SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2013; **23**:531–551
Published online 8 May 2013 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.1497

Checked coverage: an indicator for oracle quality

David Schuler*,† and Andreas Zeller

Department of Computer Science, Saarland University, Saarbrücken, Germany

SUMMARY

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. Copyright © 2013 John Wiley & Sons, Ltd.

Received 11 September 2011; Revised 22 November 2012; Accepted 13 March 2013

KEY WORDS: test suite quality; coverage metrics; dynamic slicing; mutation testing

1. INTRODUCTION

How can I ensure that my programme 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 programme itself (e.g. a failing assertion) or by the run-time system (e.g. a dereferenced null pointer). To validate computation results, however, we need checks in the test code and in the programme 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 case passes a set of paths through the parser. Interestingly enough, it never checks any of the parser results. The parser may return complete nonsense, and this test would never notice it.

Why are the parser results unchecked? Actually, the purpose of this test is not to test the parser, but the individual paths in the `paths` array; if any of these contain a syntactical error, the parser would throw an exception. If the test passes, all paths are fine. Running this test, however, also achieves a statement coverage of 83% in the parser, and one may thus assume the parser is well-tested.

*Correspondence to: David Schuler, Department of Computer Science, Saarland University, Saarbrücken, Germany.
†E-mail: ds@cs.uni-saarland.de

Copyright © 2013 John Wiley & Sons, Ltd.

```
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.

However, this test case is the only one in the entire test suite that covers the `PatternParser` class[‡], and thus, no test ever actually checks whether the parser works as advertised.

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. By using dynamic slicing, we determine the *checked coverage*—statements that were not only executed, but that actually contribute to the results checked by oracles.

Let us illustrate the concept on the `PatternParser` test in Figure 1. Because none of the results ever flows into a check, the checked coverage for a run of the whole test suite on this class is 0%, which is in contrast to the 83% traditional statement 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.

As a metric, checked coverage has significant advantages over regular coverage:

- 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.
- Developers would focus on checking as many results as possible, catching more errors related to missing checks.
- To compute checked coverage, one only needs to run the test suite once with a constant overhead, which can be more efficient than the potentially unlimited number of executions induced by mutation testing.

This paper introduces the concept of checked coverage (Section 2) and evaluates the concept on seven open-source projects (Section 3). Our results show that checked coverage is a sure indicator for oracle quality and even more sensitive to missing oracles than mutation testing. Then, we comment on the limitations (Section 4) of our approach and present an extension that includes tests checking for exceptional behaviour (Section 5). After discussing the threats to validity (Section 6) and the related work (Section 7), we close with conclusion and consequences (Section 8).

This paper extends our prior International Conference on Software Testing, Verification and Validation paper on checked coverage [1] in the following areas: it starts with a more detailed discussion of programme slicing (Section 2.1), the underlying technique of checked coverage. Then, we give a more formal definition of checked coverage (Section 2.2) and comment on infeasible requirements imposed by checked coverage (Section 2.3). We describe the intended usage scenarios for checked coverage (Section 2.5), that is, to identify deficiencies in the checks of the test suite, which can either be weak checks or unchecked programme parts. Additionally, we describe how to extend the concept of checked coverage to other coverage metrics (Section 2.6).

[‡]The source repository of JAXEN contains another test case that exercises the PatternParser class. However, the main test suite of the project does not include this test. Thus, this test was not considered for our experiments.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

The evaluation is updated by using more test splits (Section 3.4). Moreover, we extended the implementation of checked coverage to take tests that implicitly check for exceptions into account and evaluated this extension (Section 5).

2. CHECKED COVERAGE

Our concept of checked coverage is remarkably simple: Rather than computing *coverage*, the extent to which code features are *executed* in a programme, 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.*

2.1. Programme slicing

Programme slicing was introduced by Weiser [2, 3] as a technique that determines the set of statements that potentially influences the variables used in a given location. Weiser claims that this technique corresponds to the mental abstractions programmers make when they debug a programme.

A static backward slice is computed from a *slicing criterion* that consists of a statement and a subset of the variables used in the programme. A slice for a given slicing criterion is computed by transitively following *all data and control dependencies* for the variables from the statement, whereas data and control dependencies are defined as follows:

Definition 1 (Data dependency)
A statement s is data-dependent on a statement t if 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 if t is a conditional statement and the execution of s depends on t.

Definition 3 (Backward slice)
A backward slice for a slicing criterion (s, V) is the transitive closure over all data and control dependencies for a set of variables V at statement s.

Korel and Laski [4] 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 programme run, that is, the dynamic slice is computed by following only the data and the control dependencies that actually emerged during a specific run of the programme.

A *dynamic slicing criterion* specifies, in addition to the static slicing criterion, the input to the programme and distinguishes between different occurrences of a statement. A *dynamic backward slice* is then computed by transitively following all dynamic dependencies for a set of variables, which are used in a specific occurrence of a statement in a programme run that was obtained by using the specified input.

Definition 4 (Dynamic data dependency)
A statement s is dynamically data-dependent on a statement t for a run r, if there is a variable v that is defined (written) in t and referred to (read) in s during the programme run without an intermediate redefinition of v.

Definition 5 (Dynamic control dependency)
A statement s is control-dependent on a statement t for a run r, if s is executed during the run and t is a conditional statement that controls the execution of s.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Definition 6 (Dynamic backward slice)
A dynamic backward slice for a dynamic slicing criterion (s_o, V, I) is the transitive closure over all dynamic data and control dependencies for the variables V, in a run with input I, at the o-th occurrence of statement s.

In contrast to dynamic slices, static slices take into account all possible dependencies. Therefore, static slices tend to include more statements than dynamic slices.

