Assessing Oracle Quality with Checked Coverage

David Schuler
Saarland University – Computer Science
Saarbrücken, Germany
ds@cs.uni-saarland.de

Andreas Zeller
Saarland University – Computer Science
Saarbrücken, Germany
zeller@cs.uni-saarland.de

***Abstract*—A known problem of traditional coverage metrics is that they do not assess *oracle quality*—that is, whether the computation result is actually checked against expectations. In this paper, we introduce the concept of *checked coverage*—the dynamic slice of covered statements that actually influence an oracle. Our experiments on seven open-source projects show that checked coverage is a sure indicator for oracle quality—and even more sensitive than mutation testing, its much more demanding alternative.**

***Keywords*-test suite quality, coverage metrics, dynamic slicing, mutation testing**

I. Introduction

How can I ensure that my program is well-tested? The most widespread metric to assess test quality is *coverage*. Test coverage measures the percentage of code features such as statements or branches that are executed during a test. The rationale is that the higher the coverage, the higher the chances of catching a code feature that causes a failure—a rationale that is easy to explain, but which relies on an important assumption. This assumption is that we actually are able to *detect* the failure. It does not suffice to cover the error, we also need a means to detect it.

As it comes to detecting arbitrary errors, it does not make a difference whether an error is detected by the test (e.g., a mismatch between actual and expected result), by the program itself (e.g., a failing assertion), or by the run-time system (e.g., a dereferenced null pointer). To validate computation results, though, we need checks in the test code and in the program code—checks henceforth summarized as *oracles*. A high coverage does not tell anything about oracle quality. It is perfectly possible to achieve a 100% coverage and still not have any result checked by even a single oracle. The fact that the run-time system did not discover any errors indicates robustness, but does not tell anything about functional properties.

As an example of such a mismatch between coverage and oracle quality, consider the test for the `PatternParser` class from the JAXEN XPATH library, shown in Figure 1. This test invokes the parser for a number of paths and achieves a statement coverage of 83% in the parser. However, none of the parsed results is actually checked for any property. The parser may return utter garbage, and this test would never notice it.

```
public void testValidPaths() throws
  JaxenException, SAXPathException {
  for (int i = 0; i < paths.length; i++) {
    String path = paths[i];
    Pattern p = PatternParser.parse(path);
  }
}
```

Figure 1. A test without outcome checks.

To assess oracle quality, a frequently proposed approach is *mutation testing*. Mutation testing seeds artificial errors (*mutants*) into the code and assesses whether the test suite finds them. A low score of detected mutants implies low coverage (i.e., the mutant was not executed) or low oracle quality (i.e., its effects were not checked). Unfortunately, mutation testing is costly; even with recent advancements, there still is manual work involved to weed out equivalent mutants.

In this paper, we introduce an alternative, cost-efficient way to assess oracle quality. Using dynamic slicing, we determine the *checked coverage*—statements that were not only executed, but that actually contribute to the results checked by oracles. In Figure 1, the checked coverage is 0%, because none of the results ever flows into a run-time check (again, in contrast to the 83% traditional coverage). However, adding a simple assertion that the result is non-null already increases the checked coverage to 65%; adding further assertions on the properties of the result further increases the checked coverage.

Using checked coverage as a metric rather than regular coverage brings significant advantages:

- Few or insufficient oracles immediately result in a low checked coverage, giving a more realistic assessment of test quality.
- Statements that are executed, but whose outcomes are never checked would be considered *uncovered.* As a consequence, one would improve the oracles to actually check these outcomes.
- Rather than focusing on executing as much code as possible, developers would focus on checking as many results as possible, catching more errors in the process.
- To compute checked coverage, one only needs to run

the test suite once with a constant overhead—which is way more efficient than the potentially unlimited number of executions induced by mutation testing.

This paper introduces the concept of checked coverage (Section II) and evaluates the concept on seven open-source projects (Section III). Our results show that checked coverage is a sure indicator for oracle quality—and even more sensitive than mutation testing, its much more demanding alternative. After discussing the limitations (Section IV), the threats to validity (Section V), and the related work (Section VI), we close with conclusion and consequences (Section VII).

II. Checked Coverage

Our concept of checked coverage is remarkably simple: Rather than computing *coverage*—the extent to which code features are *executed* in a program—, we focus on those code features that actually contribute to the results checked by oracles. For this purpose, we compute the *dynamic backward slice* of test oracles—that is, all statements that contribute to the checked result. This slice then constitutes the *checked coverage.*

A. Program Slicing

Program slicing was introduced by Weiser [1], [2] as a technique that determines the set of statements that potentially influence the variables used in a given location. Weiser claims that this technique corresponds to the mental abstractions programmers are making when they debug a program. Korel and Laski [3] refined this concept and introduced *dynamic slicing*. In contrast to the static slice as proposed by Weiser, the dynamic slice only consists of the statements that *actually influenced* the variables used in a specific occurrence of a statement in a specific program run.

A static slice is computed from a *slicing criterion* (s, V), where s is a statement, and V is a subset of the variables used in the program. A slice for a given slicing criterion is computed by transitively following *all data and control dependencies* for the variables V used in s, where data and control dependencies are defined as follows:

Definition 1 (Data Dependency)**.** A statement s is *data dependent* on a statement t iff there is a variable v that is defined (written) in t and referred to (read) in s, and there is at least one execution path from t to s without a redefinition of v.

