# Test Redundancy Measurement Based on Coverage Information: Evaluations and Lessons Learned

Negar Koochakzadeh, Vahid Garousi
*Software Quality Engineering Group*
*Department of Electrical and Computer Engineering, University of Calgary*
{nkoochak, vgarousi}@ucalgary.ca

Frank Maurer
*Agile Methods & Web Engineering Group*
*Department of Computer Science, University of Calgary*
frank.maurer@ucalgary.ca

## Abstract

*Measurement and detection of redundancy in test suites attempt to achieve test minimization which in turn can help reduce test maintenance costs, and to also ensure the integrity of test cases. Test suite reduction based on coverage information has been discussed in many previous works. However, the applications of such techniques on real test suites and realistic measurements of redundancy have not yet been experimented thoroughly. To address such a need, we formulate in this paper two experimental metrics for coverage-based measurement of test redundancy in the context of JUnit test suites. We then evaluate the approach by measuring the redundancy of four real Java projects. The automated measures are compared with manual redundancy decisions (performed through an inspection by a tester). The results and lessons learned are interesting and somewhat surprising in that besides they show usefulness of coverage information, they present a set of shortcomings (in terms of precision) for the simplistic coverage-based redundancy measurement approach as discussed in the literature. The root-cause analysis of our observations identify several key lessons learned which should help the testing researchers and practitioners in devising better techniques for more precise measurement of test redundancy.*

## 1. Introduction

Software testing is the most visible activity in the software quality assurance. As a software system evolves, the body of its test suites grow together with the source code [1]. However, by refactoring and changing requirements, test code might start to erode [2, 3]: it becomes complex and unmanageable [4]. Although such tests might still have the ability to check the correctness of the system at present time, they can easily break when further adaptations to the application code are required [1].

The decayed parts of the application code are often referred to as *code smells* [5]. Similarly, a *test smell* is symptom of a problem in test code [6]. A smell doesn't necessarily tell us what is wrong because there may be several possible causes for a particular smell [6].

Redundancy (among test cases) is a discussed (e.g., [1]) but a seldom-studied test smell. A redundant test is one, which if removed, will not affect the overall requirement of the test process. For instance, by eliminating a redundant test case from a test suite, the fault detection effectiveness of the test suites will not change[7]. Redundant test cases can have serious consequences, e.g., if two test cases test the same feature of a unit, if one of them is updated correctly with the unit and not the other one, the integrity of the test suite will be under question, i.e., one test case will fail while the other will pass, leaving the QA team in an ambiguous situation.

The motivation for test redundancy detection is straightforward: by detecting and dealing with redundant test case (e.g., carefully removing them), we reduce the costs of validating and managing those test suites over future releases of the software [8].

Another type of test redundancy is introduced in [9] and [8], which is test code duplication. This kind of redundancy is similar to normal code duplication and is of syntactic nature.

To clearly highlight the notion of redundancy in the test smell name and also to differentiate the nature of syntactic and semantic redundancies, we refer to the above two types of test redundancy as: syntactic test redundancy and semantic test redundancy smells.

Therefore, a test case is semantically redundant when it does not improve the fault detection capability of the suite. In this work, we focus on the semantic redundancy smell which is known to be more challenging to detect in general than syntactic test redundancy [9].

There are a handful number of works in the testing literature which address the issue of semantic test redundancy. In most of those works, different types of code coverage criteria have been used as the tool for redundancy detection [7, 8, 10-13]. The rationale followed has been that if several test cases in a test suite execute the same program components, the test suite can be reduced to a smaller suite that guarantees equivalent coverage [8].

According to the definition of semantic redundancy, in addition to coverage criteria, fault detection effectiveness should also be considered. Therefore, to eliminate redundant test cases (according to coverage information) from a test suite, testers need to find how this reduction would affect the fault detection capability of that test suite. To evaluate this, some of the mentioned works on test redundancy detection [8, 12] have tried to study the effects of test set minimization on fault detection effectiveness.

Although all the existing methodologies contribute differently in targeting the issue of test redundancy, they all do it on a rather abstract level, with small examples. None of those techniques have been applied to real-world projects with real test suites.

In this work, we evaluate the redundancy detection and measurement based on coverage information, on four real Systems Under Test (SUT) written in java, with existing test suites written in JUnit. For this purpose, we formulate two metrics based on the measurement approaches reported in the previous works. We then compare the measured redundancy values with manual inspection results. In inspecting a test case to identify it as redundant or not, in addition to coverage information, we consider the fault detection effectiveness of the test case. The result of this comparison is used to evaluate the preciseness of redundancy detection based on coverage information.

The remainder of this article is structured as follows. We review the related works in Section 2. The formulation of redundancy metrics based on coverage information is discussed in Section 3. Redundancy measurement evaluation and lessons learned are discussed in Section 4. Finally, we conclude the article in Section 5 and discuss the future works.

## 2. Related Works

There exist works that address test suite minimization by considering different types of coverage criteria. Table 1 summarizes the coverage criteria, SUTs and the ratio of detected redundancy in previous works.