The code shown in Figure 2, for example, displays a method that computes the maximum for two integers, and a statement that calls this method and assigns the result to a variable. Solid arrows show the data dependencies, whereas dashed arrows display the control dependencies. The static backward slice from the last statement consists of statements $\{1, 4, 5, 7, 9, 11\}$. The increment of `countCalls` (Statement 3), for example, is not included in the trace, as there is neither a transitive static control nor data dependency from the variables used in the last statement to `countCalls`. Note that there are different definitions of which statements should be included in a slice. Although there are definitions that require the slice to be executable, we only consider those statements where an actual control or data dependency exists. Thus, some statements will never be included in a slice because there exist no dependencies, such as Statement 2 in the example, which just declares a variable.

Figure 3 shows the dynamic data and the control dependencies, for example, the dependencies that actually emerge during the programme run. The dynamic slice consists of the statements $\{1, 4, 5, 9, 11\}$. In contrast to the static slice, the statement in the else part (Statement 7) of the `max()` method is excluded because this code is not exercised by this run.

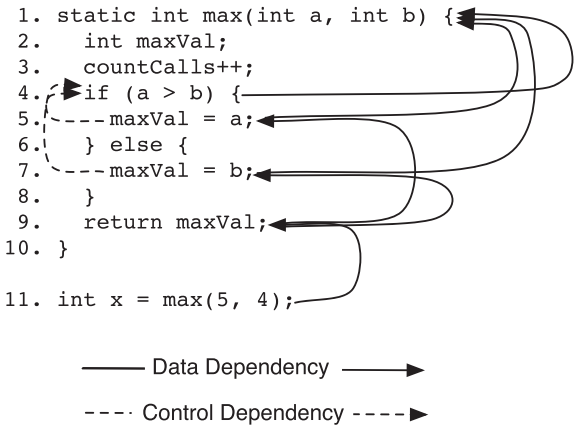

Figure 2. Static data and control dependencies for the `max()` method.

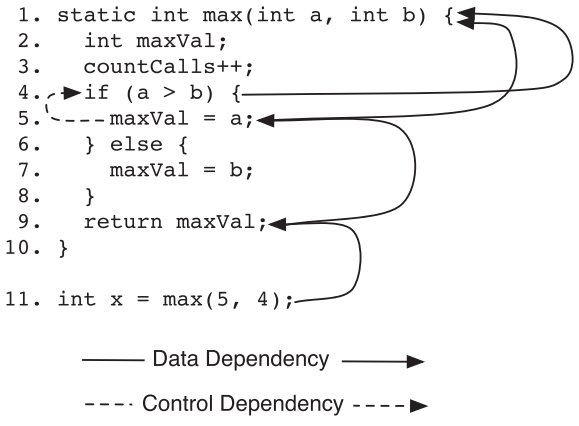

Figure 3. Dynamic data and control dependencies for the `max()` method.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

2.2. 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 are 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 a 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.

For a more formal definition of checked coverage, we adopt the style of Ammann and Offutt [5] to define coverage criteria in the form of requirements that are imposed on the tests.

Definition 7 (Coverage criterion)
A coverage criterion is a rule that imposes requirements on a test suite.

Definition 8 (Test requirement)
A test requirement is a specific element or property of the system that the test must cover or satisfy.

To define checked coverage in terms of a test requirement, we define the set of statements that can be on a slice. Some statements of a programme can never show up on a slice because there exists no data or control dependency from and to them. Therefore, for checked coverage, only the statements that read or write variables or that can influence the control flow of the programme are considered.

Definition 9 (Sliceable statements)
For a programme P, the set of sliceable statements $SL(P)$ consists of all statements that read or write a variable or manipulate the control flow.

With the help of this set, we can define the test requirements of checked coverage.

Definition 10 (Checked coverage)
Checked coverage requires each sliceable statement $s \in SL(P)$ to be on at least one dynamic backward slice from an explicit check of the test suite.

The coverage score of a test suite for a criterion is the percentage of requirements that are fulfilled by the test suite relative to the total number of requirements imposed by the criterion. For checked coverage, this is the number of statements that are on a dynamic backward slice from a check of the statement relative to all sliceable statements.

Definition 11 (Checked coverage score)
The checked coverage score is the percentage of statements that are on a backward slice from a check relative to the total number of sliceable statements.

Note that for computing the checked coverage score, only sliceable statements from the system under test are considered, whereas statements from 3rd party libraries or the Java API are not considered. The statements in these libraries, however, are considered by the slicer, that is, transitive dependencies via these statements are detected. We chose to exclude the statements from the checked coverage score because a system only uses parts of the library. Typically, when testing a system, one is not interested in thoroughly testing the libraries. However, the implementation can be modified to also include statements from libraries for the computation of the checked coverage score. Moreover, we consider elementary instructions as statements. For example, if a statement is in a for-loop, there is only a control dependency on the condition of the loop. The other statements of the loop command (initialization and increment) might be included via transitive dependencies.

Figure 4 shows a test that exercises the `max` method, similar to the slicing examples (Figures 2 and 3) in Section 2.1. To compute the checked coverage for this example, the dynamic backward slice from the call to `assertEquals` (Statement 12) is built, and only the statements that are on this slice are considered to be covered. The set of sliceable statements of the `max()` method are the

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

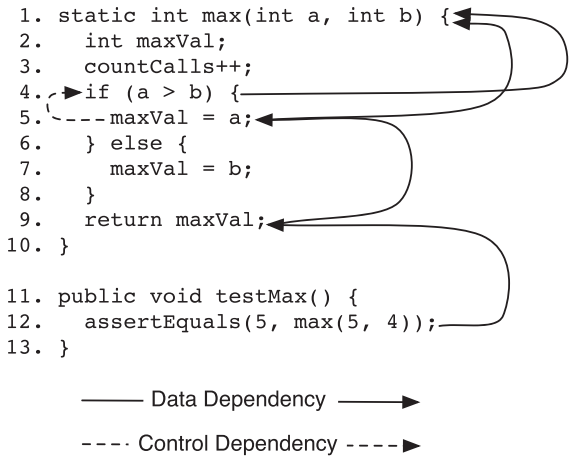

Figure 4. Dynamic data and control dependencies as used for checked coverage.