Definition 2 (Control Dependency)**.** A statement s is *control dependent* on a statement t iff t is a conditional statement and the execution of s depends on t.

A *dynamic slicing criterion* (s_o, V, I), in addition, specifies the input I to the program, and distinguishes between different occurrences o of a statement s. A dynamic slice for a criterion is then computed from a trace (trajectory) of a program run that uses I as input. The transitive closure over all dependencies of all variables V that are used in the oth occurrence of statement s forms the dynamic slice.

In contrast to dynamic slices, static slices take all possible dependencies into account. Therefore, static slices tend to be much bigger than the dynamic slices, where all actual dependencies are known. As we use the slices to measure the quality of a test suite, which represents an execution of a program, we choose dynamic slices for our approach. Furthermore, we consider all occurrences of a statement for a dynamic slice. This is done by computing the union of all dynamic slices for every occurrence of a statement.

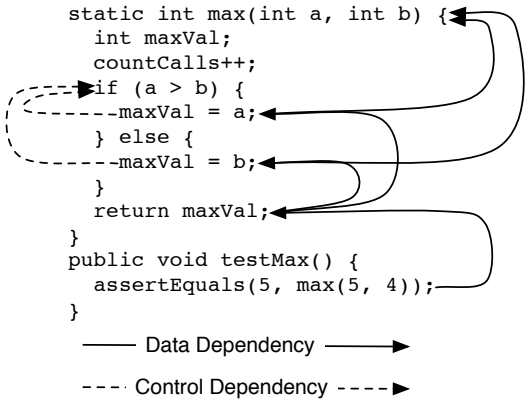

Figure 2. Data and control dependencies for a method and its test.

The code shown in Figure 2, for example, has a method that computes the maximum for two integers, and a test that checks the result for one input. Solid arrows show the data dependencies and dashed arrows the control dependencies. The static slice from the assert-statement consists of all statements in the `max()` method except for the increment of `countCalls`, as there is neither a control nor a data dependency from the variables used in the assert-statement to `countCalls`. The dynamic slice additionally excludes the else part in the `max()` method, since this code is not exercised by the test, and consequently these control and data dependencies are not present in the trace.

B. From Slices To Checked Coverage

For the checked coverage, we are interested in the *ratio of statements* that contribute to the computation of values that get later checked by the test suite. To this end, we first identify all statements that check the computation inside the test suite. Then we trace one run of the test suite, and compute the dynamic slice for all these statements. This gives us the set of statements that have a control or data dependency to at least one of the check statements. The checked coverage is then the percentage of the statements in the set relative to all coverable statements.

Table I
DESCRIPTION OF SUBJECT PROGRAMS.

Project Name	Description	Version	Program size (LOC)	Test code size (LOC)	Number of tests	Test suite runtime
ASPECTJ	AOP extension to Java	cvs: 2010-09-15	156,780	32,942	339	11s
BARBECUE	Bar code creator	svn: 2007-11-26	4,837	3,293	153	3s
COMMONS	Helper utilities	svn: 2010-08-30	18,452	29,699	1,787	33s
JAXEN	XPath engine	svn: 2010-06-07	12,438	8,418	689	11s
JODA-TIME	Date and time library	svn: 2010-08-25	26,582	50,738	2,734	37s
JTOPAS	Parser tools	1.0(SIR)	2,031	3,185	128	2s
XSTREAM	XML object serialization	svn: 2010-04-17	15,266	16,375	1113	7s

Lines of Code (LOC) are non-comment, non-blank lines as reported by `sloccount`.
For ASPECTJ, we only considered the *org.aspectj.ajdt.core* package, which has 28,476 lines of source code and 6,830 lines of test code.

C. Implementation

For the implementation of our approach, we use Hammacher's JAVASLICER [4] as a dynamic slicer for Java. The slicer works in two phases: In the first phase, it traces a program and produces a *trace file*, and in the second one, it computes the slice from this trace file.

The tracer manipulates the java bytecode of a program by inserting special tracing statements. At run time, these inserted statements log all definitions and references of a variable, and the control flow jumps that are made. By using the Java agent mechanism, all classes that are loaded by the Java Virtual Machine (JVM) can be instrumented. (There are a few exceptions though that are explained in Section IV.) Since Java is an object-oriented language, the same variable might be bound to different objects. The tracer however needs to distinguish these objects. For that purpose, the slicer assigns each object a unique identifier. The logged information is directly written to a stream that compresses the trace data using the SEQUITUR [5], [6] algorithm and stores it to disk.

To compute the slices, an adapted algorithm from Wang and Roychoudhury [7], [8] is used. The data dependencies are calculated by iterating backwards through the trace. For the control dependencies, the control flow graph (CFG) for each method is built, and a mapping between a conditional statement and all statements it controls is stored. With this mapping, all the control dependent statements for a specific occurrence of a statement can be found.

Our implementation computes the checked coverage for JUnit test suites, and works in 3 steps:

1) First all checks and all coverable statements are identified. We use the heuristic that all calls to a JUnit assert-method from the test suite is considered as a check. As coverable lines, we consider all lines that are handled by the tracer. Note that this excludes some statements such as `try`, or simple `return` statements, as they do not translate to data or control flow manipulating statements in the byte code.
2) Second all test classes are traced separately. This is a performance optimization, since we observed that it is more efficient to compute slices for several smaller files than computing it for one big trace file.
3) Finally a union of all the slicing criteria that correspond to check statements is built, since the slicer supports to build a slice for a set of slicing criteria. By merging the slices from the test classes, we obtain a set of all statements that contribute to checked values. The *checked coverage score* is then computed by dividing the number of statements in this set by the—previously computed—number of coverable statements.

