This is the post-print of a paper that has been published with the following DOI: http://dx.doi.org/10.1109/ICSTW.2013.49

On Adequacy of Assertions in Automated Test Suites: An Empirical Investigation

Junji Zhi[1], Vahid Garousi[2, 3]

1: Department of Computer Science
2: Department of Electrical and Computer Engineering
Software Quality Engineering Research Group (SoftQual)
University of Calgary, Calgary, Alberta, Canada
{zhij, vgarousi}@ucalgary.ca

3: Department of Information Systems
Informatics Institute
Middle East Technical University, Ankara, Turkey
vahid@metu.edu.tr

***Abstract* - An integral part of test case is the verification phase (also called 'test oracle'), which verifies program's state, output or behavior. In automated testing, the verification phase is often implemented using test assertions which are usually developed manually by testers. More precisely, assertions are used for checking the unit or the system's behavior (or output) which is reflected by the changes in the data fields of the class under test, or the output of the function under test. Originated from human (testers') error, test suites are prone to having inadequate assertions. The paper reports an empirical study on the Inadequate-Assertion (IA) problem in the context of automated test suites developed for open-source projects. In this study, test suites of three active open-source projects have been chosen. To investigate IA problem occurrence among the sampled test suites, we performed mutation analysis and coverage analysis. The results indicate that: (1) the IA problem is common among the sampled open-source projects, and the occurrence varies from project to project and from package to package, and (2) the occurrence rate of the IA problem is positively co-related with the complexity of test code.**

***Keywords* - Test assertions, test-suite quality, quality of test oracles, adequacy of test oracles, automated testing, mutation testing, state coverage, empirical case-study**

I. INTRODUCTION

A report from Microsoft indicates that 79% of developers write automated unit tests [1]. Data from this report indicates that unit testing (also called "developer testing" [2]) is a technique widely used to ensure the quality of the code written by developers. Studies have shown that substantial amount of manual efforts are spent into developing unit test code [3]. These test cases can contain faults on their own due to the error-proneness of human activity spent to develop/maintain them. Similar to the quote "Watching the watchman", the quality of automated test scripts should be carefully assessed and assured. If the test suites developed by testers are faulty themselves, they cannot guard the system under test (SUT) against potential faults, and using them will bring little value to the project. Similar to production software, test scripts require adequate verification and validation to ensure that they perform the intended testing activity correctly. Thus, "testing a test" is an important aspect that requires the development of appropriate techniques and tools.

A typical unit test (e.g., in the context of xUnit frameworks) consists of four phases [4]: (1) test setup, (2) exercising the system under test (SUT), (3) verification, and (4) teardown. The verification phase (also called test oracle, assertion or state verification [4]) is an integral part of testing. It determines whether a test case passes or fails by comparing the expected program outputs or behavior with the actual ones. In practice, state verification is usually implemented using assertions, which is a specific mechanism supported by almost all major unit test frameworks, e.g., `assertEquals(expected, actual)` in JUnit. Using assertions in test cases is not limited to unit testing only. In other types of testing (e.g., system, integration or even non-functional), assertions are also widely used (e.g., asserting that a GUI textbox has a certain value in system testing).

To study the adequacy of assertions in test suites, the concepts of program and object state are paramount. Program state is the snapshot (one Cartesian point) in the n-dimensional space of all values of the program's *n* variables. For example, if a program has 3 integer variables, the program state space will be a 3D space of all integer values and the program state in each instant during its execution. Similarly, object state is defined as a Cartesian point in the n-dimensional space of all values of the object's *n* member variables. As this n-dimensional space is usually large, testers may forget to write assertions to check all value updates. In other words, testers may fail to assert all the changes that have occurred in the program or object state after the system is "exercised" during testing. For example, consider an expression-parser class that contains a stack object which assists in implementing the parsing function. During the verification phase, it would be important to verify that this stack object is manipulated and changed as it should be. Such supporting data inside the object under test is a determinant of the object's state. Testers may only focus on asserting a few member variables of the expression-parser object and, due to negligence, miss writing additional necessary assertions to ensure the correct manipulation of other member variables of the main object or of the internal stack object.

Similar to Koster and Kao's study [5, 6], we define the sufficiency of test assertions based on the assumption that every update to the execution state must eventually be

This is the post-print of a paper that has been published with the following DOI: http://dx.doi.org/10.1109/ICSTW.2013.49

verified in the assertions. In principle, assertions should verify the correctness of all updates to the object/program state, otherwise the strength of the test oracles is considered not enough to guard the program against faults. In Section III, a formal definition of the *Inadequate Assertion (IA)* problem and a real code example will be provided.

We have conducted an empirical case-study to analyze the scale of the Inadequate Assertion (IA) problem and this paper reports our findings. Case-study objects (test suites under study) are sampled from three active open-source projects. In our study, we use mutation analysis and coverage analysis to assess the quality of the test suites. Finally we draw conclusions based on our quantitative results. The results show that (1) IA problem is common among the sampled open-source automated test suites and the scale of the problem ranges from 55% to 65% (e.g., 55% of the sampled unit test methods in the Log4j project suffer from the IA problem), (2) as one would expect, the occurrence of the problem varies from project to project, from package to package, and from class to class, and (3) the IA occurrence rate is positively co-related with the complexity of test code.

The goal of this case study is to characterize the scale of the IA problem and the factors contributing to this problem. As a result, the contributions of this paper are two-fold:

- Evidence on non-negligible existence of the IA problem by quantitatively showing the ratio of this problem in the test suites of three real-world open-source projects
- A preliminary investigation (root-cause analysis) on potential factors (e.g., code complexity) which would (could) lead to the IA problem

The commonness of IA problem reminds us that, to ensure a high-quality test code, we should concern not only about generating efficient test input to cover as much of the SUT as possible, but also about the adequacy of assertions in test code.