In all of the papers in Table 1, to achieve the maximum possible test reduction, the smallest test set (with least test cardinality) was created. The problem of finding the smallest test set has been shown to be NP-complete [14]. Therefore, in order to find an approximation to the minimum cardinality test set, different heuristics were used in those works.

SUTs used in these papers seem to be rather very small (except [10]). Test suites in some of those works were not originally available, and were thus produced according to black-box coverage [8, 10, 12] or mutation-based coverage [11]. As shown in Table 1, each paper considers different types of coverage criteria, while the basic idea of all of them is the same: a test case $t_i$ is considered redundant when the coverage of the test suite with and without $t_i$ stays the same.

**Table 1 –Summary of the related works**

| Ref. | Coverage Criterion | SUTs | | | Avg. % of Redundancy |
|---|---|---|---|---|---|
| | | Number | LOC Avg. | LOC STD | |
| [7] | All-def-use | One simple program | * | * | 40 |
| [12] | Statement Mutation | 10 Programs | 231 | 224.6 | 34 |
| [10] | MC/DC | Space Program | 9564 | - | 10 |
| [8] | Predicate-use Computation-use Definition-use All-use | 10 Unix Programs | 354.4 | 127.9 | 91 |
| [11] | Branch | 7 C Program | 29 | 19 | 50 |

*: Data are not available.

In [7], in addition to the experiment which was performed for all-def-use coverage criterion, it is mentioned that all the possible coverage criteria should be considered in order to detect more precise redundancy.

The ratio of reduction in test suite of SUTs is reported in all of the papers, while only [8, 12] have evaluated the results. In these works, faults were injected into the SUTs to generate mutants. Fault detection effectiveness of initial test suite was compared with the reduced one by considering the mutation scores of these two test sets. [8] concludes that test minimization based on coverage information can reduce the ability of fault detection, while [12] showed opposite conclusions. The root causes of reduction in fault detection effectiveness were not discussed in [8].

In addition to the above works, we have found a new idea of test redundancy in CodeCover tool between a pair of JUnit tests [6]. In this tool, there is a feature called *correlation* for each pair of tests which is an asymmetric function, and returns a real number in [0…1]. This metric is the ratio of covered items (e.g.,

statements) by both tests over the covered items of the first test. This is rather a novel concept in redundancy measurement in which partial values of redundancy are meaningful, e.g., a test can be 33% redundant with respect to another test.

To the best of our knowledge, no existing works other than the CodeCover tool has defined a redundancy metric. There has been no work that evaluated redundancy detection based on coverage information on large-scale systems and their test suites. Furthermore, the root causes analysis of redundancy and comparisons with manual inspections of test codes are not performed in large scales.

# 3. Formulation of Test Redundancy Metrics

In this section, we first explain the rationale on how to find redundant tests based on coverage information, extracted from test minimization algorithms presented in the literature. We then define two metrics to measure redundancy of JUnit tests.

## 3.1. Criteria

There are different test coverage criteria that specify test requirements, and are used to measure test adequacy. Each of these criteria has a coverable item set in SUT code, and each test case in the test suite may cover one or more items of these sets. To be able to identify a test case as a redundant test, based on coverage information, we should make sure that a test case does not improve a specific criterion.

To better explain the idea of finding redundancy based on coverage information, let us consider Figure 1 which illustrates the Venn diagram of a hypothetical SUT code with the coverage information of 5 test cases: $T_1,\ldots,T_5$, with respect to a typical coverage criterion (e.g., statement coverage).

According to this coverage criterion, $T_4$ and $T_5$ are fully redundant because $T_3$ is already covering the code statements covered by them.

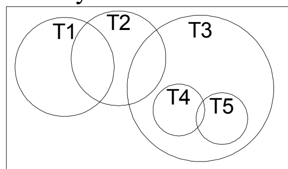


**Figure 1 - SUT coverage according to one typical criterion**

Figure 2 shows the coverage of the same SUT by considering another coverage criterion (e.g., branch coverage). According to this coverage criterion, $T_2$ and $T_5$ are redundant. Therefore, by considering both adequacy criteria, only $T_5$ will be considered redundant.

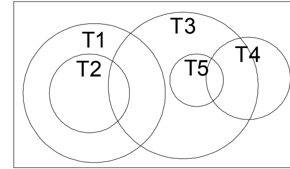


**Figure 2 - SUT coverage according to one typical coverage criterion**

According to various coverage criteria, coverable items generate separate sets and total coverable item set can be defined as the union of all those sets.

On the other hand, test redundancy can be defined in different granularities. Figure 3 shows different granularity levels of a test artifact, as supported in JUnit. Three levels of package, class and methods are grouping mechanism for test cases that are introduced in unit test frameworks like JUnit. The lowest level contains different phases of a test case [6]:

- **Setup**: The required state of the SUT for the purpose of a particular test case is set up, i.e., pre-condition of the test case.
- **Exercise**: SUT is exercised, i.e., a set of method calls are made to generate the outcome.
- **Verify**: This is where the test oracle is. The expected outcome is compared to the actual outcome.
- **Teardown**: The system (or unit) state is rolled back into the state before running the test case.