III. EVALUATION

In the evaluation of our approach, we were interested whether checked coverage can help in *improving existing test suites* of mature programs. To this end, we computed the checked coverage for seven open-source programs that have undergone several years of development and come with a JUnit test suite. We manually analyzed the results, and found examples where the test suites can be improved to more thoroughly check the computation results (see Section III-B). We also detected some limitations of our approach, summarized in Section IV.

Furthermore, we were interested how sensitive our technique is to *oracle decay*—that is, oracle quality artificially reduced by removing checks. In a second automated experiment (Section III-C), we removed a fraction of the assert-statements from the original test suites and computed the checked coverage for these manipulated test suites. This setting also allows us to compare checked coverage against other techniques that measure test quality, such as statement coverage and mutation testing.

A. Evaluation Subjects

As evaluation subjects we used seven open-source projects that are listed in Table I. The subjects come from different application areas (column 2), and we took a recent revision from the version control system (column 3)—except for JTOPAS, which was taken from the software-artifact infrastructure repository (SIR) [9]. We removed tests that fail, and tests whose outcome is dependent on the order or frequency of execution, as we consider this a flaw of the test suite.

Table II
CHECKED COVERAGE, STATEMENT COVERAGE, AND MUTATION SCORE.

Project Name	Checked Coverage %	Statement Coverage%	Mutation Score %
ASPECTJ	13	38	63
BARBECUE	19	32	66
COMMONS	62	88	86
JAXEN	55	78	68
JODA-TIME	47	71	83
JTOPAS	65	83	73
XSTREAM	40	77	87
Average	**43**	**67**	**75**

Table II gives the results for computing the checked coverage (column 2), statement coverage (column 3) and the mutation score (column 4) for our subject projects. The statement coverage values are between 70% and 90% for all projects except for ASPECTJ and BARBECUE. For ASPECTJ the results are lower, because we only used a part of the test suite, in order to run our experiments in a feasible amount of time. Although we only computed the coverage of the module that corresponds to the test suite, test suites of other modules might also contribute to the coverage of the investigated module. For BARBECUE, we had to remove tests that address graphical output of barcodes, as we ran our experiments on a server that has no graphics system installed and this causes these tests to fail. Consequently, these parts are not covered.

In all projects, checked coverage is lower than regular coverage, with an average difference of 24%. With 37% this difference is most pronounced for XSTREAM. This is due to a library class that directly manipulates memory and is used by XSTREAM in a central part. As this takes place outside of Java, some dependencies cannot be traced by the slicer, which leads to statements not being considered for checked coverage, although they should.

The traditional definition of the mutation score is the number of detected mutations divided by the total number of mutations. In our setting, we only consider the score for covered mutations (last column). These values are also lowest for ASPECTJ and BARBECUE, because of the reasons mentioned earlier.

B. Qualitative Analysis

In our first experiment we were interested in whether checked coverage can be used to improve the oracles of a test suite. We computed checked and statement coverage for each class individually. Then we manually investigated those classes with a difference between checked and regular coverage, as this indicates code that is executed without checking the computation results.

In the introduction we have seen such an example for a test of `PatternParser`, a helper class for parsing XSLT patterns, from the JAXEN project (Figure 1). The corresponding test class calls the `parse()` method with valid inputs (shown in Figure 1) and invalid inputs, and passes when no exception or an expected exception is thrown. The computation results of the `parse()` method, however, are not checked. Consequently, this leads to a checked coverage of 0%.

```
boolean checkCreateNumber(String val){
  try {
    Object obj =
        NumberUtils.createNumber(val);
    if (obj == null) {
      return false;
    }
  return true;
  } catch (NumberFormatException e) {
    return false;
  }
}
```

Figure 3. Another test with insufficient outcome checks.

Another example of missing checks can be found in the tests for the `NumberUtils` class of the COMMONS-LANG project. Some statements of the `isAllZeros()` method— that is indirectly called by the `createNumber()` method — are not checked although they are covered by the tests. The test cases exercise these statements via the `checkCreateNumber()` method shown in Figure 3. This method calls `createNumber()` and returns `false` when null is returned or an exception is thrown, or `true` otherwise. The result of `createNumber()`, however, is not adequately checked. Adding a test that checks the result of `createNumber()` would include the missing statements for the checked coverage.

```
public String next()
    throws TokenizerException {
  nextToken();
  return current();
}
```

Figure 4. A method where the return value is not checked.

Figure 4 shows the `next()` method from the `AbstractTokenizer` class of the JTOPAS project. Although this method is executed several times by the test suite, its return value is never checked and consequently reported as missing from checked coverage. This means that the method could return any value and the test suite would not fail. In the same way as for the previous examples, adding an assertion that checks the return value properly solves this problem.

Mature test suites miss checks.

