Is Bytecode Instrumentation as Good as Source Code Instrumentation: An Empirical Study with Industrial Tools

(Experience Report)

Nan Li, Xin Meng, Jeff Offutt, and Lin Deng
Software Engineering
George Mason University
nli1,xmeng,offutt,ldeng2@gmu.edu

***Abstract*—Branch coverage (BC) is a widely used test criterion that is supported by many tools. Although textbooks and the research literature agree on a standard definition for BC, tools measure BC in different ways. The general strategy is to "instrument" the program by adding statements that count how many times each branch is taken. But the details for how this is done can influence the measurement for whether a set of tests have satisfied BC. For example, the standard definition is based on program source, yet some tools instrument the bytecode to reduce computation cost. A crucial question for the validity of these tools is whether bytecode instrumentation gives results that are the same as, or at least comparable to, source code instrumentation. An answer to this question will help testers decide which tool to use.**

This research looked at 31 code coverage tools, finding four that support branch coverage. We chose one tool that instruments the bytecode and two that instrument the source. We acquired tests for 105 methods to discover how these three tools measure branch coverage. We then compared coverage on 64 methods, finding that the byte-code instrumentation method reports the same coverage on 49 and lower coverage on 11. We also found that each tool defined branch coverage differently, and what is called branch coverage in the bytecode instrumentation tool actually matches the standard definition for clause coverage.

I. Introduction

A primary purpose of testing is to discover software failures early so that faults may be identified and corrected before the software is released to customers. Repairing faults early is more efficient and has the potential for significant savings [27]. A crucial step is to design tests that are effective at discovering existing failures in efficient ways. The only way to be sure we have detected all faults is to use all inputs, which is prohibitively expensive. Thus test design can be viewed as selecting inputs to satisfy three goals: (1) a small set of inputs (since each test adds to the ongoing cost of testing); (2) using an efficient process, that is, the test design process is as inexpensive as practical; and (3) the tests are effective at finding as many of the faults as possible. Of course, these goals cannot be satisfied completely, and are in some sense contradictory. Thus testers must make engineering compromises.

One sound way to balance these goals is to use coverage criteria. A *coverage criterion* is a rule or set of rules that are applied to a software artifact (such as the source code, requirements, or a design document) to create a collection of *test requirements* [2]. Each test requirement defines a specific element in the software artifact that must be covered or satisfied by a test. Many coverage criteria have been defined, including cover every statement, take every branch, evaluate logic predicates in certain ways, and test every requirement. This paper focuses on branch coverage on source.

Test criteria can be used as "goals" that tests must be designed to satisfy, or as "metrics" to evaluate tests. Tests are commonly measured by instrumenting the program to count how many times the tests reach the statement or branch that is to be covered. For example, branch coverage is measured by placing a counter on each branch in a program.

The definitions and most of our knowledge about code coverage criteria are based on covering the source code; however some Java code coverage tools instrument bytecode. Instrumenting bytecode is easier than instrumenting source, but leaves the question of whether it matches the standard definitions. Thus, an important question for industrial testers who use coverage tools is whether achieving branch coverage at the bytecode level is the same as achieving branch coverage on the source code. If they are not the same, are the coverage scores from bytecode instrumentation consistently lower than source code instrumentation, consistently higher than source code instrumentation, or inconsistent with source code instrumentation?

The goal of this research is to understand how code coverage tools measure branch coverage and whether bytecode instrumentation gets the same branch coverage as source code instrumentation. Specifically, we address three research questions:

- RQ1: What language control structures do the tools consider when measuring branch coverage?
- RQ2: Do the tools' measurements accurately reflect the theory that branch coverage subsumes statement coverage?
- RQ3: Does bytecode instrumentation give the same branch coverage results as source code instrumentation?

Answers to these questions can be helpful in three ways. Industrial testers need consistency from tools, and this study shows how and when the tools behave consistently. Second, users need these results to understand what the results from the tools mean. Third, tool builders need to understand how to properly design and build coverage analysis tools.

The contents of the paper are as follows. Section II provides background and section III discusses related work. Section IV presents an analysis of how specific tools perform instrumentation and measure coverage, and section V empirically compares bytecode with source code instrumentation. Section VI presents possible threats to validity in our studies, and section VII presents conclusions and discusses future work.

II. Background

This section first provides background on coverage criteria in general, and branch coverage in particular. Then the most common technique to measure coverage, instrumentation, is described.

A. Code Coverage Criteria

A coverage criterion is called "code coverage" when it is applied to source code. The most commonly used coverage criterion is *node coverage*, where every node in a graph is covered. When the graph is a control flow graph where every node is a statement or basic block, this is called *statement coverage (SC)*.

A slightly more advanced criterion is **edge coverage**, where every edge in a graph is covered. When the graph is a control flow graph where edges represent control statements such as *if* and *while*, this is called *branch coverage (BC)*. If tests satisfy BC, they are guaranteed to cover every statement. Thus BC is said to *subsume* SC.

When the predicates on the branches have more than one clause ($a \wedge b \wedge c$), we sometimes want to test each clause separately. *Clause coverage* requires that each clause evaluate to both *true* and *false* [2]. Clause coverage is sometimes called *condition coverage*.