statements {1,3,4,5,7,9}. Out of these statements, {1,4,5,9} are on a backward slice from the check, whereas the Statements 3 and 7 are not, which leads to a checked coverage score of 67% (four out of six requirements are fulfilled). The assignment of `b` to `maxVal` in Line 7 is not executed. The increment of `countCalls` (Statement 3), on the other hand, is executed but not checked, which is an example for the intended use of checked coverage—*to detect computations that are not properly checked*.

Checked coverage subsumes statement coverage because to reach full-checked coverage, every statement has to be on a dynamic slice, and thereby, it also has to be executed at least once. Thus, every test suite that reaches full-checked coverage also reaches full statement coverage. Note that in general, checked coverage does not subsume all definitions' coverage. There are two reasons for this. First, statements that manipulate the control flow can also define variables. Thus, they can end up on a slice because of control dependencies, although their data dependencies are not exercised. Second, uses can also occur inside the test suite. In this case, a definition is on a slice although no use of it in the programme is exercised. If we account for these two special cases, that is, we do not allow definitions in control flow manipulating statements and we consider uses inside the test suite, checked coverage subsumes all definitions coverage. This is because a variable that is defined in a statement is always used when it appears on a slice, as the only way it can end up on a slice is via a data dependency, which implies a use of the variable. Therefore, every test suite that satisfies checked coverage also satisfies all definitions coverage, under the premise that uses in the test suite are included and no definitions are made in control flow manipulating statements.

2.3. Infeasible requirements

Checked coverage might impose *infeasible test requirements*, which are requirements that cannot be fulfilled by a test. If such infeasible requirements exist, it is impossible to reach 100% checked coverage for a project.

For example, similar to statement coverage, unreachable code imposes an infeasible test requirement for checked coverage. Because unreachable code cannot be executed, it cannot end up on a backward slice.

Unnecessary code leads to another instance of infeasible requirements. Unnecessary codes are statements that cannot contribute to the programme behaviour, for example, definitions of variables that are never used (before they are redefined) and are not part of the programme's output. Although this code can be exercised, there is no way this code can end up on a backward slice because it is never used for a computation that influences the programme's results.

Other infeasible requirements include computations that lead to nonvisible behaviour, for example, they lead to another visible behaviour not being carried out. An example would be a

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

variable that controls whether a branch is taken or not, but only taking the branch manipulates the results. Computations leading to not taking the branch cannot be on a backward slice as there is no behaviour to slice from. An example for such an instance is given in Section 4.

2.4. Implementation

For the implementation of our approach, we use Hammacher's JAVASLICER [6] as a dynamic slicer for Java. The slicer works in two phases: in the first phase, it traces a programme 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 programme 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 JVM can be instrumented. (There are a few exceptions however that are explained in Section 4.) Because 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 [7, 8] algorithm and stores it to disc.

To compute the slices, we use an adapted algorithm from Wang and Roychoudhury [9, 10]. The data dependencies are calculated by iterating backwards through the trace. For the control dependencies, the control flow graph 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 three 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. Afterwards, all test classes are traced separately. This is a performance optimization, because 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, because the slicer supports building 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.

2.5. Usage scenarios for checked coverage

Our proposed usage of checked coverage for developers is to *identify deficiencies in the checks of the test suite*. To achieve this, there are two ways to apply checked coverage. In the first scenario, the developer wants to identify parts of the code that are not checked by assertions of the test suite. In the second scenario, the developer wants to identify tests that do not check their results well enough. In both scenarios, we assume that the developers are only concerned with the checks' thoroughness, and that they are not using checked coverage to find uncovered code, which can be more easily achieved using regular statement coverage.

2.5.1. Identifying unchecked programme parts. For this usage scenario, the statement coverage score and the checked coverage score is determined for every single class of the project. Then, the difference between the statement and the checked coverage is computed. A difference between these values for a class, or in general a code fragment, indicates that the results of this fragment are poorly checked although they are executed by at least one test. By identifying the exact statements that are included by statement coverage but not by checked coverage, computations that are not properly checked can be identified. This can be performed by simple set intersection and can

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

be carried out by the tool computing the checked coverage. To improve the test suite, the tests that execute unchecked computations can be identified and new checks can be introduced. Alternatively, a newly written test can be introduced.

2.5.2. Identifying tests with weak checks. In this usage scenario, the statement and the checked coverage score is determined for every test case. A difference between the coverage and the checked coverage score for a test case might point to tests that do not check their covered computations well enough. However, in this case, the test might also cover parts of the programme it is not intended to check, for example, in its setup code. Therefore, it might be necessary to focus on parts of the programme, for example, a package or a collection of classes. The statements in the selected parts that are covered but not included by checked coverage point to computations that are not checked by the tests. A test case can be improved by introducing additional checks for the unchecked parts as indicated by the difference between statement and checked coverage.