C. Disabling Oracles

In our first experiment, we have seen cases were a test suite covers parts of the program, but does not check

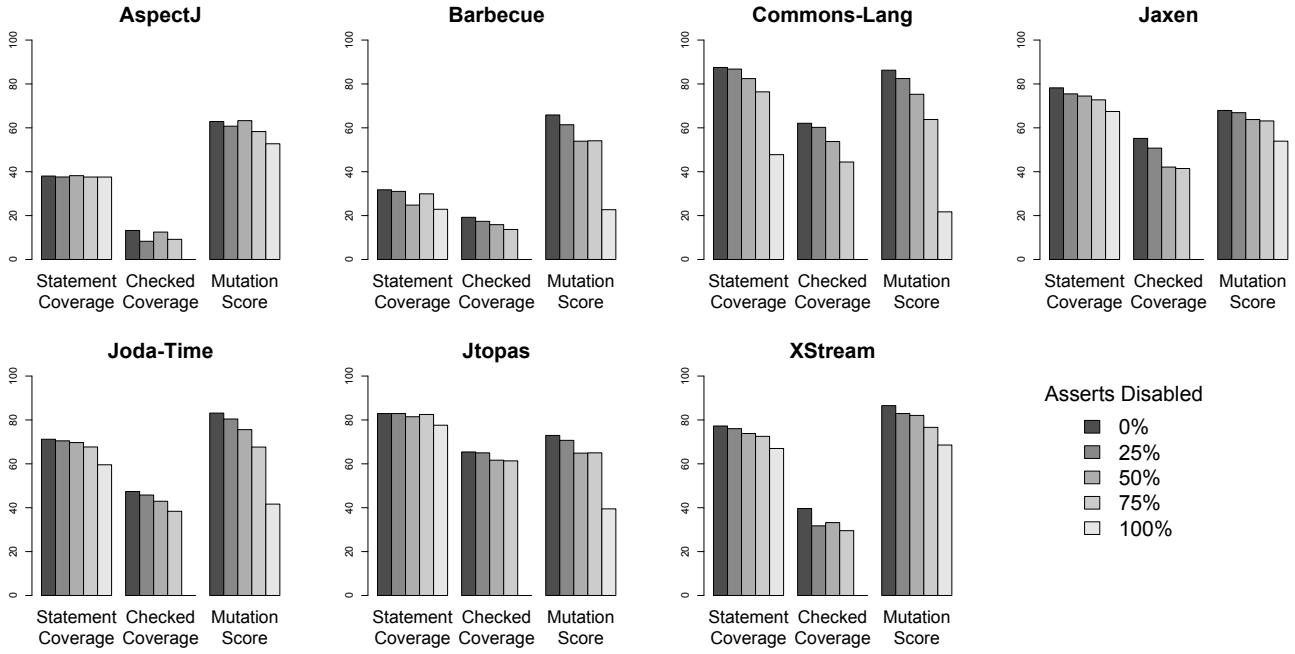

Figure 5. Coverage values for test suites with removed assertions.

the results well enough. As discussed earlier, this can be detected by checked coverage. In our second experiment, we explored the questions:

- *How sensitive is checked coverage to missing checks?*
- *How does checked coverage compare to other metrics that measure the quality of a test suite?*

To this end, we took the original JUnit test suites of the seven projects, and produced new versions with *decayed oracles*—that is, we systematically reduced oracle quality by removing a number of *assertions* executed in these oracles. In JUnit, assertions constitute the central means for checking computation results; therefore, removing assertions means disabling checks and therefore reducing oracle quality.

To disable an assertion in the source code, we completely removed the line that contains the assertion. In some cases, we had to remove some additional statements, as the new test suites were not compilable anymore, or some of the test cases failed when assertions were removed, because they relied on the side effects that happened inside the removed statements. After the test suite could be successfully compiled and had no failing tests anymore, we computed the coverage metrics for each of the test suite.

The results are given in Figure 5. For each of our subject programs there is a plot that shows the statement coverage value, checked coverage value, and mutation score for a test suite with all assertions enabled (0% removed), and with 25, 50, 75, and 100 percent of the assertions removed, respectively.

For almost all projects, all metric values decrease with a decreasing number of assertions. An exception is ASPECTJ, where the statement coverage values stay constant for all test suites. This is due to the nature of the ASPECTJ test suite that does not have any computations inside assertions. Furthermore, the checked coverage value and the mutation score are higher for the test suite with 50% assertions removed than for the suite with 25% removed. As we choose the disabled assertions randomly each time, some assertion statements that check more parts of the computation were disabled in the 25% test suite and not disabled in the 50% test suite. This also explains the higher values for statement coverage in the 75% than the 50% test suite for BARBECUE, and the difference for XSTREAM and checked coverage between 50 and 25 percent.

All test quality metrics decrease with oracle decay.

In order to compare the decrease of the different metrics, we computed the decrease of each metric relative to the value for the original test suite. Figure 6 shows the results for the seven subject programs.

For statement coverage, the decrease values are the lowest for all projects. This comes by no surprise, as it is not designed to measure the quality of checks in a test suite. Thus, it is the least sensitive metric to missing assert-statements.

Checked coverage and mutation score show a similar development for BARBECUE, COMMONS, JODA-TIME and JTOPAS for 0–75% of removed checks. For the other projects, there is a greater decrease for the checked coverage than for the mutation score. On average, when 75% of the tests are removed, checked coverage decreases by 23%, whereas the mutation testing score only decreases by 14%.

