956	0.329	0.736	0.364
oryx	0.684	0.440	0.659	0.379

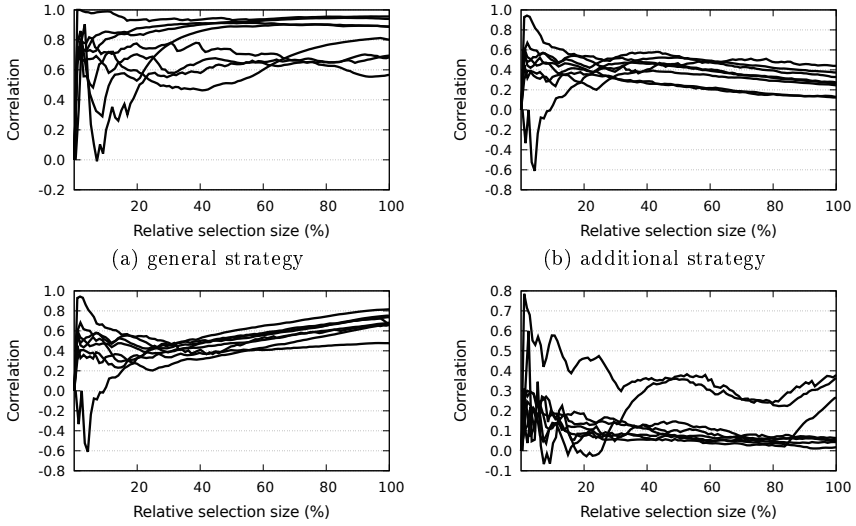
Table 12 summarizes the associated results. We can observe that the correlation for the *general* strategy is high or moderate for all programs, leading to a conclusion that the differences in the coverage tools have relatively low influence on the final rankings produced by this algorithm. The ranked lists produced by the *additional with resets* strategy show similar correlation, only slightly lower than for the *general* strategy (an exception is subject mapdb). However, the third column shows that the *additional* strategy is much more influenced by the minor differences in the coverage measurements: the highest correlation for this strategy (0.44) is much worse than the lowest one of *general* (0.567), and it has some really low values too (0.124).

These results are caused by the different behavior of the algorithms and how they react to the individual differences of the coverage vectors. In all of them, when there are more test case candidates to be selected as the next item of the prioritized list, the selection is arbitrary (the "random effect"). The good performance of the *general* strategy can be attributed to the fact that before reaching the maximum coverage, the overall coverage is dominant and the smaller differences do not have a big impact here. In the case of this algorithm, the random effect will be smaller, and it will increase only after reaching the maximum coverage. On the other hand, the different behavior of the *additional* and *additional with resets* strategies is more related to the random effect: *additional* reaches maximum coverage relatively early and then it starts extending the list with elements in an arbitrary order. The *additional with resets* strategy is more deterministic: after the reset it will work with the same determinism again and again, hence it will more resemble the behavior of *general*.

Finally, the *partition-based* strategy for fault localization prioritization is the most sensitive to the coverage differences. Except for maybe orientdb, oryx and netty, it shows no correlation at all. Note that these are the subject programs that have modules, the others are single module programs. In addition, these programs have low coverage. When selecting the next item in the ranked list, the partition-based strategy prefers the test case that best splits the method sets into two parts. In each step, the algorithm tries to split a partition into parts of mostly equal size, and select the best test case for it. But all the uncovered methods form one partition, which is very large in these cases (due to the low coverage). When this large partition is to be split, the selection of the test case is arbitrary, and the implementation will select the first test case in the original list. Thus, the correlation between the two ranked lists will depend on the correlation of the original lists. This correlation is higher for the modular programs because the list of the test cases are implicitly grouped by the modules.

The correlation values from the previous table were computed for the whole prioritized lists of test cases. However, often only the beginning of the prioritized list is used (such as for test suite reduction, which is discussed in the next section). Further, the differences in the code coverage measurements can vary depending on the stage of the algorithm: they may be different among the first selected test cases and later, when more test cases are already processed. Hence, to see these effects in more detail, we computed the correlation values for each selection size starting from the first test case in the prioritized list up to the whole set.

More precisely, Kendall correlation coefficients were computed for each prefix of the ranked lists, which are shown in Figure 5. In these graphs, the X axis shows how many elements from the beginning of the two prioritized lists were



(a) general strategy (b) additional strategy

(c) additional with resets strategy (d) partition-based strategy

Fig. 5: Test case prioritization: correlation for different selection sizes and strategies

considered for computing the correlation (in percentage), while axis Y shows the actual correlation value. The curves correspond to the different programs, but we did not distinguish between them because it bears no information for this discussion. The last value in each graph corresponds to the values in Table 12. As can be seen, the different strategies behave differently, but it is common that, at the beginning, all strategies are very sensitive to the small coverage differences. Also, the calculation of the correlation is statistically less significant with smaller number of elements in the data sets. In particular, the corresponding p-values are greater than 0.05 for the first 20–50 elements. Hence, the investigation about the data in these charts should not focus on the very beginning of the curves (also graphically they show quite erratic behavior).