2.5.3. Proposed usage. Our proposed usage scenario for checked coverage is a tool that can be used by developers to improve the quality of a test suite's checks. We do not recommend it as a tool for management to evaluate developers writing the tests because, similar to other metrics, checked coverage can be fooled. One way to fool checked coverage to obtain a higher score would be to introduce weak or nonsensical checks. For example, asserting that an object is equal to itself would include the parts of the programme that contribute to produce the checked object, but the check might not improve the quality of the test suite. In general, this is a drawback of checked coverage compared with mutation testing. Checked coverage only assesses whether results are checked but not the quality of the checks themselves. Mutation testing, on the other hand, also takes the quality of the checks into account because weak checks would not be able to detect many mutations.

However, mutation testing can also be fooled. This was observed by Aaltonen *et al*. [11] in a study where they used the mutation score to assess the quality of test suites produced by students and used it for grading. Some of the students introduced meaningless code that leads to many mutations that are easy to detect, and thus to a high mutation score for the project.

2.6. Checked coverage metrics

Checked coverage follows the intuition that every property that is covered by the test suite should also be checked by it. Our original definition of checked coverage can be seen as an extension to statement coverage. Checked coverage requires that every statement that is covered also contributes to a computation that is checked by the test suite. This concept can also be carried over to other metrics, by imposing the requirement that tests fulfilling requirements of the metric also properly check the results.

Definition 12 (Checked metric)
A coverage metric is extended to its checked version by imposing the additional requirement that every statement that contributes to fulfilling a requirement, imposed by the original metric, is also on a dynamic backward slice from a check of the test that fulfils the requirement.

By using this definition, we can extend existing coverage metrics to their checked versions. For example, we can extend *branch coverage* to *checked branch coverage*. Original branch coverage requires that every branch in the programme is taken, that is, every conditional statement is evaluated to `true` and `false`. A single test requirement demands that a specific branch is exercised by at least one test. Checked branch coverage extends this requirement by additionally demanding that the statement that controls the specific branch is on a backward slice from a check of the tests. Notice that this refers to a special instance of the statement. During the run of the test suite, there might be different instances where this statement is executed and different execution instances can lead to different branches being taken. Thus, it is required that an instance that leads to the required branch is on the backward slice.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

3. EVALUATION

In the evaluation of our approach, we were interested whether checked coverage can help in *improving existing test suites* of mature programmes. To this end, we computed the checked coverage for seven open-source programmes that have undergone several years of development and come with a JUnit test suite. We manually analysed the results, and found examples where the test suites can be improved to more thoroughly check the computation results (see Section 3.3). We also detected some limitations of our approach, summarized in Section 4.

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 3.4), 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.

3.1. Evaluation subjects

As evaluation subjects, we chose seven open-source projects that come with a reasonably good test suite. These subjects are listed in Table I. The subjects come from different application areas (Column 2), and we took a revision from the version control system (Column 3)—except for JTOPAS, which was taken from the Software-artifact Infrastructure Repository [12].

The properties of the projects' test suites are summarized in Table II. The table shows the size of the test suites (Column 2), which ranges from around 3000 lines of code for JTOPAS and BARBECUE up to 50,000 lines for JODA-TIME. The number of tests is given in Column 3, where 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. Again, JODA-TIME has the largest test suite with over 2700 tests. The last column lists the time needed to run the test suite, which ranges from 2 s for JTOPAS to 37 s for JODA-TIME.

Table I. Description of subject programmes.

Project name	Description	Version	Programme size (LOC)
ASPECTJ.ajdt.core	AOP extension to Java	cvs: 2010-09-15	28,476
BARBECUE	Bar code creator	svn: 2007-11-26	4837
COMMONS-LANG	Helper utilities	svn: 2010-08-30	18,452
JAXEN	XPath engine	svn: 2010-06-07	12,438
JODA-TIME	Date and time library	svn: 2010-08-25	26,582
JTOPAS	Parser tools	1.0(SIR)	2031
XSTREAM	XML object serialization	svn: 2010-04-17	15,266

Lines of code (LOC) are non-comment, nonblank lines as reported by `sloccount`.

Table II. Description of subject programmes' test suites.

Project name	Test code size (LOC)	Number of tests	Test suite run-time (s)
ASPECTJ.ajdt.core	32,942	339	11
BARBECUE	3293	153	3
COMMONS-LANG	29,699	1787	33
JAXEN	8418	689	11
JODA-TIME	50,738	2734	37
JTOPAS	3185	128	2
XSTREAM	16,375	1113	7

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Table III. Checked coverage, statement coverage and mutation score.

Project name	Checked coverage (%)	Statement coverage (%)	Mutation score (%)
ASPECTJ.ajdt.core	14	36	28
BARBECUE	22	39	6
COMMONS-LANG	73	99	80
JAXEN	63	97	48
JODA-TIME	50	81	54
JTOPAS	57	77	61
XSTREAM	28	94	72

3.2. Results

For our experiments, we used our implementation of checked coverage as described in Section 2.4, and to compute the mutation score, we used the JAVALANCHE framework. JAVALANCHE was developed with a focus on automation and scalability. To this end, it applies several optimizations, such as selective mutation [13, 14] and mutant schemata [15], 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 [16, 17].

Table III 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.ajdt.core and BARBECUE. For ASPECTJ.ajdt.core, the results are lower, because we only used a part of the test suite, to run our experiments in a feasible amount of time (about 1 h 22 min to compute the checked coverage and about 20 h for mutation testing). 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 66% points, 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.

3.3. 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 where checked and regular coverage differ, 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%.

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 5. 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.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

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

Figure 5. Another test with insufficient outcome checks.

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

Figure 6. A method where the return value is not checked.

Figure 6 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.