Other code coverage criteria include edge-pair, prime paths, active clause coverage, data flow, and all paths. These are not evaluated in this study.

B. Instrumenting to Measure Code Coverage

Code coverage tools measure test sets in terms of how close the tests come to satisfying all the test requirements. If the criterion is statement coverage, and the tests reach 85 out of 100 statements, a code coverage tool should report 85% coverage.

Most code coverage tools use a technique called *instrumentation*, where statements are added to the program under test to count coverage. In the example below, calls to the methods *trueReached()* and *falseReached()* in the statement coverage object *bc* are added to measure branch coverage. After executing the tests, *bc* would report to the tester how many times the true and false side of each branch was taken.

```
private void boolean check (int a, int b)
{
    if (a > b || b == 0) {
        bc.trueReached (1);
        return true;
    } else {
        bc.falseReached (1);
        return false;
    {
}
```

Coverage criteria are always defined on the source, and we think of instruments as being added to the source before compilation. Some tools implement code coverage by instrumenting the source, but others instrument the bytecode. Bytecode instrumentation has the advantages of not requiring the program to be recompiled after instruments are added, and being simpler to implement. The primary question of this research project is whether bytecode instrumentation is accurate enough for practical use.

III. Related Work

We have found few papers that evaluated coverage analysis tools. Both Yang et al. [35] and Shahid and Ibrahim [30] compared coverage-based testing tools from four perspectives: (1) programming languages supported, (2) instrumentation levels, (3) code coverage criteria, and (4) report formats. Yang et al. [35] compared 17 tools and Shahid and Ibrahim [30] compared 22. Code coverage tools are extremely volatile, and despite the papers being only two and five years old, some information is already out of date, for the following reasons:

1) Some tools are no longer available.
2) Some tools are not compatible with new programming language features and the latest versions of IDEs.
3) Some tools are not in the papers.
4) Some information about the tools is not accurate.

The comparisons in these studies were primarily based on documentation. Neither analyzed the instrumentation method or how the counting was done in detail. In our study we ran the tools on multiple programs to understand exactly how coverage is evaluated, and compared scores between bytecode and source code instrumentation.

IV. Analysis of Instrumentation Methods

The goal of this research is to understand how code coverage tools measure branch coverage and whether bytecode instrumentation gets the same branch coverage as source code instrumentation. This section addresses RQ1 and RQ2 from section I.

If a tool does not count branches for some statements or control structures (RQ1), then the tool can report that branch coverage is satisfied even when some statements have not been covered (RQ2). RQ3 took considerably more effort to answer, and is covered in the next section.

A. *Experimental Objects: Tools*

Our first step was to identify code coverage tools to examine. We needed tools that were free, actively being used, and that supported branch coverage. To maximize the chances for compatibility, we wanted them to use the same language, and Java has more free tools. We identified 31 candidate tools, as listed in table I.

In table I, Access indicates whether the tool is free or whether there is a charge for it. All tools that charged for use offered free trials to students except *AgitarOne*, *Jtest*, and *VectorCAST*. The websites for *eXVantage* and *Koalog* are no longer available. We defined Active to be whether the tool's website has been updated in the past four years. Inactive tools may no longer work with new features in the language or the current IDEs.

We used the tool's published documentation (usually on a website) to determine whether it supported BC. We were not able to decide for three tools because the documentation was unclear or ambiguous. The last column, Method, indicates whether the tool used source code instrumentation, bytecode instrumentation or both, if known. Question marks indicate the documentation was unclear or ambiguous. Some tools support more than one programming language and we only considered the Java version.

We identified four tools that are active and support branch coverage: *Clover*, *ECobertura*, *CodeCover*, and *EclEmma*. They are highlighted in bold in table I. *EclEmma* is based on the *JaCoCo* Java code coverage library, which was created by the *EclEmma* team. *ECobertura* is an Eclipse plug-in for *Cobertura*. *Clover* and *CodeCover* use source code instrumentation and *ECobertura* and *EclEmma* use bytecode instrumentation. The only other tools were JCover, which has not been updated in four years, and three tools that do not allow free trials.

Our initial use of *ECobertura* showed that it reports branch coverage scores for classes, but not individual methods, so we elected to use *Clover*, *CodeCover*, and *EclEmma*.

B. *Experimental Subjects: Programs*

Next we chose software on which to evaluate the tools. We decided to use the open source FindBugs, version 2.0.1 [25], which comes with JUnit tests. The tests are associated with individual methods through the naming convention of using the method name plus "Test." We chose 19 out of 29 classes from 10 packages as experimental subjects. Although we did not use all the classes, we used all the tests for all the classes in the same package, so we had 29 JUnit tests.

EclEmma, *CodeCover*, and *Clover* provide branch coverage scores for each method. *CodeCover* also gives clause coverage scores for each method. The coverage results are displayed as percentages, and *Clover* also states how many branches are present and covered. The coverage results do not tell us how the tools measure branches.

The tools use colors to mark if a piece of code is executed, partially executed, or not executed at all. *EclEmma* adds small pointers to the source code for each predicate and structure that it measures. If a user hovers the mouse over a pointer, it shows the numbers of total and missed branches for a predicate or structure. *CodeCover* shows the true/false values for all predicates and their clauses.