The specific observations we can make from this data are the following. In the case of the *general* strategy (Figure 5(a)) the curves are forming a V-shape: the correlation is suddenly reduced and then grows back as we compare more and more elements of the prioritized lists. The characteristics of these drops are program dependent, but at some point the correlations grow back and become steady when more elements are considered. This shows that small coverage differences have local influences on the ranking, but globally they have small impact.

The results for the *additional* strategy (Figure 5(b)) are different: as this strategy incorporates some random factor once the test cases in the first part of the prioritized list give full coverage, the correlation after this point starts to decrease as the effect of the arbitrary selection accumulates. However, the curves for the different programs are more similar to each other than those of the general strategy.

Results for the *additional with resets* strategy (Figure 5(c)) are very similar to the results of *additional* until the first reset (it is typically between 10 and 30 percent of the test cases). However, due to the elimination of the random factor

with the reset, in the remaining parts of the curves they show similar behavior to the *general* strategy. As more elements are compared, the correlation between the two lists grows until the full lists are compared, and this final correlation is comparable to the *general* strategy.

Since the *additional with resets* strategy is often seen as the best coverage-based greedy strategy for fault detection prioritization, this result indicates a high risk when using different code coverage tools for this application. Namely, when a relatively low number of tests are selected, the influence of the coverage tool is quite high on the results.

Finally, the *partition-based* strategy (Figure 5(d)) behaves very differently. As mentioned above, this strategy selects the next test case in the ranking, which best splits the method sets into two parts. As a result, the first very few elements on the two lists for each program are expected to be the same given the relatively small difference in the coverage vectors. But further selections use smaller partitions of the method sets, thus it is more probable that multiple test cases indicate the same "best" split. In this case, the selection of the next one is arbitrary among the candidates. It might happen that two "best" test cases split the corresponding method partition in different ways, which heavily affects the subsequent selections and rankings.

For 5 programs, after the first erratic values of the short list comparisons, the correlation drops down showing that the lists derived from the two coverage measurements are very different; practically, we cannot observe any correlation between them due to the cumulative differences mentioned above. These are single module programs, where all test cases are potential candidates. However, the modular architecture programs orientdb, oryx, and netty have an inherent partitioning aligned with the module boundaries. This means that the number of potential candidates for a selection is limited and randomness has smaller effect. The *partition-based* strategy results are aligned with these inherent partitions regardless of the coverage measurement method; the algorithm works like if it were ranking the test cases for each module independently. This results in more correlated rankings than for single module programs. The results for all the programs show that the first 2–3 elements match exactly, the next few are the same but in different order and from there onward the lists show very low overall correlation. To conclude, the partition-based prioritization algorithm is very sensitive to the small coverage differences of the individual test cases, which is also true for multi-module programs, but especially holds for single module ones.

5.4.2 Test suite reduction

As mentioned, in test suite reduction, a given number of elements from the beginning of the prioritized list is selected. For this experiment, we followed two scenarios. In the first one, we stop the selection when the current subset of the test cases reaches the coverage of the unreduced test suite, and then compare the attained reductions. In the second scenario we measure the differences of the coverages when any fixed size of the reduced subset is used. These experiments include reductions based on the *general*, *additional* and *additional with resets* prioritization strategies.

Note, that in both scenarios the measurements for *additional* and *additional with resets* strategies are the same: before the first reset *additional with resets*

is equivalent to *additional*, but since the first reset occurs when the maximum possible coverage is reached, the differences in the coverages after this point will be constant. So, we will present *additional* and *additional with resets* results together.

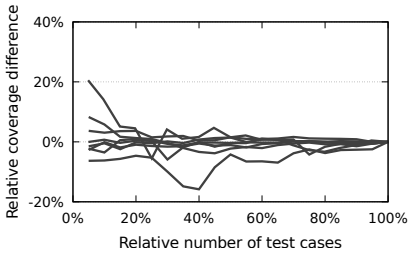
Table 13 shows the results for the first scenario: namely, the reduction values for the three fault detection algorithms that could be obtained for the subsets of test cases, which achieve the original coverage of the unreduced sets. The reduction is given as a relative number of eliminated test cases. Apart from the obvious advantage of the *additional* strategies over *general* (which is not the topic of this paper), the differences between the two coverage tools are not that obvious as we have seen for test case prioritization. For the general strategy, mapdb seems to be an outlier; the difference between the reduction rate of the results achieved by the two coverage methods is more than 11%, which could be a consequence of the high Hamming distances between the coverage vectors for this program (see Figures 1 and 2). On the other hand, checkstyle and oryx, for instance, have very different Hamming distances but still they demonstrate similar test suite reduction differences.