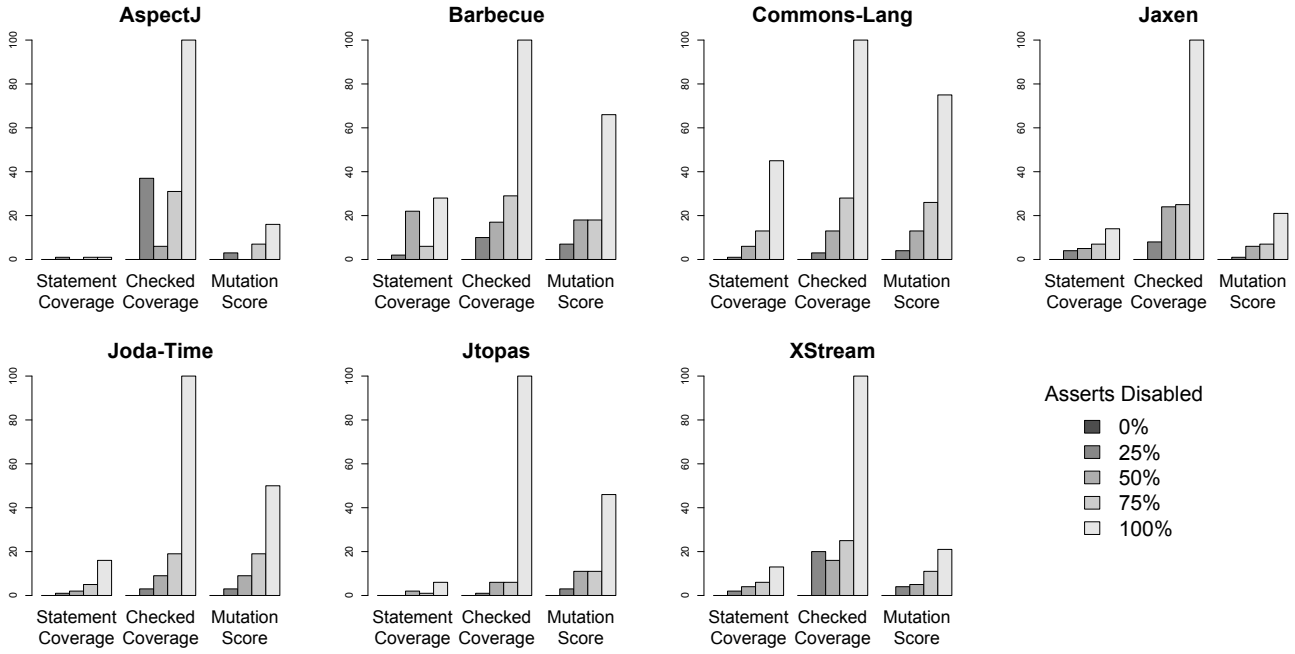

Figure 6. Decrease of the coverage values relative to the coverage values of the original test suite.

Checked coverage is more sensitive to missing assertions than statement coverage and mutation testing.

Note that when *all checks* are removed, checked coverage drops to 0% for all projects. This is by construction of the approach, as there are no statements left to slice from once all checks are removed from the test suite. Since we are interested in *poor* rather than nonexistent oracles, the results obtained for the decays of 25–75% are much more meaningful.

D. Explicit and Implicit Checks

As mutation testing is also a measure of a test suite's assertion quality, we might also expect that the mutation score dramatically drops when no checks are left. However, in the previous experiment, we have seen that test suites with no assertions still detect a significant fraction of the mutations (43 % on average). The reason for this is that a mutation can be detected by an *explicit* check of the test suite or by *implicit* checks of the runtime system. For modern object-oriented and restrictive programming languages like Java, many mutations are detected through theses implicit runtime checks.

JAVALANCHE, the mutation testing tool that we use for this study, uses the distinction made by JUnit in order to classify a mutation as either detected by explicit or implicit checks. JUnit classifies a test that does not pass either as a failure (explicit), when an `AssertionError` was thrown (e.g by an JUnit assertion method), or as an error (implicit), when any other exception was thrown. As we were interested how pronounced this effect is, we computed the fraction of mutations detected by assertions for the original test suite and the test suite with all checks removed.

Table III
MUTATIONS DETECTED BY EXPLICIT AND IMPLICIT CHECKS.

Project Name	Original Test Suite Total	Original Test Suite Explicit%	All checks removed Total	All checks removed Explicit%
ASPECTJ	5736	63%	5529	53%
BARBECUE	1036	59%	357	6%
JODA-TIME	12164	56%	6094	15%
JAXEN	4154	38%	3298	12%
JTOPAS	1295	57%	597	0%
COMMONS	13415	68%	3377	16%
XSTREAM	7764	35%	5156	5%
Total	**45564**	**55%**	**24408**	**21%**

Table III gives the results for each project. The first two values are for the original test suite. First, the total number of detected mutations (column 2) is given, and then the percentage of them that are detected by implicit checks (column 3). The remaining mutations are detected by explicit checks. The two last columns give the corresponding values for the test suite with all assert-statements removed.

For the original test suite, 32–65% of the mutations are detected by implicit checks. This is mainly due to `NullPointerExceptions` caused by mutations.

Almost half (45%) of the mutations are detected by implicit checks.

The test suites with all checks removed still detect 54% of the mutations that are detected by the original test suite; on average, 21% of the detected mutations are found by explicit

checks. For this test suite, one might expect 0% of the mutations detected by explicit checks. However, assertions in external libraries—that are not removed— cause this 21%. Examples include the `fail()` method of JUnit or the `verify()` method of a mocking framework.