3.4. Disabling oracles

In our first experiment, we have seen cases were a test suite covers parts of the programme, but does not check 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 with other metrics that measure the quality of a test suite?*

The goal of the evaluation is to assess the sensitivity of checked coverage with respect to the *oracle quality* of a test suite. However, there is no objective measure for oracle quality. Thus, we chose to remove oracles from the original test suite to simulate a decreased oracle quality. 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, we completely remove the line that contains it. For each project, we generated different versions of the test suite, and we computed the statement coverage and checked coverage and mutation score for them. From the original test suites, we produced versions with 0, 25, 50, 75 and 100% of the checks removed. When removing a fraction of a test suite's checks, there are different possibilities which checks to remove. Thus, we averaged over 100 different versions for each fraction of removed checks, and for each version, we randomly selected the checks to remove. For the test suites with all checks (0% removed) and no checks (100% removed), however, we produced only one version because there is only one way to remove no or all checks.

The results of our experiment are shown in Figure 7. For each of our subject programmes, there is a plot that shows the statement coverage score, checked coverage score and mutation score for the test suites with all assertions enabled (0% removed), and with 25, 50, 75 and 100% of the assertions removed, respectively.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

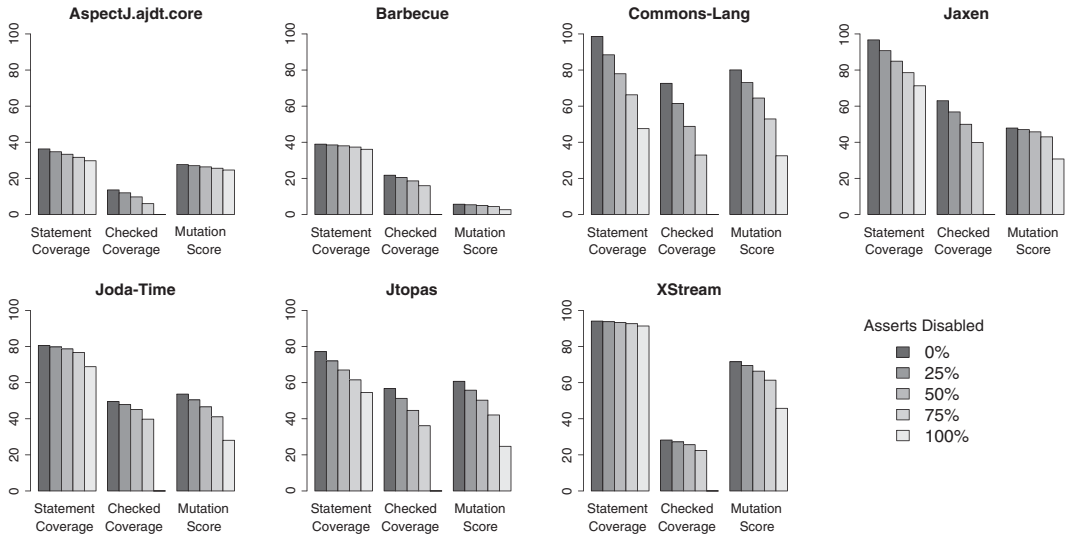

Figure 7. Coverage values for test suites with removed assertions.

For all projects, all metric scores decrease with a decreasing number of assertions. However, the absolute coverage score for the different metrics varies. When we consider the test suites with all assertions enabled, statement coverage reaches the highest coverage score for all projects. The checked coverage score is higher than the mutation score for two projects (BARBECUE and JAXEN). This might be an indicator for weak checks, because the checks cover computations but are not able to detect mutations for them. For two other projects, the mutation score is higher than the checked coverage score (ASPECTJ.ajdt.core and XSTREAM), and for the rest, they are roughly the same.

To check whether the coverage differences resulting from removed assertions are statistically significant, and are not due to the variance within the samples, we applied a Kruskal–Wallis one-way analysis of variance. By using this test, we could reject the null hypothesis —which states that there is no difference between the median coverage values for the different test suites— for the all coverage metrics (checked coverage, statement coverage and mutation score). After this null hypothesis was rejected, we investigated whether the coverage differences between the test suites (0–100% of assertions removed) are statistically significant. To this end, we applied a Wilcoxon rank-sum test with a Bonferroni correction for the p-values. The results showed that the differences between the coverage values for the test suites are statistically significant at a 95% confidence level for all metrics.

All test quality metrics decrease with oracle decay.

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 8 shows the results for the seven subject programmes.

For statement coverage, the decrease values are the lowest for all projects. This comes as 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 behave alike for four projects (BARBECUE, JODA-TIME, JTOPAS and XSTREAM) and test suites from 0% checks removed to 75% removed. For the other projects, there is a greater decrease for the checked coverage than for the mutation score (ASPECTJ.ajdt.core, COMMONS-LANG and JAXEN). On average, when 75% of the tests are removed, the checked coverage score decreases by 36%, whereas the mutation score only decreases by 29%.