Table 13: Test suite reduction without reducing coverage for the different strategies

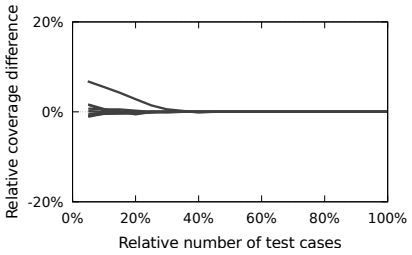
Program	General			Additional Additional with resets		
	JaCoCo	Clover	Diff.	JaCoCo	Clover	Diff.
checkstyle	0.20%	3.03%	2.83%	75.95%	78.04%	2.09%
commons-lang	0.69%	0.72%	0.03%	69.95%	70.16%	0.21%
commons-math	0.08%	0.28%	0.20%	77.04%	77.53%	0.49%
joda-time	0.54%	0.92%	0.38%	76.01%	76.21%	0.20%
mapdb	0.11%	11.17%	11.06%	89.28%	89.34%	0.06%
netty	0.23%	0.57%	0.34%	87.85%	88.40%	0.55%
orientdb	0.25%	0.25%	0.00%	70.63%	71.76%	1.13%
oryx	7.21%	3.36%	3.85%	59.13%	60.09%	0.96%

The results for our second reduction strategy are shown in Figure 6. Here, we calculated the effects of the differences for various fixed reduction values from 0–100% (in these diagrams, the test suite sizes increase from left to right). We used *general*, *additional*, and *additional with resets* strategies, and calculated the relative differences of the overall coverage values compared to Clover. The X axis represents the number of test cases (relative to the size of the whole test suite) and the Y axis shows the difference itself. Again, we do not distinguish between the subjects because only the trends are important.

It can be observed that the behavior of different programs in the *general* measurements (Figure 6(a)) are different, but in general, the smaller size the reduced suite has, the greater difference can be measured between the coverages of the reduced suites. In other words, a higher reduction rate introduces more uncertainty in the results. In general, we found that if the size of the reduced suite was over 20% of the full suite then the difference of coverages mostly remained under 5%. The shape of the difference depends on the properties of the subject program and its tests. The situation is different for the *additional* measurements (Figure 6(b)). The decrease of coverage differences is clearly visible, and it is even much faster. Except for one program, the difference of coverages remains below 2% at a 10%



(a) general strategy



(b) additional and additional with resets strategies

Fig. 6: Test suite reduction: coverage differences for general and additional strategies

test suite size and above. The fast convergence is caused by the algorithm itself, which aims to reach full coverage as quick (with as few test cases) as possible. Thus, the two coverage values will approach their maximum with monotonically smaller steps, which implies gradually smaller and smaller differences.

5.4.3 Summary of impacts

In summary for our RQ4, our investigations about the impacts of code coverage inaccuracies showed that they were really unpredictable on the chosen applications. For some algorithms the impacts were high and less for the others. A notable example is the test case prioritization algorithm *additional with resets*, which is often considered the best greedy strategy. The influence of the coverage tool was quite high in this case, when a relatively low number of tests were selected (which is often the case in practice).

6 Discussion

6.1 Interpretation of the Results

In this work, we performed experiments on the granularity of Java methods to find out the differences between the bytecode instrumentation approach and source code instrumentation with respect to the final code coverage results. In particular, most of the detailed experiments have been performed using two tools, JaCoCo and Clover, which we selected as representatives of the two instrumentation approaches. We selected Clover as the source code-based tool from two candidates in our shortlist (which produced very similar results), and used it as the comparison basis in the experiments. We started with three tool candidates in the other category, but we found out that they produce similar results, so we selected JaCoCo as the representative of bytecode-based approaches. In our experiments, we relied on real size Java systems with realistic test suites, so we believe that testing practitioners and researchers can benefit from our findings as well.

We have seen that the bytecode level and the source code level coverage measurements can produce very different results (answering RQ1). In general, the

overall differences are low (below 1.5%), but the different properties of the subject systems and the measurement methods may result in very large differences as well. This can be exemplified by the subject checkstyle, where the generated methods caused a difference of about 40%. Furthermore, differences can be identified in both directions: in some cases JaCoCo reports more coverage than Clover and vice versa.