The remainder of this paper is organized as follows. Section II discusses the related work. In Section III, we present a formal definition for the IA problem and an example of the problem. Section IV describes the case-study design and setup. Study results are presented in Section V. Finally, conclusions and future work are discussed in Section VI.

II. Related Work

The works related to this study are those which either (1) study the association of test assertions with program states, or (2) provide assessment or assurance metrics for test-code quality which involve assertions. By a literature search for related works in those areas, we found the following studies and discuss them as below.

In a popular book on xUnit test patterns [4], Meszaros discussed a list of patterns and symptoms (code and test "smells") which are related to unit testing using xUnit test frameworks (e.g., JUnit). Meszaros classified state verification into two categories: procedural sate verification and expected object. The former, in general sense, involves writing a sequence of test assertions to pick apart the final state of the SUT and verify that it is as expected. Verifications of type "expected object" define a single object to represent the expected state which is then used to be compared with the actual output in a single assertion. In this case, testers need to ensure that all data members of the expected object will be initialized and the updates in all data members during the test run will be checked in the assertion. As a result, even though there is only one assertion, it is sufficient to detect incorrect program state. However, in this paper, we only discuss the problem in procedural sate verification since it is more intuitive than expected object method and used more commonly among testing community.

Tools have been proposed to assist testers in developing or augmenting test assertions. For example, Xie [13] proposed a tool called Orstra, which uses regressions to automatically generate test oracles. These oracles can guard the problem against regression faults. Song et al. [9] introduced an Eclipse plug-in, called UnitPlus, to assist developers in writing unit tests. Their approach distinguishes *state-modifying* (e.g., setter functions) and *observer* methods (e.g., getter functions) and then identifies the affected and accessed fields for each of these methods. Using such an analysis, the tool recommends relevant observer-methods to call and assert object variables when testers are writing test oracles. However, one small difference between their approach and ours is that, their analysis is based on static code analysis. Using their tool UnitPlus will consider all the fields affected by the state-modifying method, including the ones only being modified in some conditions (e.g., definition code inside an 'if-else' block). For example, if the field F is only modified inside an "if" block and not anywhere else in the program, when the "if" predicate is false during execution, the program will not modify F. Under such circumstance, it is not necessary to check the state of F in test assertions. This is considered in our definitions.

Koster and Kao [8] proposed state coverage as a new test adequacy criterion for behavior checking. Different from coverage-based criteria or mutation adequacy criteria, state coverage measures the checks of program behavior in test suites. In more recent works, Vanoverberghe et al. [5], [6] proposed two granularities of state coverage: object insensitive and object sensitive. These two criteria differ in granularity of defining a state update. Object-insensitive state-coverage only tracks code location while object-sensitive state-coverage tracks more information. Their results showed that the latter criterion is more fine-grained than the former in terms of fault-detection effectiveness. Schuler and Zeller [12] proposed checked coverage which is computed using dynamic analysis or program slicing. Their result showed that checked coverage was a convincing indicator for test-oracle quality and even more sensitive than mutation testing.

This is the post-print of a paper that has been published with the following DOI: http://dx.doi.org/10.1109/ICSTW.2013.49

In contrast to proposing a new test-oracle-quality metric, our study focuses on the problem of assertion inadequacy and its scale in open-source test suites. Unlike [6], the metric used in our study is based on field definition and usage instead of code location. Nevertheless, our study can offer empirical evidence support for state coverage because inadequate assertion problem is due to lacking field update checking.

Knauth et al. [7] reported a similar case study on the completeness of contracts. Similar to our findings, their study results indicated the variance of incomplete assertion problem. Their approach is different from ours for three reasons. (1) Their target was contracts that are often implemented using assertions. In contrast, we analyzed test assertions. (2) They generated test cases based on the contracts instead of evaluating the existing ones while we utilized the existing test cases in the open-source projects. (3) Their approach to determine distinguishable (killed) mutants is slightly different from ours. They ran the random testing scenarios and decided a mutant is distinguishable if it changed any state of the program. In contrast, in our study we combine the mutation location and the statement coverage of test cases to determine whether that mutant should be detected by those test cases or not.

Xie and Memon [11] conducted a study on how GUI test oracles impact fault detection effectiveness and cost of a test case. Similar to ours, their results suggested that writing oracles to check the entire state, instead of checking one single active window, is preferred to achieve high effectiveness of the test case. There are several differences between their study and ours: (1) They restricted their study objects to oracles that test the GUI only while we consider assertions in automated test suites which could be unit or GUI testing levels; (2) In terms of the faults seeded into the SUT(s), they manually developed them based on real-life bug "templates" while we used mutation operators to automatically generate faults.

Staats et al. [10] extended the previous foundational theory and re-considered the role of test oracles. Specifically, they defined general oracle properties based on the interrelationships among program, test cases and oracles. They proposed the concept of "sound" oracle and pointed out an oracle that only checks a sub-set of the program states is likely to "miss any faults manifested in the variables". This insight is also re-validated by our study in this paper. Since assertions are only one type of methods to develop test oracles, the IA problem and it consequences provide appropriate evidences for theoretical work such as [10].

III. Definition of the IA Problem and an Example

A. Definition

In order to define the Inadequate-Assertion (IA) problem, we borrowed the definitions of *state-modifying* and *observer* methods from Song et al.[9].

Definition 1: A non-final field *F* (defined below) in a class *C* is identified to be *defined* by a *state-modifying* method *M* in a particular test case *TC* when *M* takes the input from *TC* and defines the value of *F* during execution.

Note that non-final fields refer to the data members that are able to be re-defined or their values can be changed during execution. The opposite of non-final field are final fields whose values are only initialized once and are not allowed to be changed again. The combination of all non-final field values at a certain moment during execution determines the state of an object.