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

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

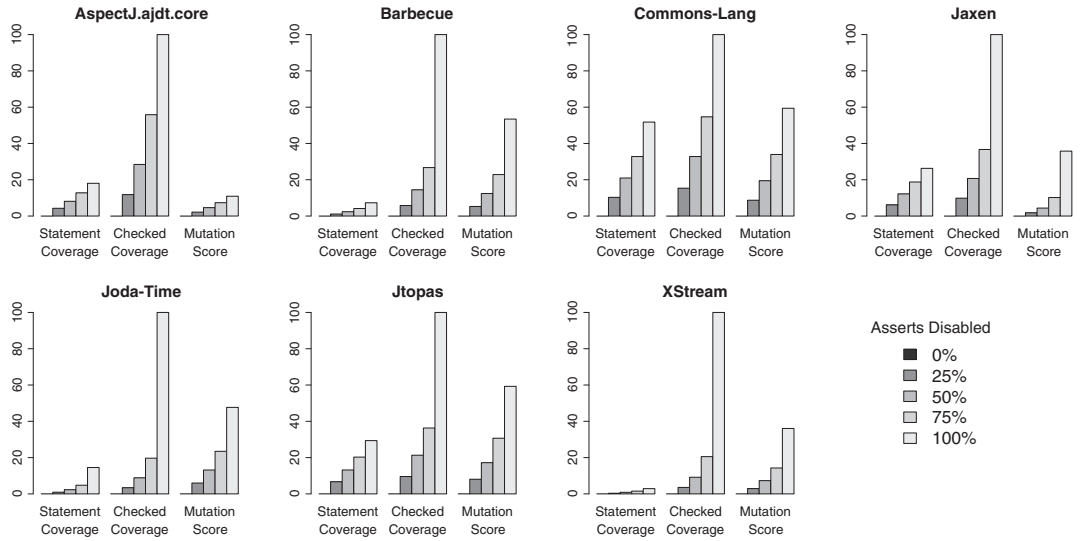

Figure 8. Decrease of the coverage values relative to the coverage values of the original test suite.

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. Because we are interested in *poor* rather than nonexistent oracles, the results obtained for the decays of 25–75% are much more meaningful.

3.5. Explicit and implicit checks

As mutation testing is 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 run-time system. For modern object-oriented and restrictive programming languages such as Java, many mutations are detected through these implicit run-time checks.

JAVALANCHE, the mutation testing tool employed in this study, uses the distinction made by JUnit 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 IV 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 explicit checks (Column 3). The remaining mutations are detected by implicit 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 (e.g. in the `verify()` method of a mocking framework)—that are not removed—and the JUnit `verify()` method cause this 21%. These methods are classified as explicit checks because they cause an `AssertionError` to be thrown.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Table IV. Mutations detected by explicit and implicit checks.

	Original test suite		All checks removed	
Project name	**Total**	**Explicit%**	**Total**	**Explicit%**
ASPECTJ.ajdt.core	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-LANG	13415	68	3377	16
XSTREAM	7764	35	5156	5
Total	**45564**	**55%**	**24408**	**21%**

One could argue that it does not matter how a mutation is being found—after all, the important thing is that it is found at all. Keep in mind, however, that implicit checks are not under control by the programmer. A mutation score coming from implicit checks thus reflects the *quality of the test inputs* because they bring the system into a state where a mutation can be detected by run-time checks. Also, the run-time system will only catch the most glaring faults (say, null pointer dereferences or out of bound accesses), but likely 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.

3.6. Performance

Table V shows the run-time 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, by 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 4).

The last column gives the time needed for mutation testing the programmes with JAVALANCHE. The run-time of mutation testing depends on two factors; the number of mutants and the time needed to run the test suite. With different choices of which operators to use and on the number of mutations to apply, the time that is needed for mutation testing a project varies. For example, to execute 10^4 mutants for a programme with a test suite that needs on average 10 s to execute on a mutant 10^5 s are needed, which is more than one day of computing time. For this experiment, we used JAVALANCHE in its standard settings as described in an earlier paper [17]. When we compare

Table V. Run-time to compute the checked coverage and the mutation score.

	Checked coverage			
Project name	**Trace**	**Slice**	**Total**	**Mutation test**
ASPECTJ.ajdt.core	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-LANG	0:32:07	0:40:37	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

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

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, however, 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.

In terms of performance, checked coverage is on par with advanced mutation testing tools like Javalanche

4. 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 because of 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. This code cannot be accessed by the tracer as it only sees the bytecode of the classes loaded by the JVM. Regular programmes rarely use this feature. In the Java standard library however, there are many methods that use the Java Native Interface. 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 become 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. Moreover, there are further situations where potential dependencies are missed. For example, when data are written and read from files or dependencies that arise from network communication.

A frequently used practice 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 9). 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.

Another limitation introduced by our approach are computations that lead to a branch not being taken. In Figure 10, for example, we have the boolean flag `stateOK`. 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.

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

Figure 9. A common JUnit pattern to check for exceptions.

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

Figure 10. Statements that lead to not taking a branch.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

5. CHECKING FOR EXCEPTIONS

In the previous section, we saw that exceptions impose a limitation on checked coverage. Many of the tests that check for exceptional behaviour do this by providing inputs under which the programme is expected to throw an exception. These tests pass when the exception is thrown and fail when the expected exception is not thrown. However, the tests do not explicitly check for the exception. They rather test for the exception implicitly by failing when the expected exception is not thrown. In the expected case, when the exception is thrown, the tests do not check for properties of the exception explicitly. Thus, the computations leading to the exception do not contribute to the checked coverage.

To include the behaviour that is checked for by these tests, we extended our approach such that exceptions that propagate to and are caught in the test suite also contribute to checked coverage.

5.1. Implementation

To include tests that check for exception into checked coverage, we manipulate test suites in such a way that method calls to log an exception are inserted for every caught exception. Then, we compute a backward slice from all invocations of this exception logging method and included the additionally covered statements into the regular checked coverage.

Figure 11 shows an example for our instrumentation. Because the original test ignores the exception, a logging call (shown in bold) is inserted that can be used for a backward slice from the exception.

Note that JUnit 4 provides support to test for exceptions via a parameter given to the test annotation. Our implementation also supports this type of checks for exceptions. However, none of the projects used in our evaluation uses these annotations yet.

5.2. Results

We were interested in how widespread this practice of checking for exceptions is, and how much these tests can contribute to checked coverage. To this end, we instrumented the test suites of our subject projects as described previously and computed the backward slices from the caught exceptions. Then, we computed three variants of the checked coverage score: (1) checked coverage only for the exceptions; (2) original checked coverage; and (3) extended checked coverage, which is the union of the statements included for regular checked coverage and the statements from the backward slices of the exceptions. Finally, we computed the increase that was obtained by including checks for exceptions.