On a more detailed analysis level, per-test case and per-method differences also showed discrepancies in both directions. Overall, in some cases the differences are minimal (below 1%), however since this is very much project dependent, we measured relatively high differences as well (higher than 20% in some cases; see Figures 1 and 2). The differences might affect a large portion of the methods of a program, even around 30%, as it can be observed from Table 7.

The causes of these differences are various (RQ2). There are tool specific ones like the different sub-module handling of the used tools, or the handling of test setup and teardown methods; these are independent from the selected instrumentation method. These can be eliminated by filtering the results (although this might not be fully automated). Other tool specific features like the influence of the instrumentation on the behavior of the subject system tests are integral parts of the tools and in general cannot be avoided. Finally, deviations like the different issues on method set recognition and name encoding are mostly determined by the instrumentation method, not the tool.

Theoretically, some of these differences could be eliminated using additional information but not all (answering RQ3). In order to assess the amount of inherent differences that are not attributed to tool specific issues, we tried to eliminate the differences, and we managed to do so in many cases by adjusting or filtering the measurements (see Table 9). However, the remaining differences still caused deviations in the coverage values, though they were much lower than the differences for the unmodified tools: at most 0.64% in the total coverage (see Table 10). These results show that with a careful tool design, more predictable results could have been achieved, but the full alignment of the different tools seems practically impossible. Since it is not expected from a tool user to make such corrective actions in the first place, as a general advice, tool users should examine the particular working methods of the tool and be aware of its limitations. Our list of possible reasons for the differences may be used as a guideline on how to avoid and workaround the inaccuracies of the bytecode level instrumentation tools with respect to the source code instrumentation approaches, and in particular to the tools we investigated.

In the last part of our experiments, we checked how the differences in the coverage measurement influenced the results of an application that used coverage as its input (RQ4). We applied different test case selection and prioritization algorithms which were all based on the coverage values computed by the two tools. We found that the coverage differences had various influence on the results of the algorithms; the impact was dependent on the different properties of the subject programs and the algorithms themselves (answering RQ4). However, for example, the most popular test prioritization algorithm, *additional with resets*, might produce a low correlation of 0.476 between the results of the two tools, which indicates that any practical application or research based on a tool with such inaccuracies imposes a high risk of the validity of the results (see Table 12).

We systematically searched for correlations between the subject program properties (modularity, method and test case numbers), raw measurement values (total

and per-test case coverages, coverage differences), and the application results (correlations, reduction rates), but we did not find notable dependencies that could be generalized. It seems that the influence of coverage difference on the applications is subject and algorithm dependent. For example, the average Hamming distance between the individual coverage vectors of mapdb and commons-lang is very different, 3.062% and 0.014%, respectively. Yet, the correlations between their prioritized lists using the *additional with resets* strategy are similar, 0.815 and 0.751. The programs joda-time and netty, which produce very similar average Hamming distances (0.157% and 0.155%) but different correlations of the prioritization (0.672 and 0.476) are examples for the opposite relation. The effect is that the impact of the inaccuracies in the coverage measurement are unpredictable, hence special care should be taken if code coverage is not used only as a general test completeness measure, but as a base for a more complex analysis.

6.2 Threats to Validity

The main aim of the paper was to investigate the effects of the different instrumentation techniques on code coverage measurement results. We applied empirical measurements using eight subject programs and two specific tools (we started the investigation with four tools). Clearly, this raises the question how generalizable the results are to other tools using similar techniques. The subjects were selected from different domains and had different sizes (both in terms of code and tests), but were all actively developed community software.

The two final tools we selected for the detailed examination were among the most widely used coverage tools representing the two instrumentation approaches, and they were mature and actively developed. We carefully analyzed the data in the preliminary experiments from Section 4.2, and came to the conclusion that there was not a big difference among the candidates from our shortlist in either category. However, limiting the detailed analysis to two tools might impose a threat to the generalizability of the results to other tools. When interpreting the results, we tried to separate the tool-specific issues from the approach-specific ones, and the results of source code instrumentation with Clover were verified with manual instrumentation.

A possible threat is that we slightly modified the instrumentation process of JaCoCo by adding a test execution listener that detected the start and the end of the execution of a test case. The results obtained with this modification may not directly translate to the coverage results everyday users would experience with the stock version of JaCoCo. However, we compared the results of the unmodified JaCoCo measurement to our version in terms of actually covered program elements and found no significant differences.

Our experiments showed results with respect to method-level coverage analysis. Generalization to other granularities such as components or statements may not directly be possible.

