The Effectiveness of Regression Testing Techniques in Reducing the Occurrence of Residual Defects

Panduka Nagahawatte* and Hyunsook Do*
*Department of Computer Science
North Dakota State University
Fargo, ND
pnagahawatte@redriv.com, hyunsook.do@ndsu.edu

***Abstract*—Regression testing is a necessary maintenance activity that can ensure high quality of the modified software system, and a great deal of research on regression testing has been performed. Most of the studies performed to date, however, have evaluated regression testing techniques under the limited context, such as a short-term assessment, which do not fully account for system evolution or industrial circumstances. One important issue associated with a system lifetime view that we have overlooked in past years is the effects of residual defects – defects that persist undetected – across several releases of a system. Depending on an organization's business goals and the type of system being built, residual defects might affect the level of success of the software products. In this paper, we conducted an empirical study to investigate whether regression testing techniques are effective in reducing the occurrence and persistence of residual defects across a system's lifetime, in particular, considering test case prioritization techniques. Our results show that heuristics can be effective in reducing both the occurrence of residual defects and their age. Our results also indicate that residual defects and their age have a strong impact on the cost-benefits of test case prioritization techniques.**

***Keywords*-Regression testing, test case prioritization, residual defects, empirical study**

I. Introduction

Regression testing is a necessary maintenance activity that can promote higher quality in modified software systems. While the importance of regression testing has been well recognized by practitioners and researchers, in practice, it has often been inadequately applied because it involves a great deal of time and effort.

For this reason, many researchers have proposed various techniques for improving the cost-effectiveness of regression testing (e.g., [1], [2], [3], [4], [5]) and empirically studied these techniques (e.g., [6], [7], [8], [9]). Although these empirical studies have provided valuable insights on regression testing techniques, the majority of them have evaluated regression testing techniques focusing on a short-term assessment (a one-time process), which does not fully account for system evolution or industrial circumstances. Fenton et al. [10], however, found that long-term (system lifetime) views could lead to conclusions very different from short-term views.

One important issue associated with a system lifetime view that has been overlooked in prior research in regression testing is the effect of *residual defects* – defects that persist undetected across *several* releases of a system before causing failures. The occurrence of residual defects could damage the reputation of organizations, and could cause some loss of revenue. However, researchers have studied residual defects only focusing on how the number of residual defects can be estimated and how the estimated residual defects can be used in judging the quality and reliability of the software [11], [12]. To date, no empirical studies have investigated regression testing problems related to *residual defects*. Rather, they have focused on faults that tied to one particular version of a system, and evaluated regression testing techniques using these faults.

We believe that the presence of these defects can also influence the cost-effectiveness of regression testing over system lifetimes. Residual defects are common in software [13], and several factors contribute to causing them. Inadequate test suites can let defects slip through testing. Regression testing techniques that discard tests (non-safe test selection or test reduction) can omit tests that would otherwise have detected faults. The amount of testing effort can affect the occurrence of residual defects. One study by Evanco [14] shows that the number of residual defects in a system could be responsible for about 22% of the total number of defects based on a residual defect estimation model. Companies often release their software products with known defects because they decide that known defects are not serious and spending time fixing those defects is not worthwhile. Also, companies often have time-to-market pressures because delayed product release increases the chance of losing customers to competing companies. However, an organization's decision to release a product early can cause residual defects to accumulate further.

Depending on an organization's business goals (e.g., achieving early market share versus building long-term stability) and the type of system being built (e.g., safety critical or not), residual defects might significantly affect the level of success of the software products and the reputation of organizations [15]. For example, if residual defects found in a later release critically degrade the system or even cause

more serious problems, such as financial losses or leakage of customers' sensitive information, these incidents may lead to the loss of customers. Moreover, the cost of correcting faults tends to increase when they persist across multiple maintenance cycles.

Therefore, we have performed a controlled experiment to investigate whether regression testing techniques can reduce the effects of residual defects, particularly focusing on test case prioritization techniques. We hypothesized that test case prioritization can be effective in reducing both the occurrence of residual defects across system lifetimes and the age of residual defects (the number of subsequent releases they persist through undetected).

The results of our experiment show that heuristics can be effective in reducing both the occurrence of residual defects and their age. Our results also indicate that residual defects and their age have a strong impact on the cost-benefits of test case prioritization techniques.

In the next section of this paper, we describe background information and related work relevant to prioritization techniques and residual defects. Section III presents our experiment setup, and Section IV presents results and analysis. Section V discusses our results, and Section VI presents conclusions and future work.