C. *Control Structures Considered (RQ1)*

The websites and documentation for the tools do not provide complete information about how they count branches. For instance, it is unclear whether the tools measure predicates in ternary operators (with three operands, that is, the "?:" operator) or count branches for *switch* statements. There is information that the tools do not measure all branching statements. For example, the documentation for *CodeCover* says "The branch coverage, that is measured for Java, does not consider the body of a looping statement as a branch." Any compiler or testing textbook will consider the loop body to be in a branch that is reached when the loop predicate evaluates true, so this omission is surprising.

To determine what control structures are measured, we hand counted the number of branches in each method, analyzed which branches should be executed by the tests, and then checked to see if the ratio matched the branch coverage scores from the tools.

All tools had one omission. If a method had no branches, for example a simple getter or setter, then the tools report 0% branch coverage even if the method was executed. That is, the tools do not consider straight line code with no branches.

EclEmma measures predicates in almost all structures, including decisions in assignment statements and

Name	**Access**	**Active**	**Supports BC**	**Method**
AgitarOne [1]	charge	Y	?	?
Clover [3]	charge	Y	Y	source
CodeCover [29]	free	Y	Y	source
CodePro Analytix [23]	free	Y		bytecode
Coverlipse [13]	free	Y		bytecode
EclEmma [11]	free	Y	Y	bytecode
ECobertura [10]	free	Y	Y	bytecode
Emma [26]	free	Y		bytecode
eXVantage	charge			
Gretel [21]	free			bytecode
GroboCoverage [24]	free			bytecode
Hansel [17]	free			bytecode
Java Code Coverage Analyzer-JCover [33]	charge		Y	both
JavaCodeCoverage [14]	free			bytecode
JavaCov [15]	charge		Y	source
Jazz [18]	free			?
JBlanket [4]	free			bytecode
JCoverage [16]	free			?
Jester [19]	free			?
JFeature [34]	charge			?
Jtest [22]	charge	Y	?	?
JVMDI [5]	free			?
Koalog	charge			?
NetBean IDE Code Coverage Plugin [20]	free	Y		bytecode
NoUnit [8]	free			bytecode
Open Code Coverage Framework [28]	free			?
PurifyPlus [12]	charge	Y		bytecode
Quilt [9]	free			bytecode
SD Test Coverage tools [6]	charge	Y		source
TestWorks [32]	charge			?
VectorCAST [31]	charge	Y	?	source

TABLE I. CODE MEASUREMENT TOOLS

method calls. For instance, if the parameter to a method call included a predicate (such as "*m.setCouldBeZero (i1.isCouldBeZero()* || *i2.isCouldBeZero());*"), *EclEmma* would measure the predicate. However *EclEmma* does not count *try-catch* statements, as in the *catch* block in the example below.

try { ... }
catch (Exception e) { ... }

The branch coverage criterion of *CodeCover* considers boolean conditions only in *if-else*, *try*, *catch*, and *finally* blocks and also counts branches for *switch* and *try-catch* statements (*finally* blocks are not counted). The clause coverage criterion of *CodeCover* measures predicates in *if-else*, *try*, *catch*, and *finally* blocks and all loop structures except enhanced *for* loops (looping over a collection or array without a loop index). Thus, the branch coverage criterion of *CodeCover* would count branches for the *catch* block in the example above. On the other hand, *CodeCover* does not count predicates in assignment statements, ternary operators, *return* statements, *assert* statements, or inside expressions.

Clover measures predicates in *if-else* blocks, traditional *for* loops, *assert* statements, *try* blocks, *catch* blocks, *finally* blocks and ternary operators. However, it does not measure branches for enhanced *for* loops or *switch* statements.

EclEmma instruments bytecode, so each clause in a predicate has two branches: true and false. Thus, in bytecode, the number of branches of a predicate is equal to two times the number of clauses. In a *switch* structure, branches are counted by the *break* statement, so code that ends with a *break* statement is counted as one branch. In the example below, two branches are counted because this *switch* structure has two *break* statements.

```
switch (variable) {
    case1: ...
    case2: ...
      break;
    default: ...
      break;
}
```

An *assert* statement is considered to have two branches even if it has no boolean expressions. So the statement "*assert a* $\geq$ *0;*" has four branches.

CodeCover and *Clover* instrument the source code, so each predicate has two branches: true and false. *CodeCover* has a different way to count branches for *switch* structure. Each case, including the default case, is considered to be a branch. Thus, *CodeCover* would count three branches for the *switch* example above. For *try-catch* statements, the *try block* is one branch and each *catch* block is one branch.

CodeCover's clause coverage criterion is very similar to *EclEmma*'s branch coverage, the only difference being that they measure different structures. For both tools, two branches are counted for each clause of a predicate: true and false.

In summary, if a method has only *if-else* blocks, the branch coverage criteria of *CodeCover* and *Clover* are the same. If a method has only *if-else* blocks and each predicate has only one clause, the branch coverage criteria of *CodeCover* or *Clover* and *EclEmma* are the same. If a method has only *if-else* blocks, *while* loops, *do-while* loops, traditional *for* loops, *assert* statements, *try* blocks, *catch* blocks, *finally* blocks and ternary operators, and each predicate has only one clause, the branch coverage criteria of *Clover* and *EclEmma* are the same. If a method has only *if-else* blocks, *while* loops, *do-while* loops, and traditional *for* loop structures, *EclEmma*'s branch coverage is equivalent to *CodeCover*'s clause coverage.