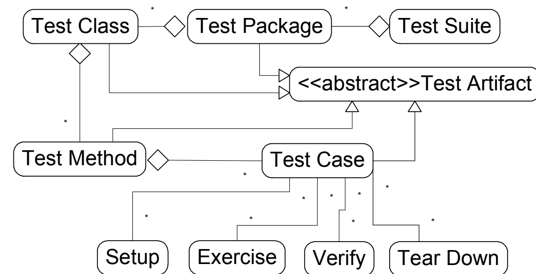


**Figure 3 – Test granularity in JUnit**

To quantitatively measure redundancy based on coverage information as discussed in the literature [7, 8, 10-13], two types of redundancy metrics are defined next: pair redundancy (to measure the redundancy of a test artifact with respect to another), and test suite Redundancy (to measure the redundancy of a test artifact with respect to all other artifacts).

These two metrics are extracted from test minimization algorithms presented in the literature. However, the bi-goal of those algorithms was to prioritize tests and their outcome was a reduced test set. There is no test prioritization goal in the proposed metrics in this work.

These metrics are defined in the next sections. Both of them can be applied on all the test artifact granularities (Figure 3).

### 3.2. Pair-wise Redundancy Metric

Figure 4 illustrates the idea behind the proposed redundancy metrics. Redundancy of each test artifact $t_j$ (i.e., a test case, a test method, or a test class in JUnit) with respect to $t_k$ can be measured by the ratio of the SUT parts that is covered by both test artifacts (area $A$) over the coverage achieved by $t_j$ (area $B \cup A$).

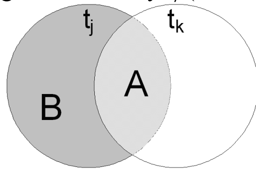


**Figure 4 –Illustration of the idea behind pair redundancy measurement**

This idea is formulated in Equation 1. Note that this type of redundancy ratio can be measured for each coverage criterion separately.

**Equation 1 – Redundancy of one test artifact with respect to another one**

$$PR(t_j, t_k) = \frac{\sum_{i \in CoverageCriteria} |Covered_i(t_j) \cap Covered_i(t_k)|}{\sum_{i \in CoverageCriteria} |Covered_i(t_j)|}$$

Where: $Covered_i(t_j)$ is item set covered by test artifact $t_j$, according to a given coverage criterion $i$.

### 3.3. Test Suite-level Redundancy Metric

We generalize the rationale behind the above pair redundancy metric to define redundancy of a test artifact with respect to all artifacts in a test suite (Equation 2), where $TS$ is the entire test set.

**Equation 2 - Redundancy of one test artifact with respect to all others**

$$SR(t_j) = \frac{\sum_{i \in CoverageCriteria} |Covered_i(t_j) \cap Covered_i(TS - t_j)|}{\sum_{i \in CoverageCriteria} |Covered_i(t_j)|}$$

Based on the design rationale of above metrics, their values are always a real number in the range of [0 …1]. Zero-value shows non-redundant tests while one-value is for full redundant tests. Other values enable us to measure redundancy in a partial quantitative domain (the concept from CodeCover). Such partial measures can help, e.g., in test maintenance by providing early warnings about the test artifacts which have the potential to be fully redundant in near future.

There might exist some rare cases in which a test artifact does not cover any type of items (according to the considered coverage criteria). In our SUTs, we have found that these cases may occur for various reasons, e.g., (1) a test case may only cover items outside the SUT (e.g., an external library), (2) a test case may verify (assert) a condition without exercising anything from the SUT, or (2) a test method may be completely empty (developed by mistake). In these cases, the nominator and the denominator of both above metrics (PR and SR) will be zero (thus causing problem). To make our redundancy criteria consistent, as these tests do not increase the coverage of SUT, we consider them as redundant tests.

As we discuss in Section 4.3, the more types of coverage criteria used to measure redundancy, the more accurate redundancy measures can be obtained. In functional testing, coverage criteria are either white-box or black-box. Since we were limited by the coverage tool we used, i.e., CodeCover [15], we employ four types of coverage criteria in this work: statement, branch, condition (MC/DC) and loop. The loop coverage criterion, as supported by CodeCover, requires that each loop is executed 0 times, once and more than once.

## 4. Evaluation of Redundancy Metrics

To assess the applicability of redundancy measurement based on coverage information and to evaluate the metrics formulated in Section 3, we applied our metrics in an experimental study to the test methods of four real-world test suites.

This section explains, in order, the experiment setup and the redundancy results of the four SUTs. Afterwards, to evaluate redundancy metrics and explain the results, we raise four research questions (RQs) and investigate the test suites manually. After each RQ, the most notable lessons learned and suggestions for the testing community in devising better (more precise) techniques for measurement of test redundancy are presented.

### 4.1. Experiment Setup

We used four open-source SUTs (all developed in Java) as objects of our case study: FitNesse [16], Lurgee [17], Allelogram [18] and JMeter [19].