This definition requires keeping track of the runtime information (e.g., whether the field is actually defined in an 'if-else' block under a certain test input) and thus differs from the static-analysis approach [9] which considers a piece of modification code as an update once it detects such a modification in the code. Using runtime information is more precise for this purpose since object-oriented mechanisms such as dynamic binding and polymorphism can only be analyzed precisely at runtime.

Definition 2: An *observer* method *M* on field *F* is a method whose return value is affected by the usage of *F* in the method body.

Note that this definition emphasizes that the usage of *F* has an impact on the *observer* method return value. In other words, the state of field *F* can be observed from the *observer* return value. The simplest form of *observer* methods is getter methods which simply return the value of a field.

Definition 3: An *observer* method *M1* is identified as *associated* with a *state-modifying* method M2 on the field *F* if *F* is *defined* in *M2*. In data-flow analysis, *state-modifying* method M2 *defines* the field *F* and *observer* method *M1* *uses F*.

The classification of these two categories of methods *state-modifying* and *observer* method is based on the observation that, for a typical object-oriented class, fields are only accessed using *observer* methods, e.g., `getX()`. In test assertions, we check the program state by invoking *observer* methods. Identifying the relationship between *observer* and *state-modifying* methods is necessary to decide what *observer* methods to invoke in the assertion.

Definition 4: An automated test case (method) *TC* that exercises a *state-modifying* method *M* is said to have the problem of the IA problem if not all field updates are verified by the assertion(s), i.e., for any defined field *F* in *M*, none of the assertions in *TC* invoke any *observer* methods that are *associated with M* on *F* to verify the program state.

According to Definition 4, if there is any field that is modified in the method under test during execution but no assertion checks such update(s) by invoking *observer* methods, then the test case is considered suffering from the IA problem. In other words, the assertions fail to cover or check all state updates.

This is the post-print of a paper that has been published with the following DOI: http://dx.doi.org/10.1109/ICSTW.2013.49

B. An Example

In this section, we illustrate the IA problem with a simple example. Figure 1 shows a simple `ClassUnderTest`. This class has two private fields: `data1` and `data2`. The two methods, `getData1()` and `getData2()` are *observer* methods for these fields, respectively. In the `method1()`, `data2` is defined only when `flag` is greater than zero while `data1` is defined when `method1()` is invoked regardless of the value of flag.

Figure 2 shows a JUnit test case that tests the `method1()`. Because the value of `testInput`=1 which is greater than zero, during the runtime exercise of `method1()`, the state of `data2` will be actually modified. According to definitions 1-3, `data2` is a *defined field* for `method1()` and `getData2()` is the only *observer* method *associated* with `method1()`on `data2`. However, in the test script, we can see that `getData2()` is not invoked to check the state of `data2`. According to Definition 4, such a test case has the IA problem.

```
2  public class ClassUnderTest {
3      private int data1;
4      private int data2;
5      public int getData1() { return data1; }
6      public int getData2() { return data2; }
7
8      public void method1(int flag){
9          if(flag>0){
10             this.data2 = 2;      //define data2
11         }
12         this.data1 = 4;          //define data1
13     }
14 }
```

Figure 1 - A simple Java class

```
4  public class ClassUnderTestTest extends TestCase {
5
6      public void testMethod1() {
7          ClassUnderTest cut = new ClassUnderTest();
8          int testInput = 1;
9          cut.method1(testInput);
10
11         assertTrue(cut.getData1()==4);
12                 //Is this assertion adequate enough?
13     }
14 }
```

Figure 2 - JUnit test code for the class `ClassUnderTest`

IV. Case-Study Design and Setup

In this case study, we followed the guidelines proposed by Runeson and Höst[14]. Goal and research questions are presented in Section IV.A. Object of study is qualitatively and quantitatively described in Section IV.B Study setup details are given in IV.C.

A. Goal and Research Questions

Based on the GQM approach [15], the goal of this case study is to *characterize* the scale *of the IA problem* and the factors *contributing to such problems*, from the point of view of *software developers* in the context of *automated test suites*.

Our research goal is driven by the following research questions (RQs):

- **RQ1:** How common is the occurrence of the IA problem in test suites of open-source projects?
- **RQ2:** Which factors would (could) usually lead to inadequate assertion problem? Does the IA problem occurrence correlate to any internal characteristics of the SUT or its test suites, for instance, the complexity of Class Under Test (CUT) or its test code?

Metrics used to answer the RQs are listed in Table 1. We use Weighted Methods per Class (WMPC) as a metric to represent the complexity of a class a whole. WMPC is based on the McCabe's Cyclomatic Complexity [16] and is defined as the sum of complexities of methods defined in a class [17].

Table 1 - Metrics used in the study

	Metrics
Object of Study	Line of Code (LOC), statement coverage, Number of packages, Number of classes, Number of methods, Number of Non-final fields
RQ 1	Occurrence rate of the IA problem (test suite TS) = Ratio of the number of test cases with the IA problem in TS/ total number of test cases in TS
RQ 2	Complexity of a given class using the Weighted Methods Per Class (WMPC) metric [17]

B. Objects of Study (Systems under Test)

To measure the scale of the IA problem in real projects, we selected three open-source projects: lurgee (version 2.1)[1], Log4j (version 1.2.16)[2] and FindBugs (version 2.0.0)[3]. Our criteria for choosing these projects were as follows: (1) they are popular open-source projects with high downloads and attracting large number of users and active developers, and (2) along with their downloadable source code, there come with existing reasonably-large JUnit test suites so that there was no need for the authors to develop new test suites for those systems. We downloaded the source codes and the existing test suites of the above three projects from their websites, and analyzed the test suites for the purpose of our study. A brief explanation of each project is provided below.