Table II summarizes the structures that are measured by each tool, answering RQ1. Note that *EclEmma* does not measure a predicate if its value can be determined statically.

D. Subsumption of SC by BC (RQ2)

If a branch coverage tool does not measure some language feature, that means it is possible to get 100% BC without reaching every statement. Thus, table II answers RQ2 directly. Since *EclEmma* does not measure *try-catch* statements, a test set can get 100% BC without ever reaching the statements in the *catch* blocks. Likewise, *CodeCover* does not measure blocks inside loops, and *Clover* does not measure blocks inside enhanced *for* loops. Thus, none of the three tools implement BC in such a way that it subsumes SC. This is problematic because testers assume if BC is achieved, they have tested every statement. This empirical analysis shows this is not true.

V. Comparison of Instrumentation Methods

This section presents an empirical comparison to evaluate whether source code and bytecode instrumentation gives the same results on the same tests. The previous section demonstrated that *EclEmma*, *Clover*, and *CodeCover* use different rules to measure branch coverage. To accurately compare bytecode instrumentation with source code instrumentation, we used programs that have only structures that are measured by both a bytecode instrumentation tool (*EclEmma*) and a source code instrumentation tool (*CodeCover* or *Clover*).

EclEmma and *CodeCover* both measure *if-else* blocks, *try* blocks, *catch* blocks, *finally* blocks and *switch* statements. They count *switch* differently and *CodeCover* counts *try* blocks and *catch* blocks as branches, so our comparison only uses methods that use *if-else* blocks.

EclEmma and *Clover* have more structures in common, including *while* loops, *do-while* loops, traditional *for* loops, *assert* statements, *try* blocks, *catch* blocks, *finally* blocks and ternary operators. *Clover* and *CodeCover* give exactly the same results for methods that have *if-else* blocks, *try* blocks, *catch* blocks, and *finally* blocks.

A. Test Preparation and Execution

We merged all JUnit tests for classes in the same package into a JUnit test suite. We ran the JUnit test suite against classes in the same package using *EclEmma*, *Clover*, and *CodeCover*. Then we recorded the branch coverage scores for all tools and the clause coverage scores from *CodeCover*.

B. Results

CodeCover and *Clover* measure different control structures, so we first show data comparing *EclEmma* (bytecode) and *CodeCover* (source) on methods that use only the control structures that both *EclEmma* and *CodeCover* measure, and then data comparing *EclEmma* and *Clover* (source) on methods that use only the control structures that both *EclEmma* and *Clover* measure. Table III shows branch coverage as measured by *EclEmma* and *CodeCover*, and condition coverage (clause coverage) measured by *CodeCover* for the 35 methods that have only *if-else* blocks.

The table shows the total number of branches (**B**), branches covered (**BC**), and branch coverage (**BC%**), for both *EclEmma* and *CodeCover*. Clause coverage is also shown for *CodeCover*. The *Total* row sums the total number of branches and covered branches for all subjects, and provides the overall branch coverage.

Table IV shows branch coverage of *EclEmma* and *Clover* for the 29 methods that have *if-else* blocks, *while* loops, *do while* loops, traditional *for* loops, *assert* statements, *try* blocks, *catch* blocks, *finally* blocks and ternary operators. The columns and Total row are the same as in table III.

Structures	EclEmma BC	CodeCover BC	CodeCover CC	Clover BC
if-else	Y	Y	Y	Y
while loop	Y		Y	Y
do-while loop	Y		Y	Y
for loop	Y		Y	Y
enhanced for loop	Y			
assignment expressions	Y			
ternary operators	Y			Y
return statements	Y			
assert statements	Y			Y
try blocks	Y	Y	Y	Y
catch blocks	Y	Y	Y	Y
finally blocks	Y	Y	Y	Y
normal statements	Y			
switch	Y	Y		
try-catch		Y		

TABLE II. CONTROL STRUCTURES THAT ARE CHECKED BY THE TOOLS.

In summary, the bytecode instrumentation tool *EclEmma* gets the same branch coverage scores as the source code instrumentation tool *CodeCover* or *Clover* on 49 methods and different results on 15 methods.

Because of the way *EclEmma* measures branch coverage at the bytecode level, we would expect it to be the same as *CodeCover*'s clause coverage. However, they are different on two methods, "setSpecialKindFromSignature" and "checkPluginRelease." The method "setSpecialKindFromSignature" has a piece of dead code that *CodeCover* measures but *EclEmma* does not. For method "checkPluginRelease," we believe *CodeCover* has a fault that causes it not to count some branches for clause coverage.

C. Discussion

Figures 1 and 2 compare the branch coverage results in bar charts. The horizontal lines represent the programs studied (35 for *CodeCover* and 29 for *Clover*). The vertical lines show the branch coverage (BC%).

Bytecode instrumentation (*EclEmma*) gets the same branch coverage as source instrumentation on 49 methods, lower branch coverage on 11 methods, and higher branch coverage on only 4 methods (program 24 for *CodeCover* and programs 10, 12, and 23 for *Clover*).