One could argue that it need not matter how a mutation is being found—after all, the important thing is that it is found at all. Keep in mind, though, that implicit checks are not under control by the programmer. A mutation score coming from implicit checks thus reflects the *quality of the run-time checks* rather than the quality of the test suite. Also, the run-time system will only catch the most glaring faults (say, null pointer dereferences or out of bound accesses), but will not check any significant functional property. Therefore, having mutations fail on implicit checks only is an indicator for poor oracle quality—just as a low checked coverage.

A test suite with no assertions still detects over 50% of the mutations detected by the original test suite; around 80% of these are detected by implicit checks.

E. Performance

Table IV shows the runtime for checked coverage and mutation testing. The checked coverage is computed in two steps. First a run of the test suite is traced (Column 2), then, using the slicer, the checked coverage (Column 3) is computed from the previously produced trace file. Column 4 gives the total time needed to compute the checked coverage. For almost all projects, the slicing step takes much longer than tracing the test suite. XSTREAM, however, is an exception. Here the slicing takes less time, because some of the central dependencies are not handled by the tracer (see Section IV). The last column gives the time needed for mutation testing the programs with JAVALANCHE. When we compare the total time needed to compute the checked coverage with the time needed for mutation testing, checked coverage is faster for four of our projects and mutation testing is faster for three of the projects. Keep in mind, though, that JAVALANCHE reaches its speed only through a dramatic reduction in mutation operators; full-fledged mutation testing requires a practically unlimited number of test runs.

Table IV
RUNTIME TO COMPUTE THE CHECKED COVERAGE AND THE MUTATION SCORE.

Project Name	Checked Coverage Trace	Slice	Total	Mutation Test
ASPECTJ	0:08:51	0:35:26	0:43:17	20:18:38
BARBECUE	0:06:10	0:15:30	0:21:40	0:06:07
COMMONS	0:32:07	3:40:37	1:12:44	1:29:06
JAXEN	0:24:21	0:37:18	1:01:39	1:29:00
JODA-TIME	0:23:53	1:38:10	2:02:03	0:45:43
JTOPAS	0:04:04	0:05:32	0:09:36	0:41:25
XSTREAM	0:40:13	0:16:35	0:56:48	1:49:13

In terms of performance, checked coverage is on par with the fastest mutation testing tools.

IV. LIMITATIONS

In some cases, statements that contribute to the computation of results that are later checked by oracles are not considered for the checked coverage due to limitations of JAVASLICER or limitations of our approach.

Native code imposes one limitation to the tracer. In Java it is possible to call code written in C, C++, or assembly via the Java Native Interface (JNI). This code cannot be accessed by the tracer as it only sees the bytecode of the classes loaded by the JVM. Regular programs rarely use this feature. In the Java standard library however, there are many methods that use the JNI. Examples include the `System.arraycopy()` method, or parts of the Reflection API. In these cases, the dependencies between the inputs and the outputs of the methods are lost.

This limitation also caused the huge differences between normal coverage and checked coverage for the XSTREAM project. XSTREAM uses the class `sun.misc.Unsafe`, which allows direct manipulation of the memory in a core part. Therefore many dependencies get lost and the checked coverage is lower than expected. Another limitation imposed by the tracer is that the `String` class is currently not handled. This class is used heavily in core classes of both the JVM and the tracer; which makes it difficult to instrument without running into circular dependencies. Handling this class would allow the slicer to detect dependencies that are currently missed.

```
try {
    methodThatShouldThrowException();
    fail("No exception thrown");
} catch(ExpectedException e) {
    // expected this exception
}
```

Figure 7. A common JUnit pattern to check for exceptions.

A frequently used practise to check for exceptions is to call a method under such circumstances that it should throw an exception, and fail when no exception is thrown (Figure 7). When the exception is thrown, everything is fine, and nothing else is checked. Therefore, in our setting, the statements that contributed to the exception are not on a slice, and thus do not contribute to the checked coverage. A remedy would be to introduce an assertion that checks for the exception.

```
private boolean inSaneState = true;
...
if(!inSaneState)
    exceptionThrowingMethod();
...
```

Figure 8. Statements that lead to not taking a branch.

Another limitation introduced by our approach are computations that lead to a branch not being taken. In Figure 8, for example we have the boolean flag `inSaneState`. Later, an exception is thrown when this flag has the value `false`. Thus only computations that lead to the variable being set to false can be on a dynamic slice, and computations that lead to the variable being set to true will *never* be on a dynamic slice. Such problems are inherent to dynamic slicing, and would best be addressed by computing static dependencies to branches not taken.

V. Threats to Validity

Like any empirical study, this study has limitations that must be considered when interpreting its results.

- **External validity.** Can we generalize from the results of our study? We have investigated seven different open-source projects, covering different standards in maturity, size, domain, and test quality. But even with this variety, it is possible that our results do not generalize to other arbitrary projects.
- **Construct validity.** Are our measures appropriate for capturing the dependent variables? The biggest threat here is that our implementation could contain errors that might affect the outcome. To control for this threat, we relied on public, well-established open-source tools wherever possible; our own JAVALANCHE and JAVASLICER frameworks are publicly available as open source packages to facilitate replication and extension of our experiments.
- **Internal validity.** Can we draw conclusions about the connections between independent and dependent variables? The biggest threat here is that we only use sensitivity to oracle decay as dependent variable—rather than a more absolute "test quality" or "oracle quality". Unfortunately, there is no objective assessment of test quality to compare against. The closest would be mutation testing [10], but as our results show, even programs without oracles can still achieve a high mutation score by relying on uncontrolled implicit checks. As it comes to internal validity, we are thus confident that sensitivity to oracle decay is the proper measure; a low checked coverage therefore correctly indicates a low oracle quality.