7 Conclusions

The results have shown that even at the method level, significant differences occur between the bytecode and the source code level instrumentation measurements. This confirms the results of some related work (*e.g.,* (Li et al, 2013; Alemerien and Magel, 2014)). Some of the differences can be eliminated, but some cannot or their elimination would not be practical. These differences, when used in different testing applications, will undoubtedly have an influence on the application results. But the kind and level of influence cannot be generally predicted, as it depends on the subject program and the application itself. A small difference in coverage may be amplified at the application level, and a big coverage difference may have minor impact.

As a conclusion, we may say that the discrepancies between the different instrumentation approaches might but not necessarily have influence on code coverage applications. It is thus safe to treat source code-based instrumentation as the correct approach to code coverage measurement, despite its disadvantages (which are summarized in Section 2.3). Our results indicate that bytecode-instrumentation may have serious disadvantages in terms of the accuracy of the results. The list of possible reasons for the differences may be used as a guideline on how to avoid and workaround the inaccuracies of the tools. This can then help assess the level of risk of measurement inaccuracies in particular applications of code coverage measurement.

The measurement data used in the experiments are available at:
http://www.sed.inf.u-szeged.hu/java-code-coverage.

Acknowledgements This work was partially supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences and by the EU-funded Hungarian national grant GINOP-2.3.2-15-2016-00037 titled "Internet of Living Things." We are grateful to Semantic Designs Inc. for providing a free research licence for their tool. The final publication is available at Springer via http://dx.doi.org/10.1007/s11219-017-9389-z.

References

Alemerien K, Magel K (2014) Examining the effectiveness of testing coverage tools: An empirical study. International Journal of Software Engineering and Its Applications 8(5):139–162

Binder W, Hulaas J, Moret P (2007) Advanced Java bytecode instrumentation. In: Proceedings of the 5th international symposium on Principles and practice of programming in Java, ACM, pp 135–144

Black R, van Veenendaal E, Graham D (2012) Foundations of Software Testing: ISTQB Certification. Cengage Learning

Emanuelsson P, Nilsson U (2008) A comparative study of industrial static analysis tools. Electron Notes Theor Comput Sci 217:5–21

Fontana FA, Mariani E, Mornioli A, Sormani R, Tonello A (2011) An experience report on using code smells detection tools. In: Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, IEEE Computer Society, ICSTW '11, pp 450–457

Fraser G, Arcuri A (2011) 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, New York, NY, USA, ESEC/FSE '11, pp 416–419

Gonzalez-Sanchez A, Abreu R, Gross HG, van Gemund AJC (2011) Prioritizing tests for fault localization through ambiguity group reduction. In: Alexander P, Pasareanu CS, Hosking JG (eds) Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on, IEEE, pp 83–92

Graves TL, Harrold MJ, Kim JM, Porter A, Rothermel G (2001) An empirical study of regression test selection techniques. ACM Trans Softw Eng Methodol 10(2):184–208

Harrold MJ, Rothermel G, Wu R, Yi L (1998) An empirical investigation of program spectra. In: Proc. of the 1998 ACM SIGPLAN-SIGSOFT workshop PASTE '98, ACM, pp 83–90

Inozemtseva L, Holmes R (2014) Coverage is not strongly correlated with test suite effectiveness. In: Proceedings of the International Conference on Software Engineering

Jia Y, Harman M (2011) An analysis and survey of the development of mutation testing. Software Engineering, IEEE Transactions on 37(5):649–678

Jones JA, Harrold MJ (2005) Empirical evaluation of the tarantula automatic fault-localization technique. In: Proc. of International Conference on Automated Software Engineering, ACM, pp 273–282

Kajo-Mece E, Tartari M (2012) An evaluation of Java code coverage testing tools. In: Proceedings of the 2012 Balkan Conference in Informatics (BCI'12), Faculty of Sciences, University of Novi Sad, pp 72–75

Kessis M, Ledru Y, Vandome G (2005) Experiences in coverage testing of a Java middleware. In: Proceedings of the 5th International Workshop on Software Engineering and Middleware, ACM, New York, NY, USA, SEM '05, pp 39–45

Li N, Meng X, Offutt J, Deng L (2013) Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools (experience report). In: Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on, pp 380–389

Lingampally R, Gupta A, Jalote P (2007) A multipurpose code coverage tool for Java. In: System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on, IEEE, pp 261b–261b

Ntafos SC (1988) A comparison of some structural testing strategies. IEEE Transactions on Software Engineering 14(6):868–874

Offutt AJ, Pan J, Voas JM (1995) Procedures for reducing the size of coverage-based test sets. In: In Proc. Twelfth Int'l. Conf. Testing Computer Softw, pp 111–123

Ostrand T (2002) White-box testing. Encyclopedia of Software Engineering

Perez A, Abreu R (2014) A diagnosis-based approach to software comprehension. In: Proceedings of the 22nd International Conference on Program Comprehension, ACM, New York, NY, USA, ICPC 2014, pp 37–47

Pinto LS, Sinha S, Orso A (2012) Understanding myths and realities of test-suite evolution. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, ACM, pp 33:1–33:11

Raulamo-Jurvanen P (2017) Decision support for selecting tools for software test automation. SIGSOFT Softw Eng Notes 41(6):1–5

Rayadurgam S, Heimdahl M (2001) Coverage based test-case generation using model checkers. In: Engineering of Computer Based Systems, 2001. ECBS 2001.