Lurgee is a strategy game framework written in Java. Log4j is a utility application used for logging . FindBugs is a tool that uses static code analysis to detect potential bugs in Java programs. Table 2 shows several size metrics and test coverage of the three chosen SUTs. From the table, we can see that FindBugs is the largest project among the three in terms of lines of code (LOC), followed by Log4j. Lurgee is the smallest among the three in terms of LOC and number of packages. Note that not all of the packages of the three

[1] www.lurgee.net
[2] logging.apache.org/log4j
[3] code.google.com/p/findbugs

This is the post-print of a paper that has been published with the following DOI: http://dx.doi.org/10.1109/ICSTW.2013.49

Table 2 - Statistics for the objects of this study

SUT	LOC (including test code)	# of packages	# of packages with test code	LOC of JUnit test code	Test code / total code	Number of test methods	Statement coverage
Log4j	29,745	20	12	9,115	30.64%	958	39.4%
Lurgee	13,374	13	7	4,191	31.34%	373	54.9%
FindBugs	100,606	48	18	2,513	2.50%	284	1.3%

Table 3 - Sampled package statistics

SUT	Sampling ratio (packages)	Names of Sampled Packages	# of SUT classes	# of JUnit classes	# of JUnit test methods	Statement coverage	Branch coverage
Log4j	17% (2 of 12 packages)	`org.apache.log4j.helpers`	24	6	48	57%	49.2%
		`org.apache.log4j.pattern`	29	4	44	64.8%	61.9%
Lurgee	29% (2 of 7 packages)	`net.lurgee.connect4`	9	6	35	75.6%	71%
		`net.lurgee.ninemensmorris`	9	7	83	86.3%	84.1%
Find Bugs	11% (2 of 18 packages)	`edu.umd.cs.findbugs.jaif`	7	1	11	15.9%	15.1%
		`edu.umd.cs.findbugs.util`	32	1	6	2.3%	2.9%

SUTS have JUnit test suites as part of the downloadable package online. For instance, only 12 out of 20 packages (60%) in Log4j have JUnit test suites. In Lurgee and FindBugs, the ratios are 7/13 (54%) and 18/48 (38%), respectively.

C. Setup

We discuss next the following aspects of our case-study design: test-case selection (sampling), methodology and procedure.

1) Test-Case Selection (Sampling)

As Table 2 shows, the three objects (SUTs) of our study had, in total, about 16 KLOC of JUnit test code and 1,615 JUnit test methods. We had to conduct manual analysis (see the empirical procedure in Section IV.C.3) since there was no tool at hand that can automatically spot IA problems. In order to meet the time constraints of our study, we could not apply our study on all the test suites. We were unable to conduct pure random sampling of classes under test (CUT) from the three SUTs, due to the following two reasons. Firstly, not all of the classes in a given package have corresponding JUnit test classes. For example, in Log4j, 8 out of 20 packages do not have corresponding JUnit test suites. Obviously, we had to exclude the packages with no test cases.

Secondly, most existing JUnit test suites accompanied with the SUT source code achieved rather low statement coverage, for example, the total statement coverage of the test suite for FindBugs is only 1.3%. In other words, the main problem with these test suites is not the IA problem, but insufficient test cases.

To get the samples, we ranked the test packages and classes by their coverage values, and selected the top ones. We sampled two Java packages from each of the three projects. This gave us sampling ratios of 11%, 17%, and 29% for the three projects in terms of number of packages selected. Table 3 shows the details. These were the data sets that we found suitable for our analysis of the IA problem, based on the following justifications: (1) They all have a certain number of existing JUnit cases; and (2) The test suites in the sampled packages achieved moderately high statement coverage to ensure that most of the source code in CUTs is exercised, except for those packages in FindBugs which had very limited test suites.

2) Methodology

To discover the IA problem, we utilized the mutation testing approach [18]. Mutation testing is a technique used to assess the fault-detection effectiveness of a test suite. A live mutant indicates a deficiency in the test suite and that in most cases that either a new test case should be added, or that an existing test case needs a stronger assertion set. Let us recall the three conditions to kill a mutant [18]: (1) reachability (coverage) of the mutated part (line) of the code, (2) infecting the program state, and (3) propagation of the infected program state to cause incorrect program behavior /output and having proper assertions to find out about incorrect program behavior /output. Let us consider a non-equivalent mutant [18] and a given test suite for assessing the quality of its assertions w.r.t. the IA problem. If the execution of the test suite on the mutant satisfies the first two conditions above (#1 and #2), but not the third condition (not finding out about incorrect program behavior), the mutant will stay live, since the test suite lacks enough test oracles (i.e., assertions). In other words, when a mutated piece of code is covered by a test suite and infects the variable values in the program state, but this infection is not verified properly, it indicates that the testers did not write adequate assertions to verify those incorrectly-changed variable values.

The development/testing environment we used is the Eclipse Java IDE version 3.4. We used two Eclipse plug-ins: Jumble[4] and Clover[3], to assist our analysis. Jumble is used for mutation testing and the latter plug-in provide code coverage measurements in the Eclipse IDE.

[4] jumble.sourceforge.net

This is the post-print of a paper that has been published with the following DOI: http://dx.doi.org/10.1109/ICSTW.2013.49

Researchers have identified a set of "sufficient" mutation operators [19]: ABS, AOR, LCR, ROR and UOI. Previous work [20] classifies these operators into three categories: (1) constant replacement, (2) operator replacement, and (3) condition negation. We compared Jumble mutation operators[4] with the above-mentioned operator set and the result is shown in Table 4. We found that Jumble operators cover all categories of 'sufficient' operators and are thus enough to indicate the quality of test cases. For quantitative analysis, we used the mutation score (reported by Jumble) as our criterion to assess the quality of test cases. We were aware of few other more powerful mutation tools such as MuClipse and Javalanche. However, we had technical (installation) issues with getting them to work in our context. In summary, Jumble worked well for our purpose.