II. Background and Related Work

A. Test Case Prioritization

Test case prioritization techniques [4], [5], [7], [8], [16] schedule test cases in an execution order according to some criterion. The goal of prioritization is to increase the likelihood that if the test cases are used for regression testing in the given order, they will more closely meet some objective than they would if they were executed in some other order. For example, testers might wish to schedule test cases in an order that achieves code coverage at the fastest rate possible, exercises features in order of expected frequency of use, or increases the likelihood of detecting faults early in testing.

Depending on the types of information available relating to tests, various test case prioritization techniques can be utilized. One type of information concerns coverage of code elements, and techniques can be distinguished in terms of the type of code elements they consider. Thus, test cases can be prioritized in terms of the number of code statements, basic blocks, or methods they executed on a previous version. For example, one technique, *total block coverage prioritization*, simply sorts the test cases in the order of the number of blocks they cover and, if multiple test cases cover the same number of blocks, orders these randomly.

A second type of information by which techniques can be distinguished involves the use of "feedback." When prioritizing test cases, having selected a particular test case as "next best", information about that test case can be used to re-evaluate the worth of test cases not yet chosen, prior to picking the next best test case. For example, *additional block coverage prioritization* iteratively selects a test case that yields the greatest block coverage, then adjusts the coverage information for the remaining test cases to indicate their coverage of blocks not yet covered, and then repeats this process until all blocks coverable by at least one test case have been covered. At this point, the process is repeated on remaining test cases.

B. Previous Empirical Work

More recent research on regression testing has employed empirical studies focusing on comparisons of techniques [7], [9], [17], [18]. Some of these studies have evaluated the cost-benefits tradeoffs among techniques using explicit cost models [6], [19], [20]. Studies such as these have allowed researchers to begin to compare techniques in terms of costs and benefits relative to actual software systems.

Many researchers have studied various approaches to estimating fault-proneness of systems [21], [22], but they have not considered residual defects. To date, only few studies have considered residual defects and their effects on testing across system lifetimes, and issues related to time constraints during the regression testing. Malaiya and Denton [12] consider the number of residual defects as a metric in evaluating software quality. Their approach uses test coverage information to estimate how many defects have been detected and how many are undetected.

In [17], Kim et al. consider an evaluation approach that treats testing as a "continuous process", and report results of empirical studies in which test application frequency is varied, as could happen across multiple releases of a software system. They find that the effectiveness of regression testing techniques changes as the frequency of regression testing changes. Whereas the study reported in [17] does not provide an explicit evaluation model, in a later paper Kim et al. [8] do present regression testing models that consider software evolution and report results of empirical studies considering techniques that utilize historical test case performance data.

Walcott et al. [5] present a technique that combines information on test execution times with code coverage information, and utilizes levels of time constraints while investigating prioritization effectiveness. Do et al. [6] have studied the effects of time constraints imposed on regression testing in an empirical study. Although they do not consider residual defects in their study, they have considered costs related to faults missed in regression testing.

In this work, we focus specifically on the effects of residual defects across system lifetimes considering test case prioritization.

C. Economic Model for Regression Testing

Most of the empirical studies on regression testing techniques performed to date have used economic models that omit important context factors and render some types of comparisons between techniques impossible. This can cause

empirical studies to improperly assess the costs and benefits of regression testing techniques in practical settings. To address this problem, we have developed an economic model, EVOMO (EVOlution-aware economic MOdel for regression testing) [23], [24], which captures costs and benefits of regression testing methodologies relative to particular regression testing processes, considering methodologies in light of their business value to organizations, in terms of the cost of applying the methodologies and how much revenue they help organizations obtain.

EVOMO[1] involves two equations: one that captures costs related to the salaries of the engineers who perform regression testing (to translate time spent into monetary values), and one that captures revenue gains or losses related to changes in system release time (to translate time-to-release into monetary values). Significantly, the model accounts for costs and benefits across entire system lifetimes, rather than on snapshots (i.e. single releases) of those systems, through equations that calculate costs and benefits *across entire sequences of system releases*. The model also accounts for the use of incremental analysis techniques (e.g., reliance on previously collected data where possible rather than on complete recomputation of data), an improvement also facilitated by the consideration of system lifetimes.

In this work, we extend EVOMO to incorporate the costs associated with residual defects particularly the costs for fixing residual defects considering the age of the residual defects.

III. Empirical Study

We wish to address the following research questions:

RQ1: Can test case prioritization be effective in reducing the occurrence of residual defects and their age?

RQ2: How do residual defects affect the cost-benefits of prioritization techniques?

To address our research questions, we designed and performed a controlled experiment. The following subsections describe the objects of analysis, independent variables, dependent variables and measures, experiment design, threats to validity, and data analysis.