Proceedings. Eighth Annual IEEE International Conference and Workshop on the, pp 83–91
Rothermel G, Untch RJ, Chu C (2001) Prioritizing test cases for regression testing. IEEE Trans Softw Eng 27(10):929–948
Rothermel G, Harrold MJ, von Ronne J, Hong C (2002) Empirical studies of test-suite reduction. Software Testing, Verification and Reliability 12(4):219–249
Tengeri D, Beszédes Á, Havas D, Gyimóthy T (2014) Toolset and program repository for code coverage-based test suite analysis and manipulation. In: Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'14), pp 47–52
Tengeri D, Horváth F, Beszédes Á, Gergely T, Gyimóthy T (2016) Negative effects of bytecode instrumentation on Java source code coverage. In: Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016), pp 225–235
Usaola MP, Mateo PR (2010) Mutation testing cost reduction techniques: A survey. IEEE Software 27(3):80–86
Vidács L, Beszédes Á, Tengeri D, Siket I, Gyimóthy T (2014) Test suite reduction for fault detection and localization: A combined approach. In: Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on, pp 204–213
Yang Q, Li JJ, Weiss DM (2009) A survey of coverage-based testing tools. The Computer Journal 52(5):589–597
Yoo S, Harman M (2012) Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability 22(2):67–120
Yoo S, Harman M, Clark D (2011) Flint: Fault localisation using information theory. Tech. rep., University College London
Yoo S, Harman M, Clark D (2013) Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. ACM Trans Softw Eng Methodol 22(3):19

Appendix A Comparison results of source code based tools

In this appendix we present additional results of comparing the results of the two methods employing source code instrumentation, Clover and Test Coverage.

As these two tools generate instrumented source code, it is possible to compare their instrumentation algorithms on a high level by investigating the instrumented code itself. To do so, we manually checked the instrumented sources and investigated the probe points, *i.e.,* the locations where extra code were injected into the original source code. We found that the two tools identified and instrumented exactly the same source code elements. The main technical difference between the two tools is that while Test Coverage uses boolean vectors to store coverage data, Clover has a complex mechanism for calculating which part of the production code is exercised (this enables per-test coverage measurement as well). Thus, in general, Test Coverage inserts less extra code into the original source. Another difference is the handling of such code which do not have a conventional form of a Java method but will be included in the bytecode as a special method (*e.g.,* static initialization code of the class or anonymous methods). Both tools recognize and instrument

these parts of the source code, but Clover reports them as methods, while Test Coverage includes them in the class coverage only.

In the next step, we calculated the raw overall coverages for our subject systems with these two tools in order to see how much their results differ. Table 14 shows the associated results. Columns 2 and 3 show the overall coverage ratios as produced by the tools, while the last column includes the percentage difference. The numbers in the last row represent the averages of the absolute differences.

Table 14: Comparison of the overall coverages computed by the source code tools (Clover and Test Coverage)

Program	Clover^{glob}	Test Coverage	Clover^{glob} vs. Test Coverage
checkstyle	93.82%	93.77%	-0.05%
commons-lang	93.28%	93.13%	-0.15%
commons-math	84.65%	85.59%	+0.94%
joda-time	89.94%	90.94%	+1.00%
mapdb	76.06%	78.58%	+2.52%
netty	46.66%	48.93%	+2.27%
orientdb	39.84%	39.92%	+0.08%
oryx	27.51%	27.68%	+0.17%
average diff.			**0.90%**

We can observe that the above mentioned differences of the tools cause small differences in the overall coverage results. Note, that we used Clover^{glob} which is the value measured globally, including cross-submodule coverage for submodule-based systems, and this is how Test Coverageworks too. Unfortunately, we were not able to produce per-test coverage values using Test Coverage, as this would have required the individual execution of the test cases and consequently large-scale modifications in the build environments. Hence, we could not perform such a comparison of the tools.

Appendix B Comparison results of bytecode based tools

In this appendix we present additional results of comparing the results of the three tools employing bytecode instrumentation: JaCoCo, JCov and Cobertura.