VI. Related Work

A. Coverage Metrics

During structural testing a program gets tested using knowledge of its internal structures. Hereby, one is interested in the *quality* of the developed tests, and how to *improve* them in order to detect possible errors. To this end, different coverage metrics have been proposed and compared against each other regarding their effectiveness in detecting specific types of errors, relative costs, and difficulty of satisfying them [11], [12], [13], [14]. Each coverage metric requires different items to be covered. This allows to compute a *coverage score* by dividing the number of coverable items by the number of items actually covered.

Best known, and most easy to compute is *statement coverage*. It simply requires each line to be executed at least once. Because some defects can only occur under specific conditions, more complex metrics have been proposed. The popular *branch coverage* requires each condition to evaluate to both true and false at least once; *decision coverage* extends this condition to boolean subexpressions in control structures. A more theoretical metric is *path coverage,* measuring how many of the (normally infinitely many) possible paths have been followed. Similar to our approach *data flow testing criteria* [15] also relate definitions and uses of variables. These techniques consider the relation between all defined variables inside the program and their uses. For example, the *all-uses* criterion requires that for each definition use pair a path is exercised that covers this pair. In contrast, our approach is only targeted at uses inside the oracles. Other definition use pairs are followed transitively from there.

Each of these proposed metrics just measures how well specific structures are exercised by the provided test input, and not how well the outputs of the program are checked. Thus, they *do not assess oracle quality* of a test suite.

B. Mutation Testing

A technique that aims at checking the quality of the oracles is *mutation testing*. Originally proposed by Richard Lipton [16], [17], mutation testing seeds artificial defects, as defined by *mutation operators*, into a program and checks whether the test suite can distinguish the mutated from the original version. A mutation is supposed to be detected ("killed"), if at least one test case fails on the mutated version that passed on the original program. If a mutation is not detected by the test suite, similar defects might be in the program that are also not detected by the test suite. Thus, these undetected mutants can give an indication on how to improve the test inputs and checks. However, not every undetected mutant helps in improving the test suite as it might also be an *equivalent mutant*; that is a mutation that changes the syntax but not the semantics of a program.

For our experiments we use the JAVALANCHE mutation testing framework, that was developed with a focus on automation and scalability. To this end, JAVALANCHE applies several optimizations, such as selective mutation [18], [19], mutant schemata [20], using coverage data to reduce the number of tests that need to be executed, and allowing parallel execution of different mutations. A more detailed description of JAVALANCHE can be found in our earlier papers [21], [22].

C. Program Slicing

Static program slicing was originally proposed by Weiser [1], [2] as a technique that helps the programmer during debugging. Korel and Laski introduced dynamic slicing that computes slices for a concrete program run. Furthermore, different slicing variants have been proposed for program comprehension; *Conditioned Slicing* [23] is a mix between dynamic and static slicing, it allows some variables to have a fixed value while others can take all possible values. *Amorphous Slicing* [24] requires a slice only to preserve the semantics of the behavior of interest, while syntax can be arbitrarily changed, which allows to produce smaller slices.

Besides its intended use in debugging, program slicing has been applied to many different areas [25], [26]. Examples include, minimization of generated test cases [27], automatic parallelization [28], [25], and the detection of equivalent mutants [29].

D. State Coverage

The concept closest to checked coverage is *state coverage* proposed by Koster and Kao [30]. It also measures the quality of checks in a test suite. To this end, all output defining statements (ODS)—statements that define a variable that can be checked by the test suite—are considered. The state coverage is defined as the number of ODS that are on a dynamic slice from a check divided by the total number of ODS. This differs from our approach, as we also consider statements that influence the computation of variables that are checked.

Furthermore, the number of ODS is dependent on the test input: For different inputs, different statements might thus be considered as output defining. This can lead to cases where a test suite is improved by adding additional tests (with new inputs), but the state coverage drops. Checked coverage stays constant or improves in such cases.

Unfortunately, there is no broader evaluation of state coverage that we can compare against. In a first short paper [30], a proof of concept based on a static slicer, and one small experiment is presented. A second short paper [31] describes an implementation based on *taint analysis* [32], but no experimental evaluation is provided.

VII. Conclusion and Consequences

To assess the quality of a test suite, developers so far had the choice between two extremes: *Coverage metrics* are efficient, but fail to assess oracle quality; and *mutation testing* detects oracle issues, but is expensive in terms of computation time and human labor. By assessing which parts of the covered computation are actually checked by oracles, *checked coverage* covers the middle ground, providing a straight-forward means to assess oracle quality at reasonable efficiency—and even more sensitive to oracle quality than mutation testing, as shown in our experiments.

Like any quality metric, checked coverage can only be an imperfect assessment of a test suite; a full assessment needs to consider the entire problem context and its associated risks. Still, some aspects of programs and their quality assurance could eventually be integrated into the concept. Besides general improvements regarding scalability, efficiency, and robustness, our future work will focus on the following topics:

- **Checking contracts.** *Contracts*—that is, runtime checks for pre- and postconditions and invariants—promise to detect errors long before they escape to the end of the computation. The concept of checked coverage naturally extends to contracts, thus providing a metric to which extent runtime checks are conducted.
- **Defensive programming.** Programs sometimes check their inputs without using explicit assertions, and throw an exception or otherwise indicate an error. Such checks could also be treated to contribute to checked coverage. The challenge is to separate "regular" from "unexpected" behavior—a separation which is explicit in assertions and tests.
- **Static verification.** If we can formally prove that some condition always holds, then the appropriate result should also be considered "covered". Checked coverage thus gives a means to assess the "proof coverage" of a program.