FitNesse is a lightweight framework used to define and execute acceptance tests. This wiki tool allows testers to create and edit web pages as their acceptance tests [16]. Lurgee is a strategy game framework that provides the means for defining the entities that comprise the game (e.g., board, players, and moves) [17]. Allelogram is a program for processing genomes and is used by biological scientists [18]. JMeter is an Apache Jakarta project that can be used as a load testing tool for analyzing and measuring the performance of web services [19].

Table 2 shows the relevant size measures of the four SUTs. JMeter is the largest one in terms of LOC and Allelogram is the smallest. SUTs are selected from various scales of small and medium sizes.

**Table 2 – SUTs used in the experiment**

| SUT | LOC | Number of Packages | Number of Classes | Number of Methods |
|---|---|---|---|---|
| FitNesse | 22,673 | 33 | 479 | 2,219 |
| Lurgee | 7,050 | 10 | 106 | 684 |
| Allelogram | 3,296 | 7 | 57 | 323 |
| JMeter | 69,424 | 106 | 704 | 6353 |

The unit test suites of all above SUTs are available through the projects websites and are developed in JUnit. Table 3 and Table 4 respectively list the most relevant metrics about the test suites of our experiment objects, and their code coverage according to the following four criteria: statement, branch, condition, and loop (measured using the CodeCover tool [15]).

As shown in Table 4, the selected test suites have varying amount of code coverage. The FitNesse test suite has the highest coverage according to all four criteria, denoting that it is more comprehensive than the other four test suites.

**Table 3 – Test suite of the experimental SUTs**

| Test Suite | Test suite LOC | Number of test classes | Number of test methods |
|---|---|---|---|
| FitNesse | 16,777 | 197 | 1241 |
| Lurgee | 2,358 | 21 | 82 |
| Allelogram | 942 | 20 | 110 |
| JMeter | 11,993 | 80 | 422 |

**Table 4 – Coverage information (%)**

| SUT | Coverage (%) | | | |
|---|---|---|---|---|
| | Statement | Branch | Condition | Loop |
| FitNesse | 82.0 | 69.0 | 62.6 | 56.9 |
| Lurgee | 23.3 | 34.7 | 35.9 | 22.2 |
| Allelogram | 18.8 | 20.2 | 15.4 | 16.7 |
| JMeter | 16.3 | 14.4 | 12.6 | 10.9 |

The CodeCover tool [15] was used in our study which is an open-source coverage tool written in Java supporting the above four coverage criteria.

The lowest implemented test artifact level in JUnit is test method (which can contain several test cases).We thus measure redundancy of all the above test suites on test method level. To automate the measurement of redundancy of each test method using the two metrics defined in Section 3, we have slightly extended CodeCover to calculate the metrics and export them to a file, once it executes a test suite.

## 4.2. Experiment Results

The result of our experiment includes suite-level redundancy for each test method and pair-wise redundancy for each pair of test methods in the test suites of the four SUTs. Table 5 reports the percentage of fully redundant test methods according to each coverage criterion and the all of the criteria according to SR =1.

**Table 5 – The % of fully redundant test methods**

| Type of Redundancy | FitNesse | Lurgee | Allelogram | JMeter |
|---|---|---|---|---|
| Statement | 85 | 84 | 77 | 68 |
| Branch | 83 | 72 | 84 | 73 |
| Condition | 83 | 70 | 83 | 73 |
| Loop | 96 | 90 | 87 | 90 |
| Suite | 77 | 66 | 69 | 58 |

As we expected, according to Table 5, ratio of full redundancy detected by all coverage criteria separately is higher than the total one (Suite Redundancy). The reason is that in suite redundancy a test method which improves the coverage of each criterion is not considered as redundant test. Therefore, in the rest of our experiment we study on the suite redundancy.

To evaluate the precision of test redundancy measures, the first author (a graduate student of software testing) manually inspected about 37% (31 out of 82) of the unit test methods in Allelogram and about 2% (10 out of 422) of test methods in JMeter. We then identified false-positive errors (non-redundant test methods identified as redundant by the measures) and true-negative errors (redundant test methods identified as non-redundant by the measures).

Manual inspection of redundancy was performed by considering one of the original definitions of test redundancy: a test method is labeled redundant if it cannot increase the potential ability of fault detection. The results of this inspection are discussed next.

95% of the test methods which were manually recognized as redundant had been detected as redundant with the SR metric (SR=1). In other words, most of the real redundant tests are detected correctly, denoting a low ratio of true-negative errors (5%).

As to false-positive errors, 52% of those test methods manually recognized as non-redundant, had been detected as redundant tests by the SR metric, meaning that the metric was quite imprecise in this aspect. Figure 5 illustrate the values of above errors for Allelogram.

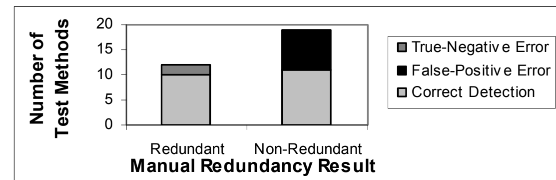


**Figure 5 – Errors in redundancy detection in Allelogram**

To investigate the root causes of the above impreciseness of redundancy detection based on coverage information, we raise and address four research questions discussed in next section.