We calculated the raw overall coverages for our subject systems with these tools in order to see how much their results differ. Table 15 shows the associated results. Columns 2–4 are the overall coverage ratios as produced by the tools "out of the box" (*i.e.*, without any modifications made to them,[23] only the necessary parameters have been set). The last three columns show the pairwise differences in the percentage, while the numbers in the last row represent the averages of the absolute differences.

[23] There was one exception to this: in Cobertura, we disabled the feature of skipping the analysis of generated source code, as this was not implemented in the other two tools.

Table 15: Comparison of the overall coverages computed by the bytecode tools (Cobertura, JaCoCo and JCov)

Program	JaCoCo	JCov	Cobertura	JCov vs. JaCoCo	Cobertura vs. JaCoCo	Cobertura vs. JCov
checkstyle	53.85%	52.20%	54.79%	-1.65%	0.94%	2.59%
commons-lang	93.29%	92.81%	94.08%	-0.49%	0.79%	1.28%
commons-math	85.59%	87.41%	88.78%	1.82%	3.19%	1.37%
joda-time	91.36%	91.33%	91.55%	-0.03%	0.19%	0.22%
mapdb	79.65%	79.12%	78.46%	-0.53%	-1.20%	-0.67%
netty	47.41%	49.10%	45.51%	1.69%	-1.90%	-3.59%
orientdb	38.40%	42.02%	42.23%	3.62%	3.83%	0.21%
oryx	29.62%	26.33%	25.60%	-3.29%	-4.03%	-0.74%
average diff.				**1.64%**	**2.01%**	**1.33%**

Quantitatively, the differences between these tools were at most 4%, and the behavior of the tools was different on the different subjects. We made some deeper but basically still quantitative analysis: we compared the per-test coverage results of the tools. Unfortunately, we were not able to produce such results using Cobertura, so we compared JaCoCo and JCov in this respect.

In Figure 7 we present the differences in test case coverage vectors. For each test case, we use a coverage vector in which each element corresponds to a single code element. We compared such vector pairs for JaCoCo and JCov for each test case using the Hamming distance measure, and normalized the result by the length of the vectors. Figure 7 shows the corresponding data in form of histograms.

For the first four subject programs, data shows that most of the vector pairs are the same and the difference is less than 1% for the others. For mapdb and netty, there are very few vector pairs that match exactly, but most of them are still close to each other. In the case of oryx and orientdb, about half of the vector pairs matches exactly, the difference in the majority of the cases is less than 1%, but there are differences as high as 5% or even 14%. After the manual investigations, most of these high differences are found to be tolerable outliers.

Similarly to the previous set of experiments, we performed per-method comparison of the coverages. Here the vectors were assigned to code elements and a vector element corresponded to a test case. Figure 8 shows the differences in these vectors for JaCoCo and JCov in form of histograms.

We got the same results as in case of test case vectors for the first 4 subject programs, namely, most of the vector pairs are the same and the rest of them differ at most 1%. Here, the last 4 programs are similar to each other: a lower part of the vector pairs matches exactly, but there are some higher differences as well. Later, these high differences are turned out to be explainable outliers.

To find the cause of the differences (especially of the high Hamming distances) we observed from these experiments, we investigated the detailed coverage of the three tools manually as well. Our conclusion was that the main cause of the differences was mostly due to the slightly different handling of compiler generated methods and nested classes in the bytecode (such as the methods generated for nested classes). Since the overall quantitative differences were at most 4% and they were concerned mostly generated methods, which are less important for code cov-