To repeat and extend these experiments, all one needs is a dynamic slicer. Hammacher's JAVASLICER is now available as open-source at

`http://www.st.cs.uni-saarland.de/javaslicer/`

To learn more about our work in assessing test suite quality, see our project page

`http://www.st.cs.uni-saarland.de/mutation/`

Acknowledgments. Yana Mileva, and Kevin Streit as well as the anonymous reviewers provided helpful feedback on earlier revisions of this paper; special thanks go to Clemens Hammacher for his detailed comments as well as his support on using JAVASLICER.

References

[1] M. Weiser, "Program slices: formal, psychological, and practical investigations of an automatic program abstraction method," Ph.D. dissertation, University of Michigan, 1979.

[2] ——, "Program slicing," *IEEE Trans. on Software Engineering*, vol. 10, no. 4, pp. 352–357, 1984.

[3] B.Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3, pp. 155–163, 1988.

[4] C. Hammacher, "Design and Implementation of an Efficient Dynamic Slicer for Java," Bachelor's Thesis, Saarland University, November 2008.

[5] C. G. Nevill-Manning, I. H. Witten, and D. Maulsby, "Compression by induction of hierarchical grammars," in *Data Compression Conference*, 1994, pp. 244–253.

[6] C. G. Nevill-Manning and I. H. Witten, "Linear-time, incremental hierarchy inference for compression," in *Data Compression Conference*, 1997, pp. 3–11.

[7] T. Wang and A. Roychoudhury, "Using compressed bytecode traces for slicing Java programs," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, 2004, pp. 512–521.

[8] ——, "Dynamic slicing on Java bytecode traces," *ACM Trans. on Programming Languages and Systems*, vol. 30, no. 2, 2008.

[9] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[10] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, 2005, pp. 402–411.

[11] J. C. Huang, "An approach to program testing," *ACM Computing Surveys*, vol. 7, no. 3, pp. 113–128, 1975.

[12] S. C. Ntafos, "A comparison of some structural testing strategies," *IEEE Trans. on Software Engineering*, vol. 14, no. 6, pp. 868–874, 1988.

[13] E. J. Weyuker, S. N. Weiss, and R. G. Hamlet, "Comparison of program testing strategies," in *Symposium on Testing, Analysis, and Verification*, 1991, pp. 1–10.

[14] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses versus mutation testing: An experimental comparison of effectiveness," *Systems and Software*, vol. 38, pp. 235–253, 1997.

[15] S. Rapps and E. J. Weyuker, "Data flow analysis techniques for test data selection," in *ICSE '82: Proceedings of the 6th international conference on Software engineering*, 1982, pp. 272–278.

[16] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the orthogonal," pp. 34–44, 2001.

[17] R. A. DeMillo, R. J. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[18] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Trans. on Software Engineering and Methodology*, vol. 5, no. 2, pp. 99–118, 1996.

[19] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei, "Is operator-based mutant selection superior to random mutant selection?" in *ICSE' 10: Proceedings of the 32nd International Conference on Software Engineering*, 2010, pp. 435–444.

[20] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *ISSTA '93: Proceedings of the International Symposium on Software Testing and Analysis*, 1993, pp. 139–148.

[21] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient mutation testing by checking invariant violations," in *ISSTA '09: Proceedings of the International Symposium on Software Testing and Analysis*, 2009, pp. 69–80.

[22] D. Schuler and A. Zeller, "(Un-)Covering equivalent mutants," in *ICST'10: Proceedings of the 3rd International Conference on Software Testing Verification and Validation*, 2010, pp. 45–54.

[23] A. D. Lucia, A. R. Fasolino, and M. Munro, "Understanding function behaviors through program slicing," in *4th International Workshop on Program Comprehension*, 1996, pp. 9–10.

[24] M. Harman and S. Danicic, "Amorphous program slicing," in *5th International Workshop on Program Comprehension*, 1997, pp. 70–79.

[25] F. Tip, "A survey of program slicing techniques," *Journal of Programming Language*, vol. 3, no. 3, pp. 121–189, 1995.

[26] M. Harman and R. M. Hierons, "An overview of program slicing," *Software Focus*, vol. 2, no. 3, pp. 85–92, 2001.

[27] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, "Efficient unit test case minimization," in *Proceedings of the 22nd international conference on Automated Software Engineering*, 2007, pp. 417–420.

[28] C. Hammacher, K. Streit, S. Hack, and A. Zeller, "Profiling Java programs for parallelism," in *IWMSE'09: Proceedings of the 2nd International Workshop on Multi-Core Software Engineering*, 2009.

[29] R. Hierons and M. Harman, "Using program slicing to assist in the detection of equivalent mutants," *Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 233–262, 1999.

[30] K. Koster and D. Kao, "State coverage: A structural test adequacy criterion for behavior checking," in *ESEC/FSE '07: Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2007, pp. 541–544.

[31] K. Koster, "A state coverage tool for JUnit," in *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, 2008, pp. 965–966.

[32] J. A. Clause, W. Li, and A. Orso, "Dytan: A generic dynamic taint analysis framework," in *ISSTA '07: Proceedings of the International Symposium on Software Testing and Analysis*, 2007, pp. 196–206.