High false-positive errors in Figure 5 denote the shortcoming of coverage-based redundancy detection. The best approach to investigate the false-positive errors is manual inspection. However, it is not practical to inspect all the tests in a large test suite. On the other side, the low ratio for true-negative error reveals that the search space of tests for manual inspection has already been reduced (with the ratios in Table 5).

### 4.3. Research Questions

When we started to manually inspect redundancy in test methods, it was hard to detect redundancy by considering potential fault detection effectiveness. To find the real redundancy of one test method, pair redundancy helped us to narrow down the rest of test suite to some test methods that have pair redundancy with the test method under investigation. The following research questions are the result of an iterative root-cause analysis we conducted to investigate the main causes of excessive impreciseness in test redundancy measurement based on coverage information:

RQ1: Should other coverage criteria (than the four we have) be also considered in redundancy measurement?

RQ2: Can separation of test cases in a JUnit test method make redundancy measurement more precise?

RQ3: Can separation of test phases improve the precision of redundancy measurement?

RQ4: Does the particular implementation of standard code coverage criteria affect accuracy of redundancy measurement?

In the following sections, we investigate each of the above research questions using the data we gathered in a set of controlled experiments, examples from our SUTs and lessons learned for each of the questions.

**RQ1 – Should other coverage criteria (than the four we have) be also considered in redundancy measurement?**

As discussed in Section 3.1, the set of covered items by one test method for different coverage criteria are different. Subsequently, the intersections between covered item sets of two test methods for different coverage criteria would be different as well. The redundancy detected by only considering a set of few criteria, may not guarantee measuring and detecting the real redundancy according to all of the possible coverage criteria (i.e., including the black-box criteria as well, e.g., equivalence classes).

As the coverage tool we chose (CodeCover [15]) provided four criteria only, we only considered those four: statement, branch, condition(MC/DC) and loop. The following examples show some cases in which not considering other coverage criteria (e.g., path coverage and equivalence class coverage) could negatively affect the accuracy of measures.

**Example 1 (Path Coverage):** In JMeter, test method *testCookieMatching* from class *TestCookieManager* in package *org.apache.jmeter.protocol.http.control* is detected as a redundant test (SR=1). The reason is that all of coverable items are covered by other test methods as well. For instance, all the statements covered by this test method is covered by the test method *testNewCookie* (from the same class) as well (e.g., PR(*testCookieMatching, testNewCookie) =1)*. Figure 6 shows the constructed control flow graph of method *getCookieHeaderForURL* that is covered by both of the above test methods. In this control flow graph, *path1* is covered by both tests, while *path2* is covered only by *testCookieMatching*. Therefore *testCookieMatching* is not really a redundant test as it covers a new path in addition to the path covered by another method.

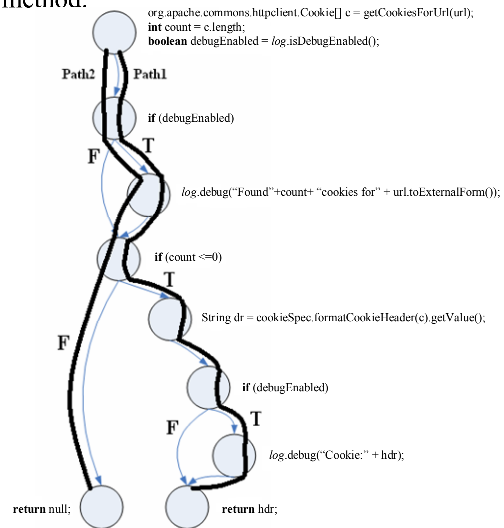


**Figure 6 - Control Flow Graph of method *getCookieHeaderForURL***

**Example 2 (Equivalence Class Coverage):** In Allelogram, the test method *testRequireAtLeastTwoAlleles* from class *GenotypeTest* in package *org.carlmanaster.allelogram.model.tests* is detected as a redundant test (SR=1). The reason is that all of the items in the SUT code covered by this test method are covered by the test method *testCreate* (from the same class) as well. However, the test method *testRequireAtLeastTwoAlleles* is testing whether there is an exception thrown by the SUT. This type of *expected* (intentional) exception is based on the specification of that function. In other words, the inputs of two above tests are from different equivalence classes (i.e., a black-box coverage criterion should be applied). Therefore *testRequireAtLeastTwoAlleles* should not be detected as a redundant test.

**Example 3 (Different execution Orders):** In Allelogram, test methods *testPredicate* and *testGenotypeConstructor* from class *GenotypeClassificationPredicateTest* in package *org.carlmanaster.allelogram.model.tests* cover the same statements of SUT but in different orders. Therefore, the goals of these two test methods are meant to be different and none of them should be considered as redundant because of the other one. According to the path coverage criterion, the two above test methods cover different items. Therefore, they would not be detected as redundant by considering this coverage criterion.

**LESSONS LEARNED:** To achieve a more reliable (precise) form of redundancy measurement, one needs to consider all types of coverage criteria. The most realistic redundancy criteria would perhaps be the union of all of them (similar to the metrics defined in Section 3).