Since *EclEmma*'s branch coverage requires each clause to be true and false, it is equivalent to the standard definition of clause coverage. Branch coverage is the same as clause coverage on 49 of 64 methods, each of which can be attributed to one of three reasons: (1) The predicates have only one clause, (2) tests satisfy all branches of both the clauses and predicates, and (3) tests do not satisfy all branches of the clauses and predicates but the two coverage scores happen to be equal. Hand analysis showed that in 42 of the 49 methods, all predicates have only one clause. This agrees with our other paper, which found that approximately 90% of predicates "in the wild" only have one clause [7].

Clause coverage is generally harder to satisfy than branch coverage, so we expect *EclEmma* to report lower scores than *CodeCover* and *Clover*. This will not always be true because the results will depend on the specific values.

Consider the example predicate $P = a \ \& \ b$. With bytecode instrumentation, *EclEmma* counts four branches ($a = T$, $a = F$, $b = T$, and $b = F$). If a test case only has $a = F$, *EclEmma* reports 25% coverage, whereas *CodeCover* reports 50% coverage. But if two tests, ($a = F$, $b = F$) and ($a = T$, $b = F$), are used, *EclEmma* reports 75% coverage. Since P is never true, *CodeCover* only reports 50% coverage.

Based on these data, bytecode instrumentation tools are more likely to report lower branch coverage scores, and the tester will need more tests to achieve the same coverage score.

Table V shows the methods for which the bytecode tool (*EclEmma*) reported lower scores than the source code tools (*CodeCover* and *Clover*). The % Diff column subtracts the bytecode percentage from the source code percentage. Table VI shows the opposite, the methods for which the bytecode tool reported higher scores. (The scores were the same for the other 49 methods.) These tables only show the methods that are directly comparable among all three tools. Table V has 11 methods to only four in table VI. Moreover, the mean and median are much higher in table V. This indicates when the branch coverage scores reported by a bytecode instrumentation tool are higher, the difference tends to be small.

VI. THREATS TO VALIDITY

As in most software engineering studies, we cannot be sure that the subject programs are representative. Another threat to external validity is that we used

Class	Method	EclEmma			CodeCover			
		B	BC	BC%	B	BC	BC%	CC%
BugInstance	BugInstance	8	5	62.50	6	5	83.30	62.50
	BugProperty.next	2	2	100.00	2	2	100.00	100.00
	BugProperty.remove	8	8	100.00	6	6	100.00	100.00
	Add	2	1	50.00	2	1	50.00	50.00
	addProperty	2	2	100.00	2	2	100.00	100.00
	ageInDays	2	1	50.00	2	1	50.00	50.00
	getBugPattern	4	3	75.00	4	3	75.00	75.00
	getPrimaryClass	2	1	50.00	2	1	50.00	50.00
	setProperty	2	1	50.00	2	1	50.00	50.00
ClassScreener	addAllowedClass	2	1	50.00	2	1	50.00	50.00
	addAllowedPackage	4	3	75.00	4	3	75.00	75.00
IntAnnotation	getShortInteger (long)	6	3	50.00	2	1	50.00	50.00
	getShortInteger (int)	6	5	83.33	2	2	100.00	83.33
Obfuscate	hashFieldSignature	2	2	100.00	2	2	100.00	100.00
OpcodeStack	item	16	5	31.20	12	5	41.70	31.20
	setSpecialKindFrom Signature	4	2	50.00	4	2	50.00	33.33
	setFlag	2	1	50.00	2	1	50.00	50.00
SAXBugCollectionHandler	getRequiredAttribute	2	1	50.00	2	1	50.00	50.00
	memorized	4	4	100.00	4	4	100.00	100.00
	setAnnotationRole	2	1	50.00	2	1	50.00	50.00
MethodHash	toUnsigned	2	2	100.00	2	2	100.00	100.00
GenericObjectType	GenericObjectType	4	2	50.00	2	1	50.00	50.00
	getParameterAt	2	1	50.00	2	1	50.00	50.00
	getTypeCategory	24	3	12.50	12	1	08.33	12.50
IsNullValue	IsNullValue	2	1	50.00	2	1	50.00	50.00
	equals	12	5	41.67	8	4	50.00	41.67
	toExceptionValue	2	1	50.00	2	1	50.00	50.00
ReturnPathType	mergeWith	6	6	100.00	6	6	100.00	100.00
ProjectFilterSettings	equals	12	9	75.00	10	8	80.00	75.00
	setMinPriority	4	1	25.00	4	1	25.00	25.00
UpdateChecker	checkPluginRelease	8	5	62.50	4	3	75.00	37.50
	getMajorJavaVersion	2	1	50.00	2	1	50.00	50.00
JAIFScanner	fillLineBuf	4	3	75.00	4	3	75.00	75.00
	nextToken	2	1	50.00	2	1	50.00	50.00
AbstractCloud	AbstractCloud	2	1	50.00	2	1	50.00	50.00
Total		**170**	**94**	**55.29**	**130**	**80**	**61.53**	

TABLE III. COVERAGE MEASUREMENTS FOR ECLEMMA AND CODECOVER–B IS TOTAL BRANCHES, BC IS BRANCHES COVERED, BC% IS PERCENT COVERED, CC% IS PERCENT CLAUSES COVERED.

one bytecode instrumentation tool and two source code instrumentation tools. Other implementations might produce different results. Furthermore, the tools may have implementation faults. We found one fault for the clause coverage criterion of *CodeCover*; it does not count branches for some clauses. For instance, in the method checkPluginRelease of the class UpdateChecker, *CodeCover* shows the test values for two branches in a predicate, but these two branches were not counted. An internal threat is that the first author analyzed by hand the tools to determine how they measure branches.