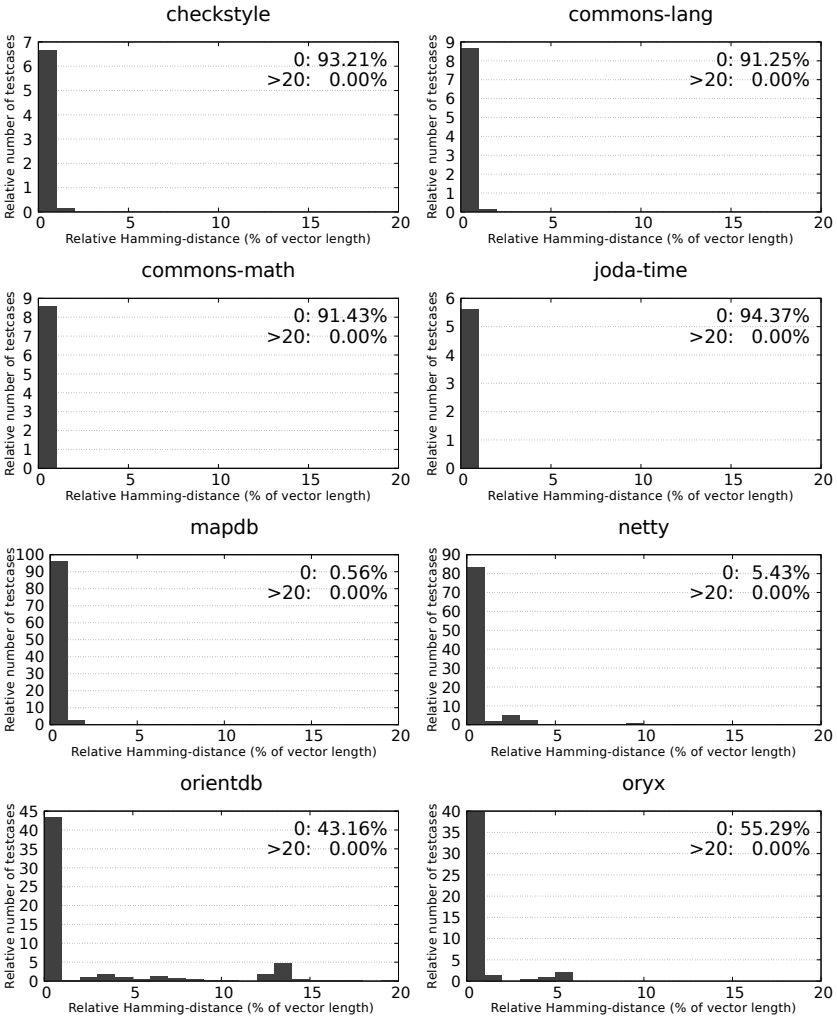


Fig. 7: Relative Hamming distances of test case vectors (JaCoCo vs. JCov)

erage analysis, and the high individual Hamming distances could also be traced back to these methods, we concluded that one representative tool of the three should be sufficient for further experiments.

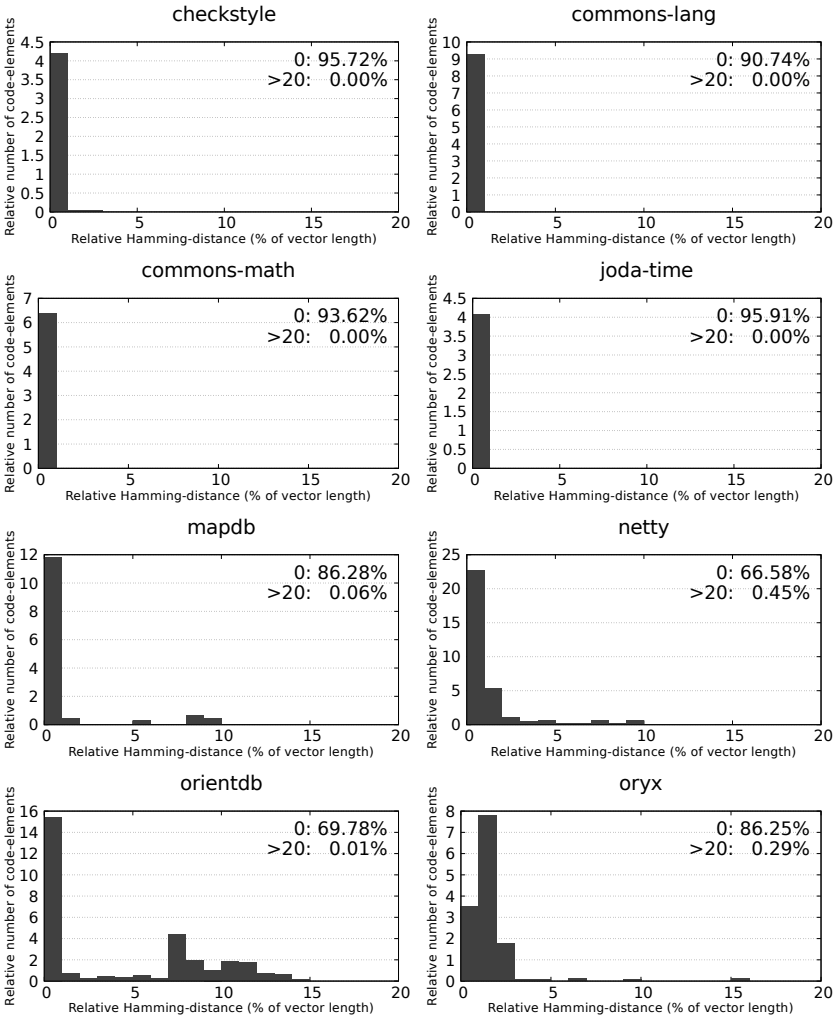


Fig. 8: Relative Hamming distances of code element vectors (JaCoCo vs. JCov)