Table VI summarizes our results. Each line shows the result for one project. First, the number of tests that catch exceptions is shown (Column 2). Then, it presents the checked coverage score for only considering the backward slices from the exceptions (Column 3). For comparison, the regular checked coverage score is shown (Column 4). Then, the score for extended checked coverage is given (Column 5). Finally, the increase that is achieved by including the exceptions is displayed (Column 6), which is the difference between regular and extended checked coverage.

The exceptions that are explicitly caught by the test suites range from 7 for JTOPAS up to 1284 for JODA-TIME. Notice that not every test catching an exception represents the pattern to check for exceptions that was described previously. Sometimes, the catch statements are just meant to react on

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

Figure 11. An instrumented test that checks for exceptions.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Table VI. Including tests for exceptions in checked coverage.

Project name	Tests catching exceptions	Checked coverage			Increase through exceptions (pp)
		Only exceptions (%)	Regular (%)	Regular and exceptions (%)	
ASPECTJ.ajdt.core	37	0.07	13.61	13.61	0.00
BARBECUE	35	2.93	21.74	23.08	1.33
JAXEN	110	17.04	63.01	64.41	1.41
COMMONS-LANG	720	10.54	72.63	78.11	5.48
JODA-TIME	1284	22.06	49.51	53.38	3.87
JTOPAS	7	0.00	56.72	56.72	0.00
XSTREAM	84	17.58	28.21	33.91	5.70
Average	**325**	**10.03**	**43.63**	**46.17**	**2.54**

a failing computation, but this is not expected by the test. For example, this is the case for the two projects (ASPECTJ.ajdt.core and JTOPAS) where including the exceptions for checked coverage has no effect on the checked coverage score.

To check whether a test suite checks well for exceptional behaviour, we can compute checked coverage on the basis of checks for exceptions (Column 3), that is, only backward slices from caught exceptions are computed. Here, we obtain a coverage score from 0% for ASPECTJ.ajdt.core and JTOPAS up to 22% for JODA-TIME. This indicates that the JODA-TIME test suite checks well for exceptional behaviour, whereas the other two fail to do so.

Furthermore, we were interested how including the slices from caught exceptions increases checked coverage for our projects. Again, we have the two projects (ASPECTJ.ajdt.core and JTOPAS) that do not check for exceptional behaviour where including exceptions has no effect. For the other projects, however, including checks for exception increases the checked coverage score. The highest increase was observed for COMMONS-LANG and XSTREAM with over 5% points, and on average for all projects, the increase is around 2.5% points.

Including tests for exceptions increases checked coverage on average by 2.5 percent points.

6. 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 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 [18], but as our results show, even programmes 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.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

7. RELATED WORK

7.1. Coverage metrics

During structural testing, a programme is tested using knowledge of its internal structures. Hereby, one is interested in the *quality* of the developed tests, and how to *improve* them 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 [19–22]. 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* [23] also relate definitions and uses of variables. These techniques consider the relation between all defined variables inside the programme 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 programme are checked. Thus, they *do not assess oracle quality* of a test suite.

7.2. Mutation testing

A technique that aims at checking the quality of the oracles is *mutation testing*. Originally proposed by Richard Lipton [24, 25], mutation testing seeds artificial defects, as defined by *mutation operators*, into a programme 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 programme. If a mutation is not detected by the test suite, similar defects might be in the programme, which 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 programme.

7.3. Programme slicing

Static programme slicing was originally proposed by Weiser [2, 3] as a technique that helps the programmer during debugging. Korel and Laski introduced dynamic slicing that computes slices for a concrete programme run. Furthermore, different slicing variants have been proposed for programme comprehension; *Conditioned Slicing* [26] is a mix between dynamic and static slicing. It allows some variables to have a fixed value, whereas others can take all possible values. *Amorphous Slicing* [27] requires a slice only to preserve the semantics of the behaviour of interest, whereas syntax can be arbitrarily changed, which allows to produce smaller slices.

Besides its intended use in debugging, programme slicing has been applied to many different areas [28, 29]. Examples include minimization of generated test cases [30], automatic parallelization [28, 31] and the detection of equivalent mutants [32].

7.4. State coverage

The concept closest to checked coverage is *state coverage* proposed by Koster and Kao [33]. 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

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

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 [33], a proof of concept based on a static slicer and one small experiment are presented. A second short paper [34] describes an implementation based on *taint analysis* [35], but no experimental evaluation is provided.

8. 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 (as described in Section 2) and human labour (because of equivalent mutants). By assessing which parts of the covered computation are actually checked by oracles, *checked coverage* covers the middle ground, providing a straightforward means to assess oracle quality at reasonable efficiency—and even more sensitive to missing oracles 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 programmes 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, run-time checks for preconditions and post-conditions 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 run-time checks are conducted.
- **Defensive programming.** Programmes 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’ behaviour—a separation that 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 programme.

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/

ACKNOWLEDGEMENTS

This project has been funded by Deutsche Forschungsgemeinschaft, grant Ze509/5-1. Yana Mileva, Nikolai Knopp, Florian Großand 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. Schuler D, Zeller A. Assessing oracle quality with checked coverage. *ICST ’11: Proceedings of the 4th International Conference on Software Testing, Verification and Validation*, Berlin, Germany, 2011; 90–99.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

2. Weiser M. Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method. *Ph.D. Thesis*, University of Michigan, Ann Arbor, USA, 1979.