**RQ2 – Can separation of test cases in a JUnit test method make redundancy measurement more precise?**

In the testing theory, the granularity of test artifacts only contains test suite and test case. Related works about test minimization, discussed in Section 2, measured redundancy by values 0(non-redundant) or 1(redundant) in test case level.

However, the JUnit test automation framework introduces a few additional middle levels (see Figure 3). In our experiment, we measure redundancy in test method level (lowest implemented level in JUnit). However, following discussion shows that redundancy in this level (with the proposed metrics) can not be measured very precisely.

In the xUnit community [6], a single test method verifying too many functionalities is referred to as an *eager* test, which is a type of test *smell* [20]. This type of test smell is often caused by trying to minimize the number of unit tests by verifying many test conditions in a single test method. While this can be a good practice for manually executed tests to perform less tests, it is not a good practice for fully automated tests in which the test runner tool run all the tests [6].

As many eager test smells exist in real test suites, there are test methods containing more than one test case and among those test cases only some of them are redundant.

**Example (Eager Test):** In JMeter, test method *testCookieMatching* from class *TestCookieManager* in package *org.apache.jmeter.protocol.http.control* has five test cases (Table 6).

First and forth test cases are redundant while others are not. Therefore a partial redundancy metric is required to show the value of 0.4 for this example. The partial redundancy metric of CodeCover [15] provides the value of SR=0.98 for this test method which represents imprecise result. In examples of the next research question, some cases in which using partial redundancy measured by proposed metrics are very risky and may mislead testers are discussed.

**Table 6 – Source code of a test method in JMeter Test Suite**

```
public void testCookieMatching(){
    URL url = new
URL("http://a.b.c:8080/TopDir/fred.jsp");
    man.addCookieFromHeader("ID=abcd; Path=/TopDir",
url);
    String s =man.getCookieHeaderForURL(url);
    assertNotNull(s);
    assertEquals("ID=abcd", s);

    url = new URL("http://a.b.c:8080/other.jsp");
    s=man.getCookieHeaderForURL(url);
    assertNull(s);

    url = new
URL("http://a.b.c:8080/TopDir/suub/another.jsp");
    s=man.getCookieHeaderForURL(url);
    assertNotNull(s);

    url = new URL("http://a.b.c:8080/TopDir");
    s=man.getCookieHeaderForURL(url);
    assertNotNull(s);

    url = new URL("http://a.b.d/");
    s=man.getCookieHeaderForURL(url);
    assertNull(s);
  }
```

**LESSONS LEARNED:** Measuring redundancy can be done more precisely in test case level without introducing partial values. Therefore, we need to separate test cases in one test method or suggest testers to avoid writing eager tests. Otherwise, partial redundancy from the proposed metrics is not precise.

**RQ3 – Can separation of test phases improve the precision of redundancy measurement?**

To avoid eager tests, there might be a lot of tests that needs the same setup. Ironically, another test smell may appear in this case; interacting test smell arises when one test depends in some way on the outcome of another test. This dependency may cause the erratic test smell in which one or more test behave erratically; (the test result depends on the result of other tests) [6].

To avoid erratic tests, each test may cover the same coverable items of SUT at the setup phase as the other tests cover and at last exercise or even verify different artifacts. In this case, intersection between the covered item sets of two tests is caused by share setup so it should not be considered as redundancy.

To prevent the false identification of redundancy, we believe the redundancy of each step should be measured separately. Examples of this section show that calculating the redundancy of each phase separately and then considering all of them to find out about the redundancy of that test case might provide a more precise measure.

Coverage information can be used to measure the redundancy of Setup, Exercise and Tear Down phases.

While Verification phase is for asserting a fact about exercise outcome, and so there is no direct relationship between how SUT is covered and the redundancy of this phase.

Separating setup and exercise phases in one test method is a challenging issue as some statements can be considered as both. Besides, we could not find a coverage tool that can be used to automate measuring coverage information of different phases. It seems that the current coverage tools can be modified to automate such measurements. However, given our project schedule, we were not able to do.

Redundancy in the verification phase can be defined as having the same oracle or not. If two test methods cover same items in the setup and exercise phases, they may still have different oracles (expected outcomes). Automating measurement of redundancy of this phase does not have direct relation with coverage information (SUT is not covered in this phase). Redundancy of this phase can be measured by inspecting the test code which seems to be very challenging (has not done in this work).

**Example 1 (Uncovered Setup):** In Allelogram, test methods *testEmpty* and *testAlleleConstructor* from class *BinGuesserTest* in package *org.carlmanaster.allelogram.model.tests* cover different class constructor methods in the setup phase, while they both cover the same method in the exercise phase.

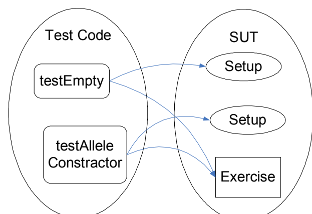


**Figure 7 – Coverage graph of two test methods in Allelogram with different setups and the same exercise**

Figure 7 shows the coverage graph of the above test methods in setup and exercise phases. Since calling different constructors in the setup phase differentiates the goal of these two tests, their real redundancy should be 0. PR value for *testEmpty* with respect to *testAlleleConstructor,* which shows the partial redundancy between these two tests, is 0.5. This example represents cases in which considering partial redundancy can mislead testers. To resolve this problem we believe we need to separate phases. If we do so, the PR of setup and exercise are 0 and 1 respectively. A test is redundant if it is redundant in all of the phases. Therefore, in this example by separating phases PR would be 0.