VII. CONCLUSIONS, FINDINGS, AND FUTURE WORK

This paper presents results on the validity of using bytecode instrumentation to measure branch coverage. We started with 31 Java code coverage tools and found four that are active and support branch coverage. We gathered our results from one bytecode instrumentation tool, *EclEmma*, and two source code instrumentation tools, *CodeCover* and *Clover*.

We first analyzed these tools to understand exactly how they measure branch coverage. We studied 105 methods from the open source project FindBugs. None of the tools measured all control structures and the only structure all tools measure are simple *if-else* blocks. This means that satisfying branch coverage does not guarantee that all executable statements were covered. That is, according to these tools, branch coverage does **not** subsume statement coverage.

Our analysis also showed that although the byte-

Class	Method	EclEmma			Clover		
		B	**BC**	**BC%**	**B**	**BC**	**BC%**
BugInstance	getInstanceHash	2	1	50.00	2	1	50.00
	getXmlProps	4	4	100.00	4	4	100.00
	lookupProperty	4	3	75.00	4	3	75.00
IntAnnotation	uniqueDigits	2	2	100.00	2	2	100.00
	getShortInteger (int)	6	5	83.33	2	2	100.00
Obfuscate	hasMethodSignature	2	2	100.00	2	2	100.00
SAXBugCollectionHandler	createSource LineAnnotation	12	10	83.33	12	10	83.33
	endElement	60	21	35.00	54	20	37.00
	parseBug InstanceContents	42	11	26.19	38	10	26.30
	parseLong	2	1	50.00	2	1	50.00
	startElement	130	63	48.46	108	52	48.10
MethodHash	compareHashes	4	4	100.00	4	4	100.00
SignatureParser	SignatureParser	10	9	90.00	8	7	87.50
	getNumParameter	2	1	50.00	2	1	50.00
GenericObjectType	createInstanceByFlagList	2	2	100.00	2	2	100.00
IsNullValue	merge	24	15	62.50	14	9	64.30
ReturnPathType	setCanReturnNormally	2	2	100.00	2	2	100.00
ProjectFilterSettings	fromEncodedString	22	12	54.55	22	12	54.55
	hiddenFromEncodedString	6	4	66.67	6	4	66.67
	hiddenToEncodedString	4	4	100.00	4	4	100.00
	toEncodedString	6	6	100.00	6	6	100.00
ClassName	extractClassName	12	10	83.33	8	7	87.50
	extractPackage Prefix	6	6	100.00	6	6	100.00
UpdateChecker	getPluginThatDisabled UpdateChecks	8	7	87.50	6	5	83.33
	getRedirectURL	8	5	62.50	6	4	66.67
	getUuid	2	1	50.00	2	1	50.00
	parseReleaseDate	2	1	50.00	2	1	50.00
	startUpdateCheckThread	10	3	30.00	4	2	50.00
AbstractCloud	getSourceLink	6	6	100.00	6	6	100.00
	printLeaderBoard	4	4	100.00	4	4	100.00
Total		**394**	**215**	**54.56**	**342**	**192**	**56.14**

TABLE IV. Branch coverage measurement for EclEmma and Clover–B is total branches, BC is branches covered, BC% is percent covered.

code instrumentation tool claimed to evaluate branch coverage, it was actually measuring a different criterion, clause coverage. This is particularly confusing, because although clause coverage usually requires more tests, it does not necessarily subsume statement coverage. Consider the predicate $P = a \ \& \ b$. The tests $(a = t, b = f)$ and $(a = f, b = t)$ satisfy clause coverage, but P evaluates to false for both tests, thus they do not satisfy branch coverage.

Next we compared bytecode and source code instrumentation on 64 methods, finding that both instrumentation methods had the same scores for 49 methods (77%), the bytecode instrumentation tool had a lower score on 11 methods, and the source code instrumentation tool had a lower score on four. We conclude that bytecode instrumentation, as implemented in the tool we evaluated, is **not** a valid way to measure branch coverage.

To summarize, we have the following four major findings:

1) None of the tools studied evaluate all the branching structures in Java.
2) Branch coverage does not subsume statement coverage for any of the tools studied. That is, branch coverage is implemented incorrectly.
3) Branch coverage using bytecode instrumentation is equivalent to clause coverage using source code instrumentation, not branch coverage.
4) Bytecode instrumentation is not a valid technique to measure branch coverage.