3. Weiser M. Program slicing. *Transactions on Software Engineering* 1984; **10**(4):352–357.
4. Korel B, Laski J. Dynamic program slicing. *Information Processing Letters* 1988; **29**(3):155–163.
5. Ammann P, Offutt J. *Introduction to Software Testing*, 2edn. Cambridge University Press: Cambridge, UK, 2008.
6. Hammacher C. Design and implementation of an efficient dynamic slicer for Java. *Bachelor's Thesis*, Saarland University, Saarbrücken, Germany, 2008.
7. Nevill-Manning CG, Witten IH, Maulsby D. Compression by induction of hierarchical grammars. *DCC '94: Proceedings of the 4th Data Compression Conference*, Snowbird, Utah, USA, 1994; 244–253.
8. Nevill-Manning CG, Witten IH. Linear-time, incremental hierarchy inference for compression. *DCC '97: Proceedings of the 7th Data Compression Conference*, Snowbird, Utah, USA, 1997; 3–11.
9. Wang T, Roychoudhury A. Using compressed bytecode traces for slicing Java programs. *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, Edinburgh, UK, 2004; 512–521.
10. Wang T, Roychoudhury A. Dynamic slicing on Java bytecode traces. *Transactions on Programming Languages and Systems* 2008; **30**(2):10:1–10:49.
11. Aaltonen K, Ihantola P, Seppälä O. Mutation analysis vs. code coverage in automated assessment of students' testing skills. *SPLASH/OOPSLA '10: Companion to the 25th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Reno, USA, 2010; 153–160.
12. Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering* 2005; **10**(4):405–435.
13. Offutt AJ, Lee A, Rothermel G, Untch RH, Zapf C. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 1996; **5**(2):99–118.
14. Zhang L, Hou SS, Hu JJ, Xie T, Mei H. Is operator-based mutant selection superior to random mutant selection? *ICSE '10: Proceedings of the 32nd International Conference on Software Engineering*, Cape Town, South Africa, 2010; 435–444.
15. Untch RH, Offutt AJ, Harrold MJ. Mutation analysis using mutant schemata. *ISSTA '93: Proceedings of the 8th International Symposium on Software Testing and Analysis*, Cambridge, MA, USA, 1993; 139–148.
16. Schuler D, Dallmeier V, Zeller A. Efficient mutation testing by checking invariant violations. *ISSTA '09: Proceedings of the 18th International Symposium on Software Testing and Analysis*, Chicago, IL, USA, 2009; 69–80.
17. Schuler D, Zeller A. (Un-)Covering equivalent mutants. *ICST '10: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, Paris, France, 2010; 45–54.
18. Andrews JH, Briand LC, Labiche Y. Is mutation an appropriate tool for testing experiments? *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, St. Louis, MO, USA, 2005; 402–411.
19. Huang JC. An approach to program testing. *Computing Surveys* 1975; **7**(3):113–128.
20. Ntafos SC. A comparison of some structural testing strategies. *Transactions on Software Engineering* 1988; **14**(6):868–874.
21. Weyuker EJ, Weiss SN, Hamlet RG. Comparison of program testing strategies. *TAV 1991: Symposium on Testing, Analysis, and Verification*, Victoria, British Columbia, Canada, 1991; 1–10.
22. Frankl PG, Weiss SN, Hu C. All-uses versus mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software* 1997; **38**:235–253.
23. Rapps S, Weyuker EJ. Data flow analysis techniques for test data selection. *ICSE '82: Proceedings of the 6th International Conference on Software Engineering*, Tokyo, Japan, 1982; 272–278.
24. Offutt AJ, Untch RH. Mutation 2000: Uniting the orthogonal. *Mutation '00: Proceedings of the 2nd International Workshop on Mutation Analysis (Mutation testing for the new century)*, San Jose, CA, USA, 2001; 34–44.
25. DeMillo RA, Lipton RJ, Sayward FG. Hints on test data selection: Help for the practicing programmer. *Computer* 1978; **11**(4):34–41.
26. Lucia AD, Fasolino AR, Munro M. Understanding function behaviors through program slicing. *WPC '96: Proceedings of the 4th International Workshop on Program Comprehension*, Berlin, Germany, 1996; 9–18.
27. Harman M, Danicic S. Amorphous program slicing. *WPC '97: Proceedings of the 5th International Workshop on Program Comprehension*, Dearborn, MI, USA, 1997; 70–79.
28. Tip F. A survey of program slicing techniques. *Journal of Programming Language* 1995; **3**(3):121–189.
29. Harman M, Hierons RM. An overview of program slicing. *Software Focus* 2001; **2**(3):85–92.
30. Leitner A, Oriol M, Zeller A, Ciupa I, Meyer B. Efficient unit test case minimization. *ASE '07: Proceedings of the 22nd International Conference on Automated Software Engineering*, Auckland, New Zealand, 2007; 417–420.
31. Hammacher C, Streit K, Hack S, Zeller A. Profiling Java programs for parallelism. *IWMSE'09: Proceedings of the 2nd International Workshop on Multi-Core Software Engineering*, Vancouver, BC, Canada, 2009; 49–55.
32. Hierons R, Harman M. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability* 1999; **9**(4):233–262.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

33. Koster K, Kao D. State coverage: A structural test adequacy criterion for behavior checking. *ESEC/FSE '07: Proceedings of the 11th European Software Engineering Conference held jointly with 15th International Symposium on the Foundations of Software Engineering*, Dubrovnik, Croatia, 2007; 541–544.
34. Koster K. A state coverage tool for JUnit. *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, Leipzig, Germany, 2008; 965–966.
35. Clause JA, Li W, Orso A. Dytan: A generic dynamic taint analysis framework. *ISSTA '07: Proceedings of the 16th International Symposium on Software Testing and Analysis*, London, UK, 2007; 196–206.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr