Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2010, Article ID 932686, 13 pages
doi:10.1155/2010/932686

Research Article

A Tester-Assisted Methodology for Test Redundancy Detection

Negar Koochakzadeh and Vahid Garousi

Software Quality Engineering Research Group (SoftQual), Department of Electrical and Computer Engineering, Schulich School of Engineering, University of Calgary, Calgary, AB, Canada T2N 1N4

Correspondence should be addressed to Negar Koochakzadeh, nkoochak@ucalgary.ca

Received 15 June 2009; Revised 16 September 2009; Accepted 13 October 2009

Academic Editor: Phillip Laplante

Test redundancy detection reduces test maintenance costs and also ensures the integrity of test suites. One of the most widely used approaches for this purpose is based on coverage information. In a recent work, we have shown that although this information can be useful in detecting redundant tests, it may suffer from large number of false-positive errors, that is, a test case being identified as redundant while it is really not. In this paper, we propose a semiautomated methodology to derive a reduced test suite from a given test suite, while keeping the fault detection effectiveness unchanged. To evaluate the methodology, we apply the mutation analysis technique to measure the fault detection effectiveness of the reduced test suite of a real Java project. The results confirm that the proposed manual interactive inspection process leads to a reduced test suite with the same fault detection ability as the original test suite.

1. Introduction

In today's large-scale software systems, test (suite) maintenance is an inseparable part of software maintenance. As a software system evolves, its test suites need to be updated (maintained) to verify new or modified functionality of the software. That may cause test code to erode [1, 2]; it may become complex and unmanageable [3] and increase the cost of test maintenance. Decayed parts of test suite that cause test maintenance problems are referred to as test *smells* [4].

Redundancy (among test cases) is a discussed but a seldom-studied test smell. A redundant test case is one, which if removed, will not affect the fault detection effectiveness of the test suite. Another type of test redundancy discussed in the literature (e.g., [5, 6]) is test code duplication. This type of redundancy is similar to conventional source code duplication and is of syntactic nature. We refer to the above two types of redundancy as semantic and syntactic test redundancy smells, respectively. In this work, we focus on the semantic redundancy smell which is known to be more challenging to detect in general than the syntactic one [5].

Redundant test cases can have serious consequences on test maintenance. By modifying a software unit in the maintenance phase, testers need to investigate the test suite to find all relevant test cases which test that feature and update them correctly with the unit. Finding all of the related test cases increases the cost of maintenance. From the other hand, if test maintenance (updating) is not conducted carefully, the integrity of the entire test suite will be under question. For example, we can end up in a situation in which two test cases test the same features of a unit, if one of them is updated correctly with the unit and not the other one, one test may fail while the other may pass, making the test results ambiguous and conflicting.

The motivation for test redundancy detection is straightforward. By detecting and dealing with redundant test case (e.g., carefully removing them), we reduce test maintenance cost and the risk of loosing integrity in our test suite, while fault detection capability of our test suite remains constant.

One of the most widely used approaches in the literature (e.g., [6–11]) for test redundancy detection, also referred to as test minimization, is based on coverage information. The rationale followed is that, if several test cases in a test suite execute the same program elements, the test suite can then be reduced to a smaller suite that guarantees equivalent test coverage ratio [6].

However, test redundancy detection based on coverage information does not guarantee to keep fault detection capability of a given test suite. Evaluation results from our previous work [12] showed that although coverage information can be very useful in test redundancy detection, detecting redundancy only based on this information may lead to a test suite which is weaker in detecting faults than the original one.

Considering fault detection capability of a test case for the purpose of redundancy detection is thus very important. To achieve this purpose, we propose a collaborative process between testers and a proposed redundancy detection engine to guide the tester to use valuable coverage information in a proper and useful way.

The output of the process is a reduced test suite. We claim that if testers play their role carefully in this process, fault detection effectiveness of this reduced test set would be equal to the original set.

High amount of human effort should be spent on inspecting a test suite manually. However, the proposed process in this paper tries to use the coverage information in a constructive fashion to reduce the required tester efforts. More automation can be added to this process later to save more cost and thus the proposed process should be considered as the first step to reduce required human effort for test redundancy detection.

To evaluate our methodology, we apply the mutation technique in a case study in which common types of faults are injected. Then original and reduced test set are then executed to detect faulty versions of the systems. The results show similar capability of fault detection for those two test sets.

The remainder of this paper is structured as follows. We review the related works in Section 2. Our recent previous work [12] which evaluated the precision of test redundancy detection based on coverage information is summarized in Section 3. The need for knowledge collaboration between human testers and the proposed redundancy detection engine is discussed in Section 4. To leverage and share knowledge between the automated engine and human tester, we propose a collaborative process for redundancy detection in Section 5. In Section 6, we show the results of our case study and evaluate the results using the mutation technique. Efficiency, precision, and a summary of the proposed process are discussed in Section 7. Finally, we conclude the paper in Section 8 and discuss the future works.

2. Related Works

We first review the related works on test minimization and test redundancy detection. We then provide a brief overview of the literature on semiautomated processes that collaborate with software engineers to complete tasks in software engineering and specifically in software testing.

There are numerous techniques that address test suite minimization by considering different types of test coverage criteria (e.g., [6–11]). In all of those works, to achieve the maximum possible test reduction, the smallest test set which covers the same part of the system was created [7]. The problem of finding the smallest test set has been shown to be NP-complete [13]. Therefore, in order to find an approximation to the minimum cardinality test set, heuristics are usually used in the literature (e.g., [7, 9]).

A few works have applied data flow coverage criteria (e.g., [7, 10]) while a few others have applied control flow criteria (e.g., [6, 9, 11]).

In [7], in addition to the experiment which was performed for all-definition-use coverage criterion on a relatively simple program (LOC is unknown), the authors mentioned that all the possible coverage criteria should be considered in order to detect redundant test cases more precisely. The authors were able to reduce 40% of the size of the test suite under study based on coverage information.

Coverage criteria used in [10] were predicate-use, computation-use, definition-use, and all-uses. The authors applied their approach on 10 Unix programs (with average LOC of 354) and 91% of the original test suites were reduced in total.

The control flow coverage criteria used in [6, 9, 11] are Branch [6], statement [9], and MC/DC [11]. In [9], mutation analysis was used to assess and evaluate the fault detection effectiveness of the reduced test suites. The ratios of reduction reported in these works were 50%, 34%, and 10%, respectively. The Systems Under Tests (SUTs) used in [6, 9] were small scale (avg. LOC of 29 and 231, resp.), while [11] used a medium size space program as its SUT with 9,564 LOC.

The need to evaluate test redundancy detection by assessing fault detection effectiveness was mentioned in [6, 11]. In those works, faults were manually injected into the SUTs to generate mutants. Then the mutation scores of original and reduced test sets were compared. Reference [6] concludes that test minimization based on coverage information can reduce the ability of fault detection, while [11] showed opposite conclusions.

In [6], faults were seeded to the SUTs manually by modifying mostly a single line of code (first order mutation), while in a few other cases, the authors modified between two and five lines of code (k-order mutation). As mentioned in [6], ten people (mostly without knowledge of each other's work) had tried to introduce faults that were as realistic as possible, based on their experience with real programs.

In [11], the manually injected faults (18 of them) were obtained from the error-log maintained during its testing and integration phase. Eight faults were in the "logic omitted or incorrect" category, seven faults belong to the type of "computational problems," and the remaining three faults had "data handling problems" [11].

In our previous work [12], an experiment was performed with 4 real Java programs to evaluate coverage-based test redundancy detection. The objects of study were JMeter, FitNesse, Lurgee and Allelogram with LOC of 69,424, 22,673, 7,050, and 3,296, respectively. Valuable lessons learned from our previous experiment revealed that coverage information cannot be the only source of knowledge to precisely detect test redundancy. Lessons are summarized in Section 3 of this paper.

To the best of the authors' knowledge, there has been no existing work to improve the shortcomings (imprecision) of coverage-based redundancy detection. In this paper, we are proposing a semiautomated process for this purpose.

Semiautomated decision supports systems leverage human-computer interaction which put together the knowledge of human users and intelligent systems to support decision-making tasks. Hybrid knowledge is very effective in such situations where the computational intelligence provides a set of qualified and diversified solutions and human experts are involved interactively in the decision-making process for final decision [14].

A logical theory of human-computer interaction has been suggested by Milner [15]. Besides, the ways in which open systems' behavior can be expressed by the composition of collaborative components is explained by Arbab [16]. There are various semiautomated systems designed for software engineering such as user-centered software design [17].

There have also been semiautomated systems used specifically in software testing. For instance, test case generation tools require tester's assistance in providing test oracles [18]. Another example of collaborative tool for testing is manual testing frameworks [19]. In these tools, testers perform test cases manually while system records them for later uses. The process proposed in this paper is a semiautomated framework with the purpose of finding test redundancy in software maintenance phase.

3. Coverage-Based Redundancy Detection Can Be Imprecise

In our previous work [12], we performed an experiment to evaluate test redundancy detection based only on coverage information. We formulated two experimental metrics for coverage-based measurement of test redundancy in the context of JUnit test suites. We then evaluated the approach by measuring the redundancy of four real Java projects (FitNesse, Lurgee, Allelogram, and JMeter). The automated test redundancy measures were compared with manual redundancy decisions derived from inspection performed by a human software tester.

In this paper, we use the term *test artifact* for different granularity levels supported in JUnit (Figure 1). Three levels of package, class, and methods are grouping mechanism for test cases that have been introduced in JUnit.

The results from that study [12] showed that measuring test redundancy based only on coverage information is vulnerable to imprecision given the current implementation of JUnit unit test framework and also coverage tools. The following discussion explains the root causes.

In the SUTs we analyzed in [12], about 50% of test artifacts, manually recognized as nonredundant, had been detected as redundant tests by our coverage-based redundancy metrics. In a Venn diagram notation, Figure 2 compares a hypothetical original test set with two reduced sets showing high number of false-positive errors. Three main reasons discovered in [12] to justify the errors are discussed next.

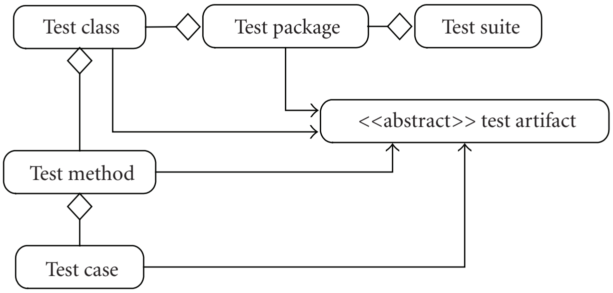


Figure 1: Test granularity in JUnit.

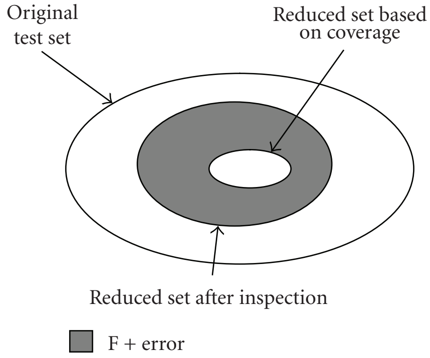


Figure 2: False-Positive Error in Test Redundancy Detection based on Coverage Information.

(1) Test redundancy detection based on coverage information in all previous works have been done by only considering limited number of coverage criteria. This fact that two test cases may cover the same part of SUT according to one coverage criterion but not the other one causes impreciseness in test redundancy detection only by considering one coverage criterion.

(2) In JUnit, each test case contains four phases: setup, exercise, verify, and teardown [4]. In the setup phase the required state of the SUT for the purpose of a particular test case is setup. In the exercise phase, the SUT is exercised. In the teardown phase the SUT state is rolled back into the state before running the test. In these three phases SUT is covered while in the verification phase only a comparison between expected and actual outputs is performed and SUT is not covered. Therefore, there might be some test cases with the same covered part of SUT with various verifications. In this case, coverage information may lead to detecting a nonredundant test as redundant.

(3) Coverage information is calculated only based on the SUT instrumented for coverage measurement. External resources (e.g., libraries) are not usually instrumented. 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.

Another reason of impreciseness in redundancy detection based on coverage information mentioned in [12]

```
public void testAlleleOrderDoesntMatter () {
  Genotype g1 = new Genotype(new double [ ] {0,1});
  Genotype g2 = new Genotype(new double [ ] [13]);
  assertTrue(g1.getAdjustedAlleleValues (2).
          equals(g2.getAdjustedAlleleValues (2)));
}
public void testOffset (){
  Genotype g = new Genotype(new double [ ]{0,1});
  g.offsetBy (0.5);
  List<Double> adjusted =
  g.getAdjustedAlleleValues (2);
  assertEquals(2, adjusted.size ());
  assertEquals(0.5, adjusted.get (0));
  assertEquals(1.5, adjusted.get (1));
  g.clearOffset();
  adjusted = g.getAdjustedAlleleValues (2);
  assertEquals(0.0, adjusted.get (0));
  assertEquals(1.0, adjusted.get (1));
}
```

Algorithm 1: Source code of two test methods in the Allelogram test suite.

was some limitations in coverage tools implementation. For example, the coverage tool that we used in [12] was *CodeCover* [20]. The early version of this tool (version 1.0.0.0) was unable to instrument `return` and `throw` statements due to a technical limitation. Hence, the earlier version of the tool excluded covering of such statements from coverage information. This type of missing values can lead to detecting a nonredundant test as redundant. However, this limitation has now been resolved in the newest version of CodeCover (version 1.0.0.1 released on April 2009) and we have updated our redundancy detection framework by using the latest version of this tool. Since in [12] this problem was a root cause of false positive error, here we just report this as a possible reason of impreciseness in redundancy detection, while in this paper we do not have this issue.

Algorithm 1 shows the source code of two test methods from *Allelogram* test suite as an example of incorrect redundancy detection by only applying coverage information. In this example, test method *testAlleleOrderDoesntMatter* covers a subset of covered items by the test method *testOffset* both in setup and exercise phases. The setup phase includes calling *Genotype(new double)* constructor. The exercise phase contains calling *getAdjestedAlleleValues(int)* method by passing the created *Genotype* object, which both are called in the second test method as well. However, the assertion goal in the first test is completely different from the assertion goal in the second one. In the first test method, the goal is comparing the output value of *getAdjestedAlleleValues* method for two *Genotype* objects, while in second one, one of the goals is checking the size of output list from the *getAdjestedAlleleValues* method. Therefore, although according to coverage information the first test method is redundant, in reality it is nonredundant.

4. The Need for Knowledge Collaboration with Testers

Reduced test set based on coverage information contains those test artifacts that cover at least one coverable item not covered by any other test artifact. Therefore these test artifacts contribute to achieving more coverage and according to the concept of test coverage, they may increase the fault detection capability of the test suites.

Based on the above discussion, it is worthwhile to use coverage information for test redundancy detection to reduce the number of test artifacts that might be redundant.

On the other side, high ratio of false-positive errors shows that the coverage-based results alone are not reliable and we may inaccurately detect many nonredundant test artifacts as redundant ones.

The above advantages and disadvantages of coverage-based redundancy detection have motivated us to improve the test redundancy detection process by leveraging knowledge from human testers. The three main root causes of imprecision discussed in Section 3 should be considered in such a tester-assisted approach.

First, the more coverage criteria are applied, the more precise test redundancy will be detected. However, all of the existing test coverage tools support a limited number of coverage criteria. White-box criteria are more usually supported, while there are only a few tools supporting black-box criteria (e.g., JFeature [21]). In addition, usually there are no precise formal specifications for some units in some systems. Thus, automated measurement of black-box coverage is impossible in those cases. Also, there is a lack of coverage tools which automatically measure both white-box and black-box coverage criteria at the same time. Combing the coverage results from various coverage tools might be a solution. However, lack of formal specification for many real projects makes it very challenging for us testers to consider automated measurement of black-box coverage for the purpose of redundancy detection in this work []. For projects with full formal specifications, if test minimization is performed precisely with respect to all available coverage criteria, loss of fault detection ability can be minimized or eliminated altogether. However, since formal specifications are not available for many real projects, we propose to involve human testers in the process of test redundancy detection.

For this purpose, testers can use their knowledge to write formal specification for the SUT and use them in black-box coverage tools, or apply black-box coverage manually. For instance, if test t_1 covers a subset of covered items by t_2, and the main goal of t_1 is to check whether there is an exception thrown by the SUT while t_2 has a different goal, t_1 is not redundant. In other words, the inputs of two above tests are from different equivalence classes (i.e., a black-box coverage criterion should be applied).

Second, the verification phase of JUnit test methods should be analyzed separately. As explained in Section 3, this phase is independent of coverage information, and is thus a precision threat to redundancy detection. Assertion statements in JUnit tests should be compared to find if they cause redundancy or not. In some cases, the *actual* and *expected* values in assert statements have complicated data flow. In such cases, comparing assertions in verification phase would require sophisticated source code analysis (e.g., data flow analysis). For example, the actual outcomes of the two `assertEquals` statements (located in two test methods) in Figure 3 are the same: `adjusted.get()`. However, determining whether their expected outcomes (`a` and 1.5) have the same value or not would require data flow analysis in this example. Automating such an analysis is possible, but is challenging while in this step we use human tester for this purpose by leaving its automation as a future work.

Third, in addition to the SUT, all the external libraries used should be considered. However, as the source code of those libraries is not probably available, we need to instrument the class files in Java systems or to monitor coverage through the JVM. As per our investigations, automating this instrumentation and calculating coverage information for the external libraries and combining them with coverage information of the source code of the SUT is challenging and is thus considered as a future work. At this step, we propose the human tester to analyze the test code to find out how an external library affects test results and consider that in comparing test artifacts.

As explained previously, although it is possible to increase the degree of automation to cover the shortcoming of redundancy detection only based on limited number of coverage criteria, there is one main reason that does not allow full automation for this process, which is the lack of precise and formal specification for real world project. In other words, in the process of test redundancy detection the existence of human testers is necessary to confirm the real redundancy of those test artifacts detected as redundant by the system. The human tester has to conduct a manual inspection with guidelines proposed in this work and has to consider the three root causes to prevent false positive errors.

```
...
double a = getDefaultAdjusted(0);
...
assertEquals(a, adjusted.get(0));
...
```

```
...
assertEquals(1.5, adjusted.get(0));
...
```

Figure 3: The challenge of comparing assertions: excerpts from the test suite of Allelogram.

Using the three above guidelines helps testers to collaborate more effectively in the proposed redundancy detection process by analyzing test codes. Testers who have developed test artifacts are the best source of knowledge to decide about test redundancy by considering the above three lessons. However, other test experts can also use our methodology to find the redundancy of a test suite through manual inspection. For instance, in the experiment of this work, the input test suite was created by the developers of an open source project while the first author has performed the process of test redundancy detection.

5. A Collaborative Process for Redundancy Detection

To systematically achieve test redundancy detection with lower false-positive error, we propose a collaborative process between an automated redundancy detection system and human testers. The system will help the tester to inspect test artifacts with the least required amount of effort to find the actually redundant tests by using the benefits from coverage information while the fault detection capability of the reduced test suite is not reduced.

Figure 4 illustrates the activity diagram of the proposed interactive redundancy detection process. The input of this process is the original test suite of a SUT. Since human knowledge is involved, the precision of the inspection conducted by the human tester is paramount. If the tester follows the process and the three above guidelines carefully, the output would be a reduced test with the same fault detection effectiveness as the original one.

As the first step in this process, redundancy detection system uses a coverage tool to calculate coverage information, which is used later to calculate two redundancy metrics (discussed next).

Two redundancy metrics were proposed in [12]: Pair Redundancy and Suite Redundancy. The Pair Redundancy is defined between two test artifacts and is the ratio of covered items in SUT by the first test artifact with respect to the second one. In Suite Redundancy, this ratio is considered for one test artifact with respect to all other tests in the test suite.

Equations (1) and (2) define the Pair and Suite Redundancy metrics, respectively. In both of these equations, $CoveredItems_i(t_j)$ is the set of code items (e.g., statement and branch) covered by test artifact t_j, according to a given

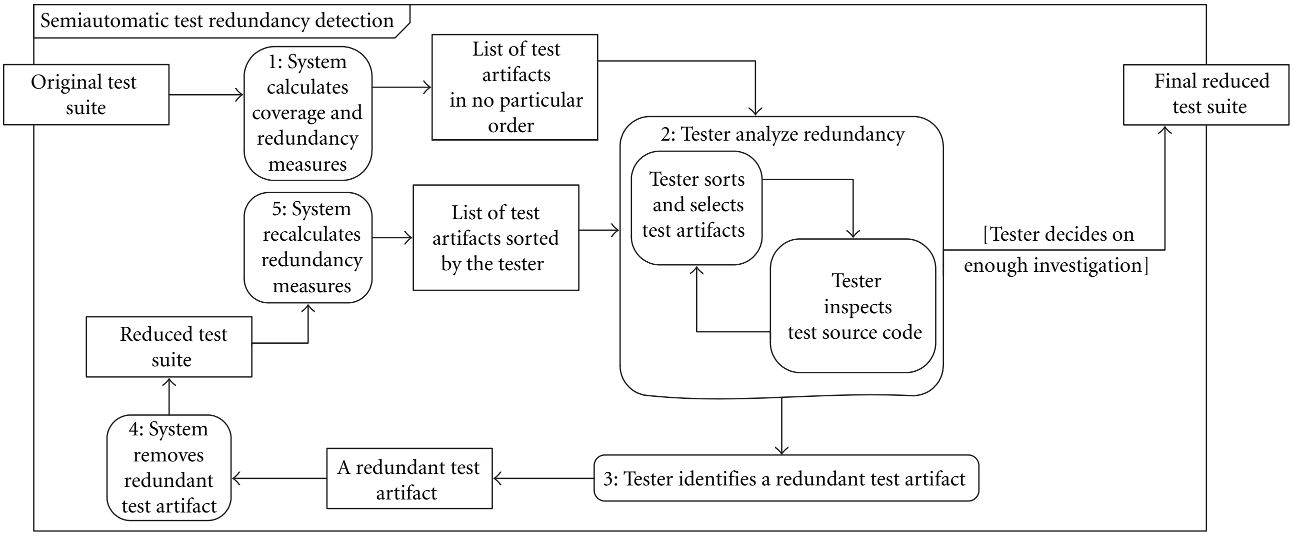

Semiautomatic test redundancy detection
Original test suite
1: System calculates coverage and redundancy measures
List of test artifacts in no particular order
2: Tester analyze redundancy
Tester sorts and selects test artifacts
Tester inspects test source code
[Tester decides on enough investigation]
Final reduced test suite
5: System recalculates redundancy measures
List of test artifacts sorted by the tester
Reduced test suite
4: System removes redundant test artifact
A redundant test artifact
3: Tester identifies a redundant test artifact

Figure 4: Proposed collaborative process for test redundancy detection.

coverage criterion i (e.g., statement coverage). *CoverageCriteria* in these two equations is the set of available coverage criteria used during the redundancy detection process.

Based on the design rationale of the above metrics, their values are always a real number in the range of $[0 \cdots 1]$. This enables us to measure redundancy in a quantitative domain (i.e., partial redundancy is supported too).

However, the results from [12] show that this type of partial redundancy is not precise and may mislead the tester in detecting the redundancy of the test. For instance, suppose that two JUnit test methods have similar setups with different exercises. If for example 90% of the test coverage is in the common setup the pair redundancy metrics would indicate that they are 90% redundant with respect to each other. However different exercises in these tests separate their goals and thus they should not be considered as redundant with respect to each other while 90% redundancy can mislead the tester about their redundancy.

Equation (1) shows Redundancy of test artifact ($\mathbf{t_j}$) with respect to another one ($\mathbf{t_k}$):

$$\mathrm{PR}\left(t_j, t_k\right) = \left(\sum_{i \in CoverageCriteria} \left| CoveredItems_i\left(t_j\right) \cap CoveredItems_i(t_k) \right| \right) / \left(\sum_{i \in CoverageCriteria} \left| CoveredItems_i\left(t_j\right) \right| \right), \tag{1}$$

equation (2) shows Redundancy of one test artifact (t_j) with respect to all others:

$$\mathrm{SR}\left(t_j\right) = \left(\sum_{i \in CoverageCriteria} \left| CoveredItems_i\left(t_j\right) \cap CoveredItems_i\left(\mathrm{TS} - t_j\right) \right| \right) / \left(\sum_{i \in CoverageCriteria} \left| CoveredItems_i\left(t_j\right) \right| \right). \tag{2}$$

However, partial redundancy concept can be useful in some cases to warn testers to refactor test code. To find these cases, in [12], we have offered to separate phases in a test case. As this approach is considered as a future work, in this work we do not consider partial redundancy concept. A test artifact can be redundant or nonredundant. The suite redundancy metric is used as a binary measure to separate test artifacts into these two groups: redundant, and nonredundant. If SR value of a test artifact = 1, that test is considered as redundant otherwise it is nonredundant.

In some cases, a test artifact does not cover any type of items (according to the considered coverage criteria). In [12], we have found that these cases may occur for various reasons, for example, (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 (3) 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 the 0-divide-by-0 problem). We assign the value of NaN (Not a Number) to the SR metric for these cases leaving them to be manually inspected to determine the reason.

After calculating coverage and redundancy metrics, the system prepares a list of test artifacts in no particular order. All the information about coverage ratios, number of covered items and redundancy metrics (both SR for each test and PR for each test pair) is available for exploration by the tester.

Step 2 in the process is the tester's turn. He/she should inspect the tests which are identified as a redundant test by the SR value (=1) to find out whether they are really redundant or not. This manual redundancy analysis should be performed for each test artifact separately. Therefore tester needs to choose a test from a set of candidate redundant tests.

The sequence in which test artifacts are inspected may affect the final precision of the process. Test sequencing often becomes important for an application that has internal state. Dependency between test artifacts may cause the erratic test smell in which one or more tests behave erratically (the test result depends on the result of other tests) [4]. However, in this work we do not consider this smell (our case study does not have this problem and thus we did not have any constraints for sequencing the test artifacts).

Our experience with manual redundancy detection in our case study (discussed in next section) helps us to find that the locality principle of test artifacts is an important factor that should be considered in test sequencing. In other words, for instance, test methods inside one test class have more likelihood of redundancy with respect to each other and should be inspected simultaneously.

There can be different strategies for ordering test artifacts and picking one to inspect at a time. One strategy can be defined according to number of covered items by each test artifact. As discussed next ascending and descending orders of number of coverage items each may have their own benefits.

A test expert may prefer to first choose a test artifact with higher redundancy probability. In this case, we hypothesize that the ascending order based on number of covered items is more suitable. The rationale behind this hypothesis is that the likelihood of covering fewer code items (e.g., statement, branch) by more than one test artifact is more than covering more items by the same test artifacts. Relationship between numbers of covered items by a test artifact with probability of redundancy of that test needs to be analyzed in an experiment. However, this is not the main goal of this paper and we leave it as a future work.

Descending order can have its own benefits. A test expert may believe that having test cases with more covered items would lead to the eager test smell (i.e., a test with too many assertions [22]). In this case, he/she would prefer to first analyze a test that covers more items in the SUT.

Finding a customized order of two above extreme cases by considering their benefits and costs is not discussed in this paper. Also other factors more than redundancy and coverage information may be useful in finding a proper test order.

Another strategy for sorting the existing test cases would be according to their execution time. If one of the objectives of reducing test suite is reducing the execution time, by this strategy test cases which need more time to be executed have more priority of redundancy candidates. However, we believe that in unit testing level execution time of test cases is not as important as other smells like being eager.

After picking appropriate test artifact, tester can use PR values of that test with respect to other tests. This information guides tester to inspect source code of that test case and compare it with source code of those tests with higher PR values. Without this information, manual inspection would take much more time from testers since he/she may not have any idea how to find another test to compare the source code together.

As discussed in Section 4, the main reason of need for human knowledge is to cover shortcomings of coverage-based redundancy detection. Therefore testers should be thoroughly familiar with these shortcomings and attempt at covering them.

After redundancy analysis, the test is identified as redundant or not. If it was detected as redundant by tester (Step 3), system removes it from original test set (Step 4). In this step, the whole collaborative process between system and tester should be repeated. Removing one test from test suite changes the value of $CoveredItems_i(TS - t_j)$ in (2). Therefore system should recalculate Suite Redundancy metric for all of the available tests (Step 5). In Section 6 we show how removing a redundant test detected by tester and recalculating the redundancy information can help the tester not to be misled by initial redundancy information and reduce the required effort of the tester.

Stopping condition of this process depends on tester's discretion. To find this stopping point, tester needs to compare the cost of process with savings in test maintenance costs resulting from test redundancy detection. Process cost at any point of the process can be measured by the time and effort that testers have spent in the process.

Test maintenance tasks have two types of costs which should be estimated: (1) costs incurred by updating (synchronizing) test code and SUT code, and (2) costs due to fixing integrity problems in test suite (e.g., one of two test cases testing the same SUT feature fails, while the other passes). Having redundant tests can lead testers to updating more than a test for each modification. Secondly, as a result of having redundant tests, the test suites would suffer from integrity issues, since the tester might have missed to update all the relevant tests.

To estimate the above two cost factors, one might perform change impact analysis on the SUT, and subsequently effort-prediction analysis (using techniques such as [23]) on SUT versus test code changes.

To decide about stopping point of the process, a tester would need to measure the process costs spent so far and to also estimate the maintenance costs containing both the above-discussed cost factors. By comparing them, he/she may decide to either stop or to continue the proposed process.

In the outset of this work, we have not systematically analyzed the above cost factors. As discussed before, we suggest testers to inspect all the tests with the value SR = 1 as many as possible. However, according to high number

Table 1: The size measures of Allelogram code.

SLOC	Number of packages	Number of classes	Number of methods
3,296	7	57	323

Table 2: The size measures of Allelogram test suite.

Test suite SLOC	Number of test packages	Number of test classes	Number of test methods
2,358	6	21	82

of false-positive errors, other tests in this category (with SR = 1) which were not inspected, should be considered as nonredundant. If the SR metric of a test artifact is less than 1, it means that there are some items in the SUT which are covered only by this test artifact. Thus, they should also be considered as nonredundant.

To automate the proposed process for test redundancy detection, we have modified the CodeCover coverage tool [20] to be able to measure our redundancy metrics. We refer to our extended tool as *TeReDetect* (Test Redundancy Detection tool). The tool shows a list of test artifacts containing coverage and redundancy information of each of them, it lets the tester to sort test artifacts according to his/her strategy (as explained before) and to introduce a real detected redundant test to the system for further metrics recalculation. After detecting a redundant test method, system automatically recalculates the redundancy metrics and updates the tester with new redundancy information for the next inspection iteration. A snapshot of the *TeReDetect* tool, during the process being applied to Allelogram, is shown in Figure 5. *TeReDetect* is an open source project (it has been extended to the SVN repository of CodeCover http://codecover.svn.sourceforge.net/svnroot/codecover). *TeReDetect* is not a standalone plug-in, rather it has been embedded inside the CodeCover plug-in. For instance, *ManualRedundancyView.java* is one of the extend-ed classes for our tool which is available from http://codecover.svn.sourceforge.net/svnroot/codecover/trunk/code/eclipse/src/org/codecover/eclipse/views/.

6. Case Study

6.1. Performing the Proposed Process. We used Allelogram [24], an open-source SUT developed in Java, as the object of our case study. Allelogram is a program for processing genomes and is used by biological scientists [24]. Table 1 shows the size measures of this system.

The unit test suite of Allelogram is also available through its project website [24] and is developed in JUnit. Table 2 lists the size metrics of its test suite. As the lowest implemented test level in JUnit is test method, we applied our redundancy detection process on the test method level in this SUT.

As the first step of proposed redundancy detection process, coverage metrics are measured. For this purpose, we used the CodeCover tool [20] in our experiment. This tool is an open-source coverage tool written in Java supporting the following four coverage criteria: 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.

Table 3: Coverage information (%).

	Coverage (%)			
	Statement	Branch	Condition	Loop
Entire Allelogram	23.3	34.7	35.9	22.2
Without GUI components	68.0	72.9	71.4	43.0

Table 4: The percentage of fully redundant test methods.

Coverage criteria	Percentage of fully redundant test methods
Statement	77%
Branch	84%
Condition	83%
Loop	87%
All	69%

Table 3 shows the coverage metrics for our SUT. The first row in this table is the coverage ratios of the whole Allelogram system which are relatively low. We also looked at the code coverage of different packages in this system. Our analysis showed that the Graphical User Interface (GUI) package of this SUT is not tested (covered) at all by its test suite. This is most probably since JUnit is supposed to be used for unit testing and not GUI or functional testing. By excluding the GUI package from coverage measurement, we recalculated the coverage values shown in the second row of Table 3. These values show that the non-GUI parts of the system were tested quite thoroughly.

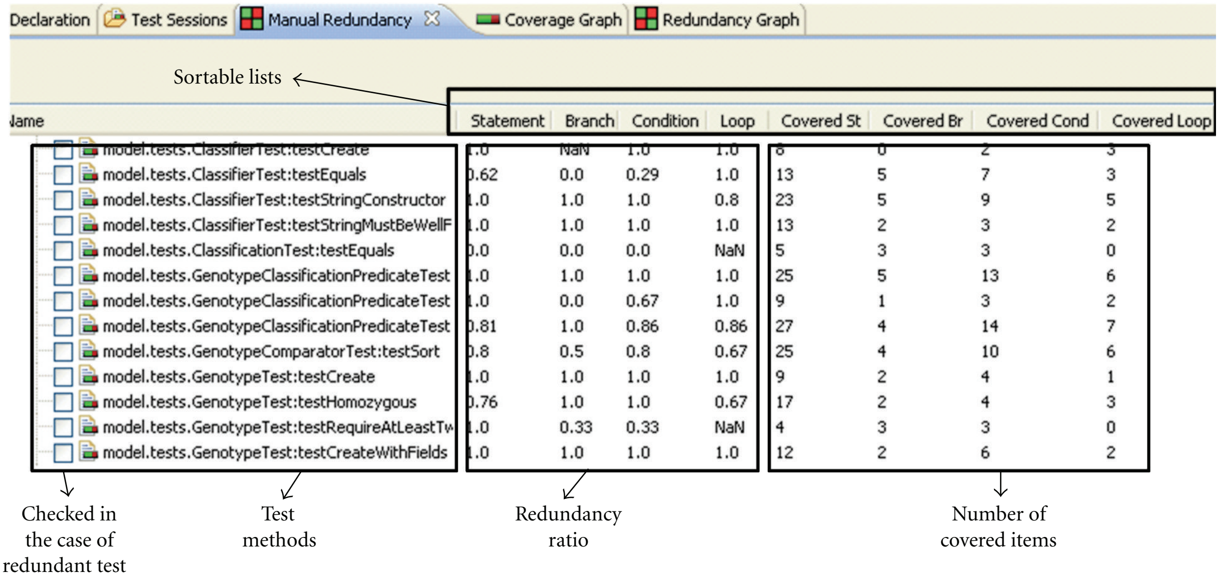
The next step in the process is the calculation of suite-level redundancy for each test method and pairwise redundancy for each pair of test methods in the test suite of our SUT.

To automate the measurement of redundancy of each test method using the two metrics defined in Section 5 ((1) and (2)), we have modified CodeCover to calculate the metrics and export them into a text file, once it executes a test suite.

Table 4 reports the percentage of fully redundant test methods (those with SR = 1) according to each coverage criterion and also by considering all of the criteria together.

As we expected, according to Table 4, ratio of full redundancy detected by considering each coverage criteria separately is higher than the case when all of them are considered. This confirms the fact that the more coverage criteria used in redundancy detection, the less false positive error can be achieved. In other words, *All* coverage criterion detects those tests as nonredundant that improve the coverage ratio values of at least one of the coverage criteria. As *All* criterion is more precise than the others, in the rest of our case study we consider the suite redundancy based on *All* criterion.

According to the suite redundancy result by considering all four coverage criteria (Table 4), 31% (100 − 69) of the tests in test suites of Allelogram are nonredundant. To confirm the nonredundancy of those methods, we randomly sampled



Sortable lists

Name	Statement	Branch	Condition	Loop	Covered St	Covered Br	Covered Cond	Covered Loop
□ model.tests.ClassifierTest:testCreate	1.0	NaN	1.0	1.0	8	0	2	3
□ model.tests.ClassifierTest:testEquals	0.62	0.0	0.29	1.0	13	5	7	3
□ model.tests.ClassifierTest:testStringConstructor	1.0	1.0	1.0	0.8	23	5	9	5
□ model.tests.ClassifierTest:testStringMustBeWellF	1.0	1.0	1.0	1.0	13	2	3	2
□ model.tests.ClassificationTest:testEquals	0.0	0.0	0.0	NaN	5	3	3	0
□ model.tests.GenotypeClassificationPredicateTest	1.0	1.0	1.0	1.0	25	5	13	6
□ model.tests.GenotypeClassificationPredicateTest	1.0	0.0	0.67	1.0	9	1	3	2
□ model.tests.GenotypeClassificationPredicateTest	0.81	1.0	0.86	0.86	27	4	14	7
□ model.tests.GenotypeComparatorTest:testSort	0.8	0.5	0.8	0.67	25	4	10	6
□ model.tests.GenotypeTest:testCreate	1.0	1.0	1.0	1.0	9	2	4	1
□ model.tests.GenotypeTest:testHomozygous	0.76	1.0	1.0	0.67	17	2	4	3
□ model.tests.GenotypeTest:testRequireAtLeastTw	1.0	0.33	0.33	NaN	4	3	3	0
□ model.tests.GenotypeTest:testCreateWithFields	1.0	1.0	1.0	1.0	12	2	6	2

Checked in the case of redundant test

Test methods

Redundancy ratio

Number of covered items

Figure 5: Snapshot of the *TeReDetect* tool.

a set of test methods in this group and inspected them. We found few cases that seem as redundant tests which are in fact true-negative errors as reported in [12]. However, according to our inspection and code analysis, such test methods cover at least one coverable item not covered by any other test method. For instance, a test method named *testOneBin* in Allelogram covers a loop only once while some other test methods cover that loop more than one time. Therefore, loop redundancy of this method is slightly less than 1 (0.91) and thus detected as nonredundant by our redundancy metrics. For the same test method, the other types of redundancy considering only statement, branch, and condition coverage are 1. In fact, the above test cases contribute to loop coverage and we thus mark it as nonredundant since it covers a loop in a way (only once) not covered by other test methods.

Having a candidate set of redundant test methods (redundant tests based on *All* criterion: 69%), tester needs to decide about their order to inspect their source code. In this study, the first author (a graduate student of software testing) manually inspected the test methods. Recall the heuristics discussed in Section 5 about the sorting strategy of test method in the proposed process: test methods with fewer numbers of covered items have higher likelihood of being redundant. We thus decided to order the tests in the ascending order of the number of covered items (e.g., statement). In this case, we hoped to find redundant test methods sooner which may lead to a reduction in the search space (discussed next).

As the next step, manual inspection of a test was performed by comparing the source code of the test with other tests having high pair redundancy with the current one. The main focus of this step should be detecting redundancy by covering the shortcomings of coverage-based redundancy detection discussed in Section 5.

Redundancy of one test affects the redundancy of others. For instance, if test method A is redundant because it covers the same functionality covered by test method B (while there are no other tests to cover this functionality), test method B cannot be redundant at the same time. Therefore, while both of them are candidates for being redundant tests according to coverage information, but only one of them should be considered redundant finally. We refer to such effects as inter-test-method-redundancy effects

By only using redundancy information from the beginning step of the process, tester would need to keep track of all the tests previously detected as redundant during the process and apply the inter-test-method-redundancy effects by him/her self. However, recalculating the coverage information, after each redundancy detection, can reduce the search space (as explained next). Therefore, detecting redundant tests one by one and subsequently recalculating redundancy metrics increase precision and efficiency of the tester.

In this case study, we manually inspected the whole test suite of Allelogram. Figure 6 illustrates the whole process results by showing the size of five different test sets manipulated during the process. Those five test sets are discussed next.

We divide test methods into two categories: redundancy known and redundancy unknown. The test artifacts in the redundancy-unknown set are pending inspection to determine whether they are redundant or not (Set 1*)*. Redundancy-known set contains redundant (Set 2) and nonredundant test sets whose decisions have been finalized. Furthermore, the set of nonredundant tests inside redundancy-known category contains three different sets: those identified through inspection (Set 3), those identified without inspection (Set 4), and the ones that were identified by system as nonredundant after nonredundancy has been detected through inspection (Set 5).

At the beginning of the process, by calculating redundancy metrics based on coverage information, test methods

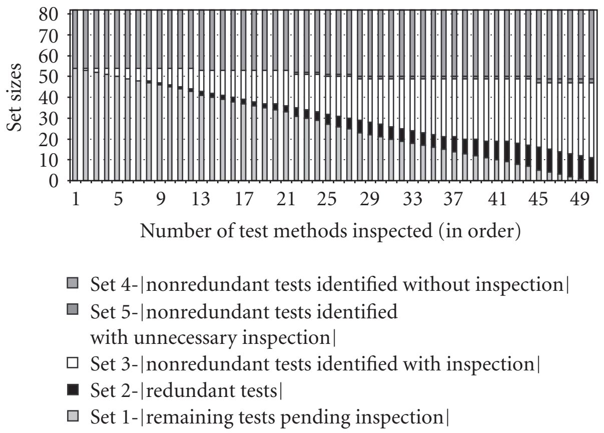


Figure 6: Labeling the test cases through the redundancy detection process.

are divided into two sets of *Nonredundant Tests without Inspection* and *Remaining Tests Pending Inspection* sets. As the figure shows, 28 test methods were recognized as nonredundant, while 54 (82 − 28) test methods needed to be inspected.

After each test method inspection, redundancy of that test is identified. This test method then leaves the *Remaining Tests Pending Inspection* set and Nonredundant test joins *Nonredundant Tests with Inspection* set while each redundant test joins *Redundant Tests* set. In the second case, redundancy metrics are recalculated.

In this case study, as shown in Figure 5, 11 test methods are recognized as redundant (test methods numbered in the x-axis as 7, 12, 19, 21, 24, 27, 36, 38, 40, 41, and 44). In these cases, new iterations of the process were performed by recalculating the redundancy metrics. In 5 cases (test methods numbered 12, 21, 24, 27, and 44), the recalculating led to search space reduction (5 test methods left the *Remaining Tests Pending Inspection* set and joined the *Nonredundant Tests without Inspection* set). In 2 of them (test methods 21 and 44), recalculating caused 2 test methods to leave *Nonredundant Tests with Inspection* set and join *Nonredundant Tests with Unnecessary Inspection* set.

At the beginning of the process, the size of the *Remaining Tests Pending Inspection* set was 54 (our initial search space). However, through the process, recalculating reduced the number of test methods that needed to be inspected to 49. In this case study, we ordered test methods in the ascending order of number of their covered items.

The final result of the process is a reduced test set containing 71 test methods instead of 82 (the original test suite of Allelogram). Stopping point of this process is considered by inspecting all the redundant candidate test methods (with SR = 1) and no cost estimation is applied for this purpose.

6.2. Evaluating the Proposed Process. To evaluate the preciseness of the proposed process, we considered the main purpose of test redundancy detection as discussed by many researchers. Test minimization should be performed in a way that the fault detection effectiveness of the test suite is preserved. Therefore, the process is successful if it does not reduce the fault detection capability.

One way to evaluate the above success factor of our test minimization approach is to inject probable faults in the SUT. Mutation is a technique that is widely used for this purpose ([25, 26]). The researches in [27, 28] show that the use of mutation operators is yielding trustworthy results and generated mutants can be used to predict the detection effectiveness of real faults.

In this work, we used the mutation analysis technique for the evaluation of the fault detection effectiveness of the reduced test suites generated by our technique. However, after completing this research project, we found out that, as another approach, we could also use the mutation analysis technique to detect test redundancy in a different alternative approach as follows. If the mutation scores of a given test suite with and without a particular test case are the same, then that test case is considered redundant. In other words, that test case does not kill (distinguish) any additional mutant. We plan to compare the above test redundancy detection approach with the one we conducted in this paper in a future work.

To inject simple faults into our case study, we used the MuClipse [29] tool which is a reincarnation of the MuJava [30] tool in the form of an Eclipse plug-in. Two main types of mutation operators are supported by MuClipse: method level (traditional) and class level (object oriented) [30].

To inject faults according to the traditional mutation operators, MuClipse replaces, inserts or deletes the primitive operators in the program. 15 different types of traditional mutation operators are available in MuClipse [29]. One example of this operators is the Arithmetic Operator Replacement (AOR) [31].

The strategy in object-oriented mutation operators is to handle all the possible syntactic changes for OO features by deleting, inserting, or changing the target syntactic element. 28 different types of OO mutation operators are available in MuClipse [29]. One example is Hiding variable deletion (IHD) which deletes a variable in a subclass that has the same name and type as a variable in the parent class [32].

All the available above mutation operators were used in this experiment. During this step, we found that MuClipse generates some mutants which failed to compile. These types of mutants are referred to as stillborn mutants which are syntactically incorrect and are killed by the compiler [29]. The total number of mutants for Allelogram that were not stillborn was 229.

To evaluate the fault detection effectiveness of the reduced test set by our proposed process compared to original test set, we calculated their mutation scores. We used MuClipse to execute all the created mutants with the two test sets (original and reduced). Table 5 shows the mutation score of three test sets: original test set, reduced test set only based on coverage information, and reduced test set through collaboration process with a tester.

The result shows that every mutant that is killed by original test set is killed by the reduced set (derived by

Table 5: Mutation score of three test suites for Allelogram.

Test set	Cardinality	Mutation score
Original	82	51%
Reduced (coverage based)	28	20%
Reduced (collaborative process)	71	51%

Table 6: Cost/benefit comparison.

	Cost	Benefit
Full automation	Low	Imprecise reduced set
Full manual	High	Precise reduced set
Semiautomated	Mid	Precise reduced set

the collaborative process) as well. In other words, the effectiveness of these two test sets is equal while the reduced set (solely based on coverage information) has 11 (82 − 71) less tests than the first one. That test suite thus has lower fault detection effectiveness.

Mutation score decreasing from 51% in original test set to 20% in the reduced set only based on coverage information confirms our discussion in Section 3 about impreciseness of test redundancy detection based only on coverage information.

7. Discussion

7.1. Effectiveness and Precision. Let us recall the main purpose of reducing the number of test cases in a test suite (Section 1): decreasing the cost of software maintenance. Thus, if the proposed methodology turns to be very time consuming, then it will not be worthwhile to be applied.

Although the best way to increase the efficiency of the process is to automate all required tasks, at this step we suppose that it is not practical to automate all of them. Thus, as we discuss next, human knowledge is currently needed in this process.

To perform manual inspection on test suite with the purpose of finding redundancy, testers need to spend time and effort on each test source code and compare them together. To decrease the amount of required effort, we have devised the proposed approach in a way to reduce the number of tests needed to be inspected (by using the suite redundancy metric). Our process also suggests useful information such as pair redundancy metric to help testers find other proper tests to compare with the test under inspection.

We believe that by using the above information, the efficiency of test redundancy detection has been improved. This improvement was seen on our case study while we first spent on average more than 15 minutes for each test method of Allelogram test suite before having our process. But inspecting them using the proposed process took on average less than 5 minutes per test method (the reason of time reduction is that in the later we knew other proper test methods to compare them with the current test). Since only one human subject (tester) performed the above two approaches, different parts of the Allelogram test suite were analyzed in each approach to avoid bias (due to learning and gaining familiarity) on time measurement.

However the above results are based on our preliminary experiment and it is thus inadequate to provide a general picture about the efficiency of the process. For a more systematic analysis in that direction, both time and effort should be measured more precisely with more than one subject on more than one object. Such an experiment is considered as a future work.

In addition to the efficiency of the process, precision of redundancy detection was also evaluated in our work. As explained in Section 6.2, this evaluation has been done in our case study by applying mutation technique. The result of analysis on one SUT confirmed the high precision of the process.

However, human's error is inevitable in collaborative processes which can affect the precision of the whole process. To decrease this type of error, the tester needs to be familiar with the written tests. Therefore, we suggest having the original test suite developers involved in the redundancy detection process if possible or that they be at least available for the possible questions during the process. In other words, a precise teamwork communication is required to detect correct test redundancy.

7.2. Cost/Benefit Analysis. According to above discussions, our redundancy detection technique has the following benefits.

(i) Reducing the size of test suite by keeping the fault detection effectiveness of that.

(ii) Preventing possible future integrity issues in the test suite.

(iii) Reducing test maintenance costs.

Different types of required costs in this process are summarized as follows.

(i) TeReDetect installation costs.

(ii) System execution time during the process (steps 1, 4, and 5 in Figure 4).

(iii) Redundancy analysis by human testers (steps 2 and 3 in Figure 4).

The first and second cost items are not considerable while the main part of the cost is about the third one which contains human efforts.

Table 6 shows an informal comparison of above costs and benefits in three approaches of full automation, full manual, and semiautomated process proposed in this paper. In the second and third approaches that human has a role, it is inevitable that the preciseness of human affects the benefits of the results.

7.3. Scalability. In large-scale systems with many LOC and test cases, it is not usually feasible to look at and analyze the test cases for the entire system. However, as mentioned before, in TeReDetect it is possible to select a subset of test suite and also a subset of SUT. This functionality of TeReDetect increases the scalability of this tool to a great extent by making it possible to divide the process of redundancy detection into separate parts and assign each part to a tester. However a precise teamwork communication is required to make the whole process successful.

Flexible stopping point of the proposed process is another reason for its scalability. According to the tester's discretion, the process of redundancy detection may stop after analyzing the subset of test cases or continue for all existing tests. For instance, in huge systems, by considering the cost of redundancy detection, project manager may decide to analyze only the critical part of the system.

7.4. Threats to Validity

7.4.1. External Validity. Two issues limit the generalization of our results. The first one is the subject representativeness of our case study. In this paper the process has been done by the first author (a graduate student). More than one subject should be experimented in this process to be able to compare their results to each other. Also, this subject knew the exact objective of the study which is a threat to the result. The second issue is the object program representativeness. We have performed the process and evaluate the result on one SUT (Allelogram). More objects should be used in experiments to improve the result. Also our SUT is a random project chosen from the open source community. Other industrial programs with different characteristics may have different test redundancy behavior.

7.4.2. Internal Validity. The result about efficiency and precision of the proposed process might be from some other factors which we had no control or had not measured. For instance, the bias and knowledge of the tester while trying to find redundancy can be such a factor.

8. Conclusion and Future Works

Measuring and removing test redundancy can prevent the integrity issues of test suites and decrease the cost of test maintenance. Previous works on test set minimization believed that coverage information is useful resource to detect redundancy.

To evaluate the above idea we performed an experiment in [12]. The result shows that coverage information is not enough knowledge for detecting redundancy according to fault detection effectiveness. However, this information is a very useful starting point for further manual inspection by human testers.

Root-cause analysis of above observation in [12] has helped us to improve the precision of redundancy detection by covering the shortcomings in the process proposed in this paper.

We proposed a collaborative process between human testers and redundancy system based on coverage information. We also performed an experiment with that process on a real java project. This in turn led us to find out that the sharing the knowledge between the human user and the system can be useful for the purpose of test redundancy detection. We conclude that test redundancy detection can be performed more effectively when it is done in an interactive process.

The result of the case study performed in this paper shows that fault detection effectiveness of the reduced set is the same as the original test set while the cost of test maintenance for reduced one is less than the other (since the size of the first set is less than the second one).

The efficiency of this process in terms of time and effort is improved comparing to the case of manual inspection for finding test redundancy without this proposed process.

In this paper, the efficiency factor was discussed qualitatively. Therefore measuring precise time and efforts spent in this process is considered as a future experiment.

Finding the stopping point of the process needs maintenance and effort cost estimation which is not studied thoroughly in this work and is also considered as a future work.

As explained in Section 5, the order of the tests inspected in the proposed process can play an important role in the test reduction result. In this work we suggested a few strategies with their benefits to order the test while this needs to be studied more precisely. Also, test sequential constraints such as the case of dependent test cases are not discussed in this work.

Visualization of coverage and redundancy information can also improve the efficiency of this process extensively. We are now in the process of developing such a visualization technique to further help human testers in test redundancy detect processes.

In addition to above, some tasks which are now done manually in this proposed process could be automated in future works. One example is the automated detection of redundancy in the verification phase of JUnit test methods which will most probably require the development of sophisticated code analysis tools to compare the verification phase of two test methods.

Acknowledgments

The authors were supported by the Discovery Grant no. 341511-07 from the Natural Sciences and Engineering Research Council of Canada (NSERC). V. Garousi was further supported by the Alberta Ingenuity New Faculty Award no. 200600673.

References

[1] S. G. Eick, T. L. Graves, A. F. Karr, U. S. Marron, and A. Mockus, "Does code decay? Assessing the evidence from change management data," *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 1–12, 2001.

[2] D. L. Parnas, "Software aging," in *Proceedings of the International Conference on Software Engineering (ICSE '94)*, pp. 279–287, Sorrento, Italy, May 1994.

[3] B. V. Rompaey, B. D. Bois, and S. Demeyer, "Improving test code reviews with metrics: a pilot study," Tech. Rep., Lab

on Reverse Engineering, University of Antwerp, Antwerp, Belgium, 2006.

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

[5] A. Deursen, L. Moonen, A. Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP '01)*, Sardinia, Italy, May 2001.

[6] 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," in *Proceedings of the Conference on Software Maintenance (ICSM '98)*, pp. 34–43, Bethesda, Md, USA, November 1998.

[7] M. J. Harrold, R. Gupta, and M. L. Soffa, "Methodology for controlling the size of a test suite," *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 3, pp. 270–285, 1993.

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

[9] A. J. Offutt, J. Pan, and J. M. Voas, "Procedures for reducing the size of coverage-based test sets," in *Proceedings of the 11th International Conference on Testing Computer Software (ICTCS '95)*, pp. 111–123, Washington, DC, USA, June 1995.

[10] W. E. Wong, J. R. Morgan, 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.

[11] W. E. Wong, J. R. Horgan, A. P. Mathur, and Pasquini, "Test set size minimization and fault detection effectiveness: a case study in a space application," in *Proceedings of the IEEE Computer Society's International Computer Software and Applications Conference (COMPSAC '97)*, pp. 522–528, Washington, DC, USA, August 1997.

[12] N. Koochakzadeh, V. Garousi, and F. Maurer, "Test redundancy measurement based on coverage information: evaluations and lessons learned," in *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation (ICST '09)*, pp. 220–229, Denver, Colo, USA, April 2009.

[13] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, Calif, USA, 1990.

[14] A. Ngo-The and G. Ruhe, "A systematic approach for solving the wicked problem of software release planning," *Soft Computing*, vol. 12, no. 1, pp. 95–108, 2008.

[15] R. Milner, "Turing, computing, and communication," in *Interactive Computation: The New Paradigm*, pp. 1–8, Springer, Berlin, Germany, 2006.

[16] F. Arbab, "Computing and Interaction," in *Interactive Computation: The New Paradigm*, pp. 9–24, Springer, Berlin, Germany, 2006.

[17] M. Takaai, H. Takeda, and T. Nishida, "A designer support environment for cooperative design," *Systems and Computers in Japan*, vol. 30, no. 8, pp. 32–39, 1999.

[18] Parasoft Corporation, "Parasoft Jtest," October 2009, http://www.parasoft.com/jsp/products/home.jsp?product=Jtest.

[19] IBM Rational Corporation, "Rational manual tester," January 2009, http://www-01.ibm.com/software/awdtools/tester/manual/.

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

[21] nitinpatil, "JFeature," June 2009, https://jfeature.dev.java.net/.

[22] 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 Software Engineering*, vol. 33, no. 12, pp. 800–816, 2007.

[23] L. C. Briand and J. Wüst, "Modeling development effort in object-oriented systems using design properties," *IEEE Transactions on Software Engineering*, vol. 27, no. 11, pp. 963–986, 2001.

[24] C. Manaster, "Allelogram," August 2008, http://code.google.com/p/allelogram/.

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

[26] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Transactions on Software Engineering*, vol. 3, no. 4, pp. 279–290, 1977.

[27] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608–624, 2006.

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

[29] B. Smith and L. Williams, "MuClipse," December 2008, http://muclipse.sourceforge.net/.

[30] J. Offutt, Y. S. Ma, and Y. R. Kwon, "MuJava," December 2008, http://cs.gmu.edu/∼offutt/mujava/.

[31] Y. S. Ma and J. Offutt, "Description of method-level mutation operators for java," December 2005, http://cs.gmu.edu/∼offutt/mujava/mutopsMethod.pdf.

[32] Y. S. Ma and J. Offutt, "Description of class mutation mutation operators for java," December 2005, http://cs.gmu.edu/∼offutt/mujava/mutopsClass.pdf.