We are considering three future directions. The first is to perform a similar study for statement coverage. More tools can be considered, since more tools support statement coverage. The second is to extend the study to more tools. A third is to try to measure which tool

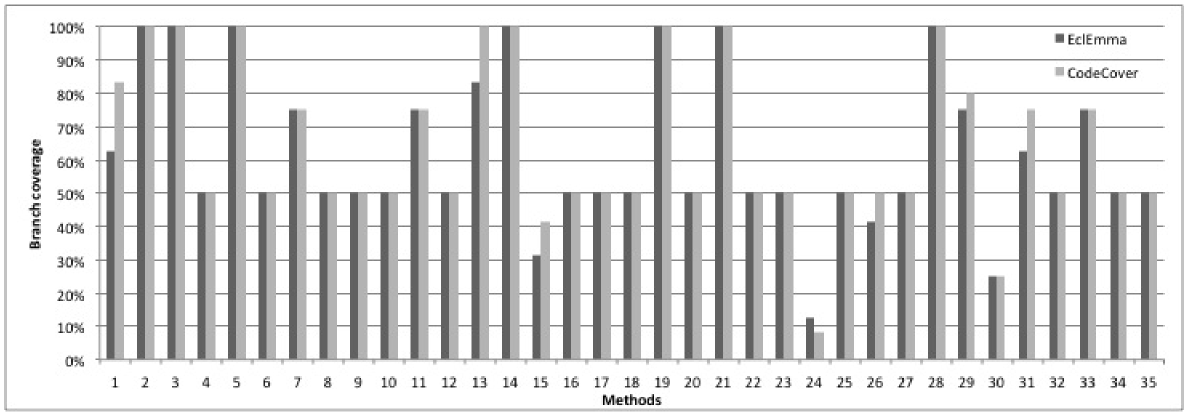

Fig. 1. A comparison of branch coverage between *EclEmma* and *CodeCover*

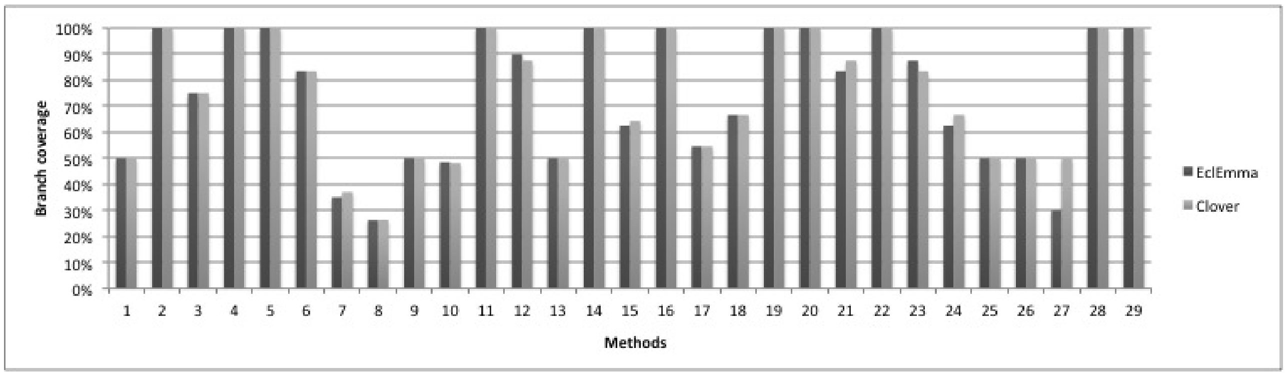

Fig. 2. A comparison of branch coverage between *EclEmma* and *Clover*

will help testers design better quality tests.

REFERENCES

[1] Agitar. AgitarOne functional coverage tracker. Online, 2013. http://www.agitar.com/solutions/products/functional_coverage_tracker.html, last access Feb 2013.

[2] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, Cambridge, UK, 2008. ISBN 0-52188-038-1.

[3] Atlassian. Clover. Online, 2013. http://www.atlassian.com/software/clover/, last access Feb 2013.

[4] Joy Augustin. Jblanket. Online, 2006. http://csdl.ics.hawaii.edu/research/jblanket/, last access Feb 2013.

[5] Joel Crisp. Jvmdi code coverage analyzer for Java. Online, 2006. http://jvmdicover.sourceforge.net/, last access Feb 2013.

[6] Semantic Designs. SD Test Coverage tools. Online, 2013. http://www.semdesigns.com/Products/TestCoverage/, last access May 2013.

[7] Vinicius H. Durelli, Jeff Offutt, Nan Li, and Marcio Delamaro. Predicates in the wild: Gauging the cost-effectiveness of applying active clause coverage criteria to open-source Java programs. Submitted for publication, 2013.

[8] FirstPartners. Nounit. Online, 2006. http://nounit.sourceforge.net/faq.html, last access Feb 2013.

[9] Apache Software Foundation. Quilt. Online, 2003. http://quilt.sourceforge.net/index.html, last access Feb 2013.

[10] Joachim Hofer. ECobertura. Online, 2011. http://ecobertura.johoop.de/, last access Feb 2013.

[11] Marc R. Hoffmann, Brock Janiczak, and Evgeny Mandrikov. Eclemma-Java code coverage tool for eclipse. Online, 2006. http://www.eclemma.org/, last access May 2013.

[12] IBM. Purifyplus. Online, 2013. http://www.ibm.com/developerworks/rational/products/purifyplus/, last access May 2013.