Table 4 - Mapping of Jumble mutation operator to "sufficient" set of mutation operators (proposed by [19]**)**

Jumble mutation operators	Corresponding mutation category
Conditional	Condition negation
Binary Arithmetic	Operator replacement
Increments	Operator replacement
Inline Constants	Constant replacement
Class Pool Constants	-
Return Values	-
Switch Statements	-

3) Procedure

Our procedure on each class under test was performed in two steps:

Step 1: We performed mutation analysis on the sampled test suites. We then collected the information about live mutants, including (1) the mutation operators that produce those mutants, and (2) source file line number where the mutation operator is performed. These two types of information helped us to analyze the reason why the test cases failed to kill each mutant.

Step 2: After mutating one class under test, we used the Clover to analyze the coverage of the class or method under test. From Step 1, we would know which mutant was not killed and where in the code-base it was located. We examined that line in the class under test. If that mutated line was covered by the test suite, it meant that the test case had covered that line but failed to kill that mutant. We determined whether this failure is due to that the test case is missing enough assertions or because the mutant was equivalent to the correct program. Afterwards, we attempted to add the extra assertions to the test method to kill (distinguish) the live mutant.

We recorded several important attributes for each CUT in each of sampled packages. These attributes include the number of non-final fields (NFF), total number of test methods of the corresponding JUnit test case class, the number of test methods we diagnosed with IA problem and the percentage of such problematic test methods against the total number of test methods.

V. RESULTS

Results for RQ 1 and RQ 2 are discussed in Sections V.A and V.B, respectively. Section V.C discusses the consequences, importance and implications of our findings. Section V.D discusses the potential threats to the validity of our study and steps we have taken to minimize or mitigate them.

A. RQ 1: Ratio (Scale) of the IA Problem

We measured the scale of the IA problem in two levels: package-level results, and class-level results, which are discussed next.

1) Package-level Results

Figure 3 shows the breakdown of number of test methods with the IA problem versus total number of test methods for each package. We can see that, while the IA problem is common among these open-source project test suites, it varies from project to project, or even from package to package, in terms of its percentage. For example, in package `org.apache. log4j.helpers`, 36 out of 48 test methods suffer from the IA problem, while in `edu.umd.cs.findbugs.util`, none of the test methods has the IA problem. Recall from Section IV.A that discussion of potential factors leading to the IA problem will be addressed by RQ 2 (Section V.B).

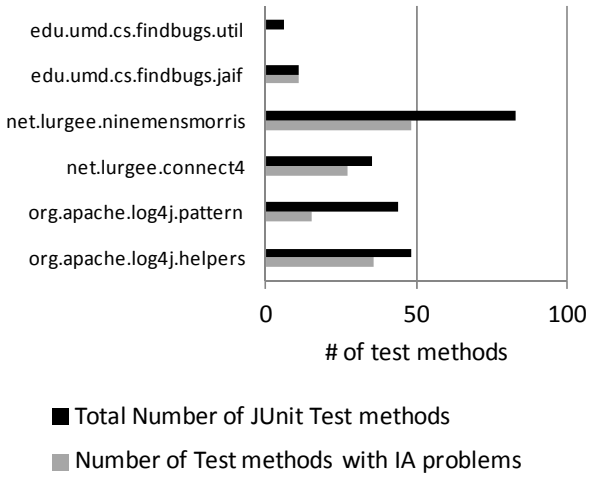


Figure 3 - Ratio of test methods with the IA problem versus total number of test methods for each package

2) Class-level Results

To investigate the rate of IA problem occurrence among each class under test, we present the class-level IA rates in Figure 5. **T**he labels on the X-axis are the names of each CUT and each name corresponds to two columns with different colors which represent the total number of test methods for that class and the number of test methods with IA problems, respectively. Figure **4** shows the percentages of test methods with IA problem for each class of each SUT

as an "individual-value plot", which have been calculated using the data in Figure 5.

As Figure **4** and Figure 5 show, certain classes have higher IA problem occurrence rate than the others, for example, all of test methods in classes `(lurgee)DateLayout`, `(lurgee) NineMensMorrisBoardLinks` and `(Findbugs) JAIFScanner` suffered from IA problem. However, most classes (64%) have no occurrence of IA. In other words, IA problem occurrence occurs on a small subset of classes only. More precisely, among the sampled 25 CUTs from the three projects, 16 classes (64%) had no occurrence of the IA problem, 5 classes (20%) had partial occurrence of the IA problem and 4 classes (16%) had full occurrence of the IA problem (all of their test methods had the problem).

One possible interpretation of the problem occurrence variance in different packages and classes is quite trivial: test suite for each package was developed by different programmers who have different expertise and skill levels and thus developed test suites with different quality. Some testers are careful to write complete assertions when checking the test output, while others paid less attention to the sufficiency of test assertions. In our RQ 2, we explore the correlation of the IA occurrence with other factors.

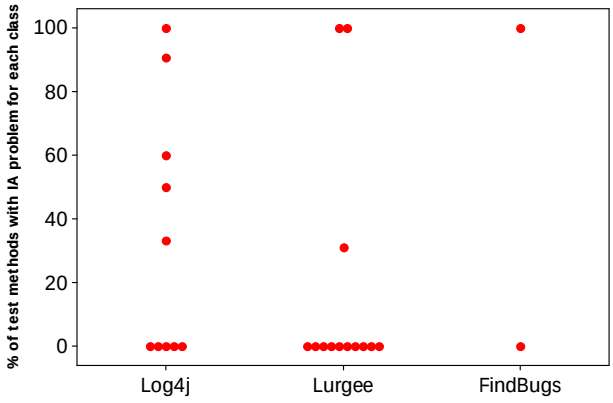


Figure 4 - **Percentage of test methods with IA problem for each class for each SUT**

B. RQ 2: Potential Factors Leading to the IA Problem

We list below three potential factors that we think could impact IA occurrence for a particular CUT:

- Number of non-final fields (NFF) in the CUT
- Code complexity of the CUT
- Code complexity of JUnit test suites of the CUT