**Example 2 (Covered Setup, Uncovered Exercise):** In Allelogram, test methods *testToScreen* and *testToData* from class *ScaleTest* in package *org.carlmanaster.allelogram.gui.tests* cover the same method of the SUT in the setup phase, while they cover different parts of the SUT in the exercise. Figure 8 illustrates a hypothetical coverage graph of the above test methods in setup and exercise phases. Since calling different functionality of the system in the exercise phase differentiates the goal of these two tests, the real redundancy should be 0. Similar to previous example in this case partial redundancy may mislead testers. The idea of separating phases in this example generates more precise results.

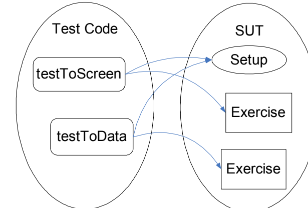


**Figure 8 – Coverage graph of two test methods in Allelogram with the same setup and different exercises**

**Example 3 (Covered Setup, Covered Exercise, Different Verification):** In Allelogram, test method *testAlleleOrderDoesntMatter* from class *GenotypeTest* in package *org.carlmanaster.allelogram.model.tests* covers a subset of covered items by the test method *testOffset* both in setup and exercise phases. However, the assertion goal in *testAlleleOrderDoesntMatter* is completely different from the assertion goal of *testOffset*. Therefore, although the redundancy in setup and exercise is 1, the real redundancy of *testAlleleOrderDoesntMatter* should be 0, while PR value for *testAlleleOrderDoesntMatter* with respect to *testOffset* is 1.

**LESSONS LEARNED:** measuring redundancy in verification phase can not be measured base on coverage information. Therefore, to detect redundancy more precisely we need to separate phases and measure verification redundancy by using more information than coverage based metrics.

**RQ4 – Does the particular implementation of standard code coverage criteria affect accuracy of redundancy measurement?**

CodeCover (the coverage tool used in this work), provides a limited approach to measure statement coverage. This tool implements statement coverage criteria by considering statements that have exactly one coverable item and is covered if it was executed successfully at least once. For instance, throwing an exception (even if it was intentional) is not considered a successful execution.

As it is mentioned in [15] for technical reasons, CodeCover has a problem to instrument `return` and `throw` statements. Hence, the tool excludes covering of such statements from the coverage information. The

following example shows that if covering return statements is not counted in coverage information, it can lead to imprecise redundancy results.

**Example (Return Statement):** In Allelogram, test methods *testContains* and *testToString* from class *BinTest* in package *org.carlmanaster.allelogram.model.tests* cover separate methods in the exercise phase: method *bin.contains()* and *bin.toString()*. Method *bin.toString()* contains only a return statement. As CodeCover does not consider return statements in coverage calculation, the measurement considers *testToString* as a redundant test (PR =1) which is not right.

LESSIONS LEARNED: The coverage tools should be designed in a way that they do not exclude return statements from coverage information. This type of missing value can affect the precision of redundancy measurement.

### 4.4. Summary: Is coverage-based information alone a precise tool for measuring redundancy?

Research questions from Section 4.3 show that measuring test redundancy base on coverage information is vulnerable given the current implementation of JUnit unit test framework and also the CodeCover coverage tool.

Based on the analysis of the question, we can conclude that by considering all different coverage criteria (RQ1), separating test cases in test methods (RQ2), separating test phases in test cases (RQ3) and by using more precise coverage tool (RQ4), we might be able to measure redundancy more precisely.

Using only four coverage criteria measured by CodeCover and not considering required information for measuring redundancy in verification phase are the most important causes of high false-positive error reported in Section 4.2.

For measuring redundancy in test method level, we need to use partial redundancy concept. However, our experience illustrated that this type of metrics based only on coverage information of test method (as proposed in CodeCover) is not precise. To improve that, we need to separate test cases in each test methods. In other hand, there are some cases (e.g., same setup) in which partial redundancy mislead testers more than helping to detect real redundancy.

Manual inspection of the four test suite led us to understand that even by considering all above conditions, the redundancy measured only base on coverage information might not be very precise. The following discussions are the cases of this observation.

Coverage information is calculated only based on the instrumented SUT. External resources (e.g., libraries) are not instrumented (it may not be even possible to instrument all of them as their source codes are not usually available). There are cases in which two test methods cover different libraries. In such cases, the coverage information of the SUT alone is not enough to measure redundancies.

As mentioned earlier in this article, the main goal of finding redundancy is to detect test artifacts that do not increase the potential ability of fault detection. Therefore, faults or exceptions in SUT which are detected by the current test suite should be considered in detecting redundancy. For instance, detecting pair redundancy between two test methods with different test result is not correct. Coverage-based redundancy detection should thus be improved by considering the test results.