[13] Matthias Kempka. Coverlipse. Online, 2009. http://coverlipse.sourceforge.net/, last access Feb 2013.

[14] Raghu Lingampally, Atul Gupta, and Pankaj Jalote. A multi-purpose code coverage tool for Java. In *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, HICSS '07, page 261b. IEEE Computer Society, 2007.

[15] Alvicom Ltd. Alvicom JavaCov. Online, 2013. http://www.alvicom.hu/eng/javacov.php, last access Feb 2013.

[16] JCoverage Ltd. Jcoverage. Online, 2013. http://www.jcoverage.com/, last access May 2013.

[17] Niklas Mehner. Hansel. Online, 2006. http://hansel.sourceforge.net/, last access Feb 2013.

[18] Jonathan Misurda, Jim Clause, Juliya Reed, Bruce R. Childers, and Mary Lou Soffa. Jazz: A tool for demand-driven structural testing. In *Proceedings of the 14th international conference on Compiler Construction*, CC '05, pages 242–245, Edinburgh, UK, 2005. Springer-Verlag.

Method	**ByteCode BC%**	**Source Code BC%**	**% Diff**
getShortInteger	83.33	100.00	16.67
Item	31.20	41.70	10.50
endElement	35.00	37.00	2.00
parseBug InstanceContents	26.19	26.30	0.11
equals	41.67	50.00	8.33
merge	62.50	64.30	1.80
equals	75.00	80.00	5.00
extractClass Name	83.33	87.50	4.17
heckPlugin Release	62.50	75.00	12.50
getRedirect URL	62.50	66.70	4.20
startUpdate CheckThread	30.00	50.00	20.00
Mean			7.75
Median			5.00
Standard Deviation			6.48

TABLE V. METHODS FOR WHICH BYTECODE INSTRUMENTATION REPORTS LOWER SCORES THAN SOURCE CODE INSTRUMENTATION

Method	**ByteCode BC%**	**Source Code BC%**	**% Diff**
startElement	48.46	48.10	0.36
Signature Parser	90.00	87.50	2.50
getType Category	12.50	8.30	4.20
getPluginThat DisabledUpdateChecks	87.50	83.30	4.20
Mean			2.82
Median			3.35
Standard Deviation			1.82

TABLE VI. METHODS FOR WHICH BYTECODE INSTRUMENTATION REPORTS HIGHER SCORES THAN SOURCE CODE INSTRUMENTATION

[19] Ivan Moore. Jester - The JUnit test tester. Online, 2005. http://jester.sourceforge.net/, last access Feb 2013.

[20] NetBean. Netbean code coverage plugin. Online, 2010. http://plugins.netbeans.org/plugin/38945/unit-tests-code-coverage-plugin-updated-for-netbeans-7-0, last access March 2013.

[21] University of Oregon. Gretel. Online, 2002. http://www.cs.uoregon.edu/Research/perpetual/dasada/Software/Gretel/, last access Feb 2013.

[22] Parasoft. Jtest. Online, 2013. http://www.parasoft.com/jsp/products/jtest.jsp, last access Feb 2013.

[23] Eclipse Project and Inc. Google. CodePro Analytix. Online, 2001. https://developers.google.com/java-dev-tools/codepro/doc/, last access March 2013.

[24] GroboUtils Project. Grobocoverage. Online, 2004. http://groboutils.sourceforge.net/codecoverage/, last access Feb 2013.

[25] Bill Pugh, Andrey Loskutov, and Keith Lea. Findbugs. Online, 2003. http://findbugs.sourceforge.net, last access May 2013.

[26] Vlad Roubtsov. Emma. Online, 2006. http://emma.sourceforge.net/, last access Feb 2013.

[27] Research Triangle Institute (RTI). The economic impacts of inadequate infrastructure for software testing. Technical report 7007.011, NIST, May 2002. http://www.nist.gov/director/prog-ofc/report02-3.pdf.

[28] Kazunori Sakamoto, Kiyofumi Shimojo, and Ryohei Takasawa. Open Code Coverage Framework. Online, 2013. http://code.google.com/p/open-code-coverage-framework/, last access Feb 2013.

[29] Rainer Schmidberger. Codecover. Online, 2011. http://codecover.org/, last access Feb 2013.

[30] Muhammad Shahid and Suhaimi Ibrahim. An evaluation of test coverage tools in software testing. In *2011 International Conference on Telecommunication Technology and Applications Proc. of CSIT*, volume 5. IACSIT Press, 2011.

[31] Vector Software. VectorCAST Code Coverage. Online, 2013. http://www.vectorcast.com/testing-solutions/code-coverage-analysis.php, last access Feb 2013.

[32] Inc. Software Research. Testworks. Online, 2006. http://www.testworks.com/, last access Feb 2013.

[33] Man Machine Systems. Java code coverage analyzer. Online, 2009. http://www.mmsindia.com/JCover.html, last access Feb 2013.

[34] Technobuff. Jfeature. Online, 2005. http://www.technobuff.net/webapp/product/showProduct.do?name=jfeature, last access Feb 2013.

[35] Qian Yang, J. Jenny Li, and David Weiss. A survey of coverage based testing tools. *The Computer Journal*, 52(5):589–597, March 2007.