Note that developer's or tester's expertise is also a candidate factor that might also lead to the rate of IA occurrence, but in this paper, we mainly explore the three factors discussed above and leave the expertise factor to

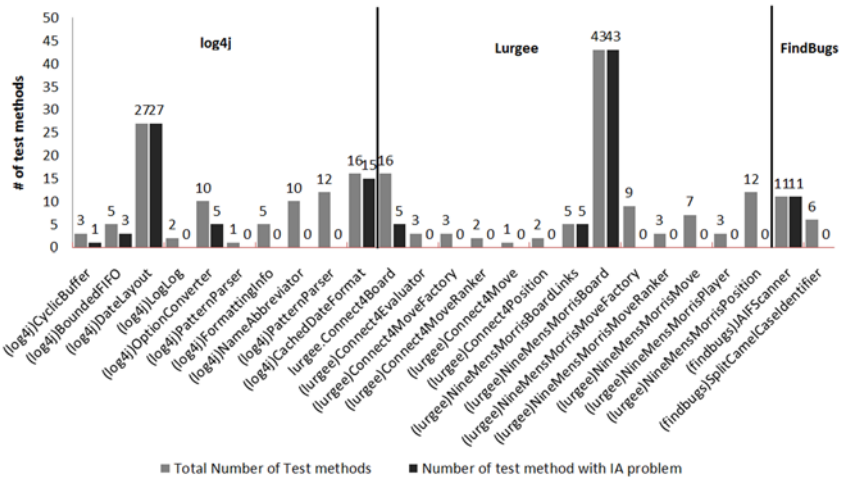


Figure 5 - Ratio of test methods with the IA problem for each class under test

future work. For each of the above three factors, we propose a hypothesis next and verify it with quantitative correlation analysis.

Hypothesis 1: IA occurrence rate is positively correlated to the number of NFFs in the CUT.

This assumption is based on the assumption that the more non-final data members a class has, the more likely testers forget to check all the state updates of these fields in the assertions.

Figure 6(a) shows the scatter plot of the IA occurrence percentage versus the number of NFF, grouped by each SUT. Each dot in the chart represents one class. From the scatter plot, we can see that IA problem occurs more frequently when the number of NFF increases to four or more (the case for log4j classes). However, there is an exception for this, as it can be seen in Figure 6(a): `org.apache.log4j.helpers.PatternParser` with eight NFFs. If there are less than three NFFs in the class, the IA issue is less probable to occur (but there is an exception class for this: the class `NineMensMorrisBoard` in project Lurgee). In Table 5, we can see that the correlation value between IA occurrence rate and the number of NFFs is 0.51 which reflects moderate correlation relationship.

This observation has practical sense because when there are more than a few NFFs in the class under test, without getting help from tools such as UnitPlus [9], a typical tester is quite likely to miss writing assertions to check the correctness of all NFFs, some of which may change during the exercise phase of the test method.

Hypothesis 2: IA occurrence rate is positively correlated to the complexity of the CUT.

Some testers might examine the inner structure of the class source code to derive test cases for it (e.g., white-box testing). This process is often based on static code inspection. However, a class with complex structure (e.g., with high number of nested "if-else" branches or "for" loops, etc.) is difficult to 'simulate' the control flow in

testers' minds without the help of automated tools. In such cases, testers are likely to miss trace all changes in the fields. This could be a factor leading to the IA problem. Therefore, hypothesis #2 is based on the assumption that the more complex the CUT is, the more likely it is for testers to write inadequate test assertions.

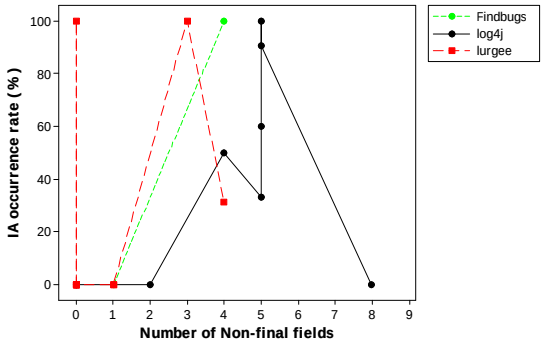
Figure 6(b) shows the correlation scatter plot between IA occurrence rate and WMPC of the class under test. Similar to the situation of the number of NFFs, we can observe that only when the complexity of CUT reaches a certain level, we start to see higher chances of the IA problem. The positive relationship between these two variables is apparent in Lurgee. We can easily spot one outstanding point where the complexity is close to 200 and IA occurrence rate close to 65%, which is an evidence of Hypothesis 2. However, in log4j we do not spot the similar trend. As the complexity of CUT increases, we do not witness the proportional increase in IA occurrence rate. In terms of FindBugs, it is difficult to predict its trend due to limited data.

Hypothesis 3: IA occurrence rate is positively correlated to the complexity of JUnit test class.

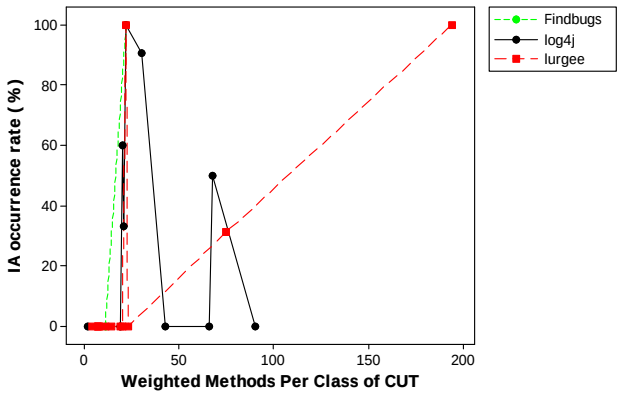
This is based on the assumption that if testers write complex JUnit test class, it is possible that they will make mistakes in the assertions. Figure 6(c) shows the correlation scatter plot between IA occurrence rate and WMPC of JUnit test class. Similar to previous case, we observed that only when the complexity of JUnit test class reached a certain level, we started to see higher chances of the IA problem.