Above shortcomings of the coverage-based information suggest not to use that information for finding test redundancy. However, this knowledge is a very useful starting point for inspections. Manual investigation in this experiment could have been much more time consuming without using coverage information and the proposed redundancy metrics.

### 4.5. Threats to Validity

External Validity: Three issues limit the generalization of our results. The first issue is the subject representativeness in the manual inspection activity. Manual redundancy detection has been done by the first author (a graduate student). Test suite developers of the SUTs may detect redundancy differently, and more precisely than us. The second issue is the number of test methods that were inspected manually. As inspecting a test method to find out the redundancy is very time consuming, we only covered about 37% of the test suite of one of our SUTs. The third issue is the object program representativeness. The four SUTs are random projects chosen from the open source community. Other industrial programs with different characteristics may have different test redundancy behavior.

Internal Validity: In addition to the factors discussed in Sections 4.3 and 4.4, potential faults in manual redundancy detection could have an impact on our results.

Construct Validity: The redundancy metric that we have introduced is not the only possible measures for test redundancy. For example, we can consider fault detection effectiveness of each test method by mutating the SUTs and then distinguishing (killing) the mutants by executing the test methods to analyze redundancy.

## 5. Conclusions and Future Works

Detecting and resolving test redundancy leads to have integrity in test suite and decreases the cost of

software maintenance. Previous works on test set minimization, believed that coverage information is useful resource to detect redundancy. In this work we performed an experiment to evaluate the above idea.

The result of our experiment shows that coverage information is not enough knowledge for detecting redundancy according to fault detection effectiveness. We found some conditions under which we can improve this approach.

Although test redundancy measurement based on coverage information has a few shortcomings, reducing search space of tests and helping to find appropriate tests for comparison in manual inspections are some benefits of this approach.

To improve the precision of redundancy measurement base on coverage information, the required automation such as separating test cases or test phases are considered as the future works of this project. Improving redundancy detection by applying more coverage criteria (e.g., path coverage, black-box coverage) and by using more precise coverage tool is also considered as future work of this project.

As discussed in this article, redundancies in some test levels like test method can be partial. The partial redundancy concept in literature is not precise and need to be improved in future.

## Acknowledgements

The authors were partially supported by the NSERC. Vahid Garousi was further supported by an Alberta Ingenuity new faculty award.

## References

[1] R. Reichhart and T. Girba, "Rule-based Assessment of Test Quality," *Object Technology,* vol. 6, no. 9, 2007.

[2] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does Code Decay? Assessing the Evidence from Change Management Data," *IEEE Trans. on Software Eng.,* vol. 27, no. 1, pp. 1-12, 2001.

[3] D. L. Parnas, "Software Aging," *Proc. of Int. Conf. on Software Eng.*, pp. 279-287 1994.

[4] B. V. Rompaey, B. D. Bois, and S. Demeyer, "Improving test code reviews with metrics: a pilot study," *Technical report, Lab On Re-Eng., University of Antwerp,* 2006.

[5] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*: Addison Wesley, 1999.

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

[7] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Transactions on Soft. Eng. and Methodology (TOSEM),* vol. 2, no. 3, pp. 270 - 285, 1993.

[8] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong, "An empirical study of the effects of minimization on the fault detection capabilities of test suites," *Proc. of Int. Conf. on Soft. Maintenance*, pp. 34 - 43, 1998.

[9] A. v. Deursen, L. Moonen, A. v. d. Bergh, and G. Kok, "Refactoring Test Code," *Proc. of Int. Conf. on eXtreme Programming and Flexible Processes in Soft. Eng. XP2001*, 2001.

[10] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *Soft. Eng. IEEE Transactions* vol. 29, no. 3, pp. 195 - 209, 2003.

[11] A. J. Offutt, J. Pan, and J. M. Voas, "Procedures for Reducing the Size of Coverage-based Test Sets " *In Proc. of Int'l Conf. on Testing Comp. Soft.*, pp. 111-123, 1995.

[12] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," *Software—Practice & Experience,* vol. 28, no. 4, pp. 347 - 369, 1998.

[13] W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini, "Test set size minimization and fault detection effectiveness: acase study in a space application," *Proc. of Int. Conf. on Computer Soft. and Applications*, pp. 522 - 528, 1997.

[14] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*, 1990.

[15] T. Scheller, "CodeCover. http://codecover.org," 2007.

[16] Martin R. and Martin M., "Fitnesse. http://www.fitnesse.org," Retrieved on 02/27/2008.

[17] M. Patricios, "strategy game framework. http://lurgee.net/home/," Retrieved on 08/10/2008.

[18] C. Manaster, "Allelogram. http://code.google.com/p/allelogram/," Retrieved on 08/20/2008.

[19] A. S. Foundation, "JMeter. http://jakarta.apache.org/jmeter/," Retrieved on 09/02/2008.

[20] B. V. Rompaey, B. D. Bois, S. Demeyer, and M. Rieger, "On The Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test " *IEEE Transactions on Soft. Eng.,* vol. 33, no. 12, pp. 800-817, 2007.