3) Correlation Analysis

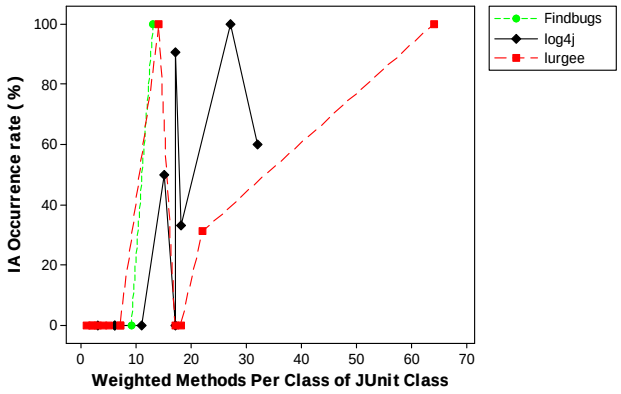
To quantitatively analyze the correlation among the three relevant factors (IA occurrence rate, WMPC of CUT, and WMPC of JUnit test class), we calculated the Pearson Correlation Coefficients (PCC) and p-values between each pair of variables. The results are shown in Table 5. From the table we can see that there is a strong correlation (PCC=0.75, P-value=0.000) between WMPC of CUT and WMPC of JUnit test class. Surprisingly, IA problem occurrence rate is more closely related to WMPC of JUnit test class (PCC= 0.64, P-value=0.001) rather than WMPC of CUT (PCC= 0.36, P-value=0.074). This result indicates that Hypothesis #3 is a better supported by the quantitative results than Hypothesis #2.



(a)



(b)



(c)

Figure 6 - IA occurrence rate vs. number of NFFs, WMPC of CUT, and WMPC of JUnit test class

Table 5-Results of Correlation Analysis

Metrics	# of NFF (Hypothesis 1)	WMPC of CUT (Hypothesis 2)	WMPC of JUnit class (Hypothesis 3)
IA occurrence rate	PCC=0.51 p-value=0.009	PCC=0.36 p-value=0.074	PCC=0.64 p-value=0.001
# of NFF	-	PCC = 0.39 p-value = 0.055	PCC=0.39 p-value =0.054
WMPC of CUT	-	-	PCC = 0.75 p-value=0.000

C. Consequences, Importance and Implications of Findings

The consequences of having the IA problem in test code include having less powerful test suites (i.e., low fault detection effectiveness). In other words, the IA problem directly impacts the quality of test suites. To ensure a high-quality production code, we clearly need a high-quality test suite, and a test suite with high ratio of IA problem would not be effective to guard programs against faults.

Our results imply that testers should pay attention to adequacy of assertions in test suites. While having enough test cases to achieve high coverage values and mutation scores is important, it is notable to have sufficient assertions to ensure sufficient verification of the CUT state. We need to mention at this point that considering and including all

This is the post-print of a paper that has been published with the following DOI: http://dx.doi.org/10.1109/ICSTW.2013.49

necessary assertions in test code is usually costly and effort intensive. Thus, a form of tradeoff exists in this context between the completeness of assertions and the time/effort spent by testers.

Under some circumstances, testers may need to prioritize addressing IA problems over adding new test cases (with the hope of increasing white-box or black-box coverage). As Figure 3 and Figure 4 show, IA occurrence varies in different systems or packages. Let us consider package `net.lurgee.ninemensmorris` as an example. This package had a large number of test methods (Figure 3) and its test suite achieves relatively high statement and branch coverage (Table 3). Yet it suffered from high IA occurrence rate, i.e., more than 50% of its test methods (Figure 3). The insufficiency of assertions may harm the fault-detection ability of existing test cases. In cases similar to this package where high occurrence rate is observed (a threshold level may be set) IA, we believe that it is wise for testers to consider adding the missing assertions to existing test cases, instead of developing new test cases, which may take substantial efforts but not guarantee improved fault detection ability. Development of methods for systematic decision support to help testers decide whether to address IA problems for a given CUT, or to add new test cases is out of scope of our study. But the above example can be used as a starting point for such efforts.

Fortunately, we can get support from existing tools to mitigate IA problems. For developing test cases while preventing the IA problem, semi-automated tools such as UnitPlus [9] can assist testers in developing all necessary assertions in test code. For fixing the IA problem, we can use tools such as Orstra [13] to improve test assertions, solving IA problems.

Besides, our correlation analysis is a preliminary attempt to investigate the root cause of IA problems. It sheds lights into the relationship among the IA occurrence rate and complexity of the CUT or JUnit test class. Within our sampled data, we witnessed a positive correlation between WMPC of JUnit class and IA occurrence rate, between WMPC of JUnit class and WMPC of CUT. This result initiates an interesting perspective on the cause of IA problem. Further investigation on whether human aspects (e.g., expertise level) lead to the IA problem should be put in our future research agenda.

D. Threats to Validity

Potential threats to the validity of our study and steps we have taken to minimize or mitigate them are discussed next.

External validity: The test suites collected in the study are sampled from three open-source java projects. Data is limited so that it is difficult to generalize our findings on IA problem. Our mitigation is that among the three sample SUTs, we attempted to select projects that attracted active developers or testers. In this way, the test suites developed such practitioners might represent the real testing practice at a certain level.

Construct validity: We used mutation testing techniques to assess the quality of automated test code in this study. The validity issues lie in whether using mutation testing can reveal the adequacy of test assertions. We utilized the 'sufficient set' of mutation operators proposed by previous researchers [20] in order to create real-world defects. The other issue is, without using dynamic analysis that records run-time field updates and checks, it is likely that some IA problems are missed. For this issue we argue that, by using the automatic mutation tools (i.e., Jumble), nearly all lines of codes that read or update fields will be mutated. Thus, a large number of mutants that cover a spectrum of faulty field updates will be generated. If a test case has IA problems, i.e. not enough assertions, it is highly possible that such a test case cannot detect the whole set of various mutants. In other words, the large set of various mutants can ensure that nearly all IA problems will be detected.

Conclusion validity: The assumption behind out analysis is that testers, due to negligence, forgot to write enough assertions to ensure the correctness of the program state. However, another possible explanation might be that testers are actually aware of the assertions but ignore it because those unchecked data is not worthy of checking in terms of their effect on the program state. Our mitigation is to sample our data from several projects which were obviously developed by different projects. The common occurrence of IA problem might indicate common unawareness of testers about IA issues.

VI. CONCLUSION AND FUTURE WORK

Test assertions, as an integral part of test case to verify the program state, are usually derided manually by testers. The sufficiency of assertions is a potential indicator of test code quality. By sufficiency, it refers to the extent of whether changes in data members are checked by assertions. In this paper, we report an empirical study on such Inadequate Assertion (IA) problem. We collected test case data from three active open-source projects. In the case study we performed mutation analysis and coverage analysis to assess the quality of the sampled test cases. Through the statistical analysis, we found that:

- In the package level, IA problem is common among the sampled open-source project test codes, but it varies from project to project, from package to package;
- In the class level, IA program occurrence is not regular. In other words, certain classes have higher IA problem occurrence rate than the others;
- In terms of the correlation between IA program occurrence and other variables, our limited data shows that (1) IA problem occurs more frequently when the number of NFF increases to 4 or more; (2) IA problem occurrence rate is more closely related to WMPC of JUnit test class rather than WMPC of CUT, which indicates that the more complex a test case is, the more easily testers forget to write enough assertions to check the program states;

For future work, much can still be done to improve the applicability of the study. This work includes:

- Conducting further in-depth empirical studies on the IA problem. We need to have further evidence from test suites of commercial software or other open-source projects;
- Conducting studies on how human factors (e.g., level of experience) might lead to the IA problem. As discussed in Section IV, the level of software developer/tester's experience and its their relationship with the IA problem occurrence rate should be studied;
- Solving the IA problems in existing test suites of the three SUT by adding needed assertions and investigate whether we can detect real faults in the current or future versions of the SUTs;
- Analyzing the IA problems in partnership with industrial software testers and to assess the extent and (negative) consequences of these problems in real-world projects.

ACKNOWLEDGMENT

The authors would like to thank Tao Xie and Dongcheng Deng for their insightful comments. The authors were supported by the Discovery Grant no. 341511-07 provided by the NSERC. Vahid Garousi was also additionally supported by the Visiting Scientist Fellowship Program (#2221) of the Scientific and Technological Research Council of Turkey (TÜBİTAK).

REFERENCES

[1] G. Venolia, R. DeLine, and T. LaToza, "Software Development at Microsoft Observed: It ' s about people … working together," *Microsoft Research*, 2005.

[2] X. Xiao, S. Thummalapenta, and T. Xie, "Advances on Improving Automation in Developer Testing," in *Advances in Computers*, Burlington: Academic Press, 2012, pp. 165-212.

[3] S. Wang and J. Offutt, "Comparison of Unit-Level Automated Test Generation Tools." Fifth Workshop on Mutation Analysis (Mutation 2009), 2009.

[4] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[5] D. Vanoverberghe, J. D. Halleux, N. Tillmann, and F. Piessens, "State Coverage: Software Validation Metrics beyond code coverage - extended version," 2011.

[6] D. Vanoverberghe, J. D. Halleux, N. Tillmann, and F. Piessens, "State Coverage: Software Validation Metrics beyond code coverage," in *SOFSEM 2012: Theory and Practice of Computer Science*, 2012.

[7] T. Knauth, C. Fetzer, and P. Felber, "Assertion-Driven Development: Assessing the Quality of Contracts Using Meta-Mutations," *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pp. 182-191, 2009.

[8] K. Koster and D. Kao, "State Coverage: A Structural Test Adequacy Criterion for Behavior Checking Matching Checks to Definitions," in *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2007, pp. 541-544.

[9] Y. Song, S. Thummalapenta, and T. Xie, "UnitPlus: Assisting Developer Testing in Eclipse," in *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, 2007, pp. 26-30.

[10] M. Staats, M. W. Whalen, and M. P. E. Heimdahl, "Programs , Tests , and Oracles : The Foundations of Testing Revisited," in *33rd International Conferences on Software Engineering (ICSE'11)*, 2011, pp. 391-400.

[11] Q. Xie and A. M. Memon, "Designing and comparing automated test oracles for GUI-based software applications," *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 1, p. 4-es, Feb. 2007.

[12] D. Schuler and A. Zeller, "Assessing Oracle Quality with Checked Coverage," in *4th IEEE International Conference on Software Testing, Verification and Validation*, 2011, pp. 90-99.

[13] T. Xie, "Augmenting Automatically Generated Unit-Test Suites," in *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP 2006)*, 2006, pp. 380-403.

[14] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131-164, Dec. 2008.

[15] V. Caldiera and H. Rombach, "The Goal Question Metric Approach," *Encyclopedia of Software Engineering*, vol. 2, pp. 528–532, 1994.

[16] T. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, no. 4, pp. 308-320, 1976.

[17] "Weighted Methods Per Class." [Online]. Available: http://eclipse-metrics.sourceforge.net/descriptions/pages/WeightedMethodsPerClass.html(Last accessed: August 2012).

[18] A. P. Mathur, *Foundations of Software Testing: Fundamental Algorithms and Techniques*. Pearson Education, 2008.

[19] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An Experimental Determination of Sufficient Mutation Operators," *ACM Transactions on Software Engineering and Methodology*, 1996.

[20] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria," vol. 32, no. 8, pp. 608-624, 2006.