A. Objects of Analysis

We used nine versions of the *ant* program with JUnit test cases which came with the program, obtained from the Software-artifact Infrastructure Repository (SIR) [25], as the object of study. *Ant* is an open source make utility that extends Java capabilities. It provides Java developers with a make utility that is based on XML configuration and is different from conventional shell based make utilities.

Each version's metrics are listed in Table I. Version one (*v1*), which is a base version of *ant*, is not listed in the table because regression testing starts from the second release of the system. We, however, used information from *v1* to obtain mutants for *v2*. We used RSM1 (Resource Standard Metrics), a source code metrics tool to obtain the LOC and the number of Java class files.[2] "Tests" shows the number of test cases available for each version.

Table I
METRICS FOR EACH VERSION OF *ant*

Version	Tests	LOC	Class files	Mutants
v2	112	41715	473	54
v3	137	55582	553	30
v4	219	55655	553	21
v5	215	72422	718	21
v6	520	74507	729	214
v7	557	74598	730	53
v8	559	97194	905	42
v9	877	97267	906	17

Our experiment requires object programs that contain faults, so we utilized mutation faults provided with the program [26]. Because our focus is regression testing and detection of regression faults (faults created by code modifications), we considered only mutation faults located in modified methods. The "Mutants" column shows the number of mutants used for each version. The mutation tool generated a large number of mutants for each version, but we listed only the number of mutants that are revealed by our test cases.

B. Variables and Measures

1) Independent variable: Our empirical study manipulated one independent variable, prioritization technique. We consider one *control* technique and two *heuristic* prioritization techniques.

The control technique, the original test case order (*To*), serves as experimental control. (*To*) utilizes the order in which test cases are executed in the original testing scripts provided with the object program, and thus, serves as one potential representative of "current practice".

We utilized two techniques as heuristic techniques: total coverage prioritization (*Ttot*) and additional coverage prioritization (*Tadd*). *Ttot* orders test cases based on the total code coverage they achieve. The level of granularity for instrumentation is that of basic blocks. *Tadd* is a variation of *Ttot* technique in which feedback mechanism is employed.

As noted earlier, one of the main causes of residual defects comes from a lack of testing performed [14], [27]. To simulate this situation, for each of the foregoing techniques, we foreshorten the test execution process by 50%.[3] In other words, we simulate the effects of having the testing process halted half way through. For *ant*, test execution time varies only slightly, so we implemented the amount of testing efforts by reducing the number of test cases by 50%.

[1]Due to space limitations, here we just summarize each equation briefly. See [23], [24] for detailed descriptions.

[2]http://msquaredtechnologies.com/m2rsm/index.htm

[3]As a pilot study, we chose 50% as the amount of testing efforts, but we can easily extend this work considering different amount of testing efforts.

2) Dependent Variables and Measures:

The number of residual defects and their age values

For RQ1, our dependent variables are the number of residual defects and their age values. For each version of the program, we measured the number of residual defects that slip through regression testing from previous versions. For example, as shown in Table II, suppose we have 10 defects in *v1* (*v0* is a base version), we detected 6 of them while regression testing *v1*, and 4 of them were left undetected and propagated into *v2*. Suppose *v2* also has 15 defects unique to it, we applied the same procedure to *v2*, and we have 9 of 15 detected and 6 of 15 undetected. Suppose 1 of 4 propagated defects from *v1* detected and 3 of 4 undetected. Under this scenario, our number of residual defects are 4 residual defects for *v1*, and 9 for *v2* (6+3). Since we define the age value as "the number of subsequent releases they persist through undetected", under this scenario, we assign an age value of "0" for 4 residual defects in *v1*, an age value of "1" for 3 propagated from *v1* to *v2*, and an age value of "0" for 6 residual defects in *v2*.

Table II
DEFECTS AND THEIR AGE

	v1	*v2*	
	Unique defects for *v1*	Unique defects for *v2*	Propagated defects from *v1*
No. of defects	10	15	4
No. of undetected	6	9	1
No. of detected	4	6	3
Age of defects	0	0	1

Relative cost-benefit values

The fact that a test case prioritization technique reduces the number of residual defects and their age value does not guarantee that the technique will be cost-effective. That is, even if the number of residual defects and their age value are reduced, if this reduction does not lead to savings in testing, then we cannot gain benefits by applying techniques. Thus, in RQ2, we further evaluate cost-effectiveness of techniques. For RQ2, our dependent variable is a relative cost-benefit value produced by applying EVOMO economic model presented in Section II, using a further calculation described below (Equation 1). The cost and benefit components are measured in dollars. To determine the *relative cost-benefit* of prioritization technique *T* with respect to baseline technique *base*, we use the following equation:

$$(\text{Benefit}_T - \text{Cost}_T) - (\text{Benefit}_{base} - \text{Cost}_{base}) \quad (1)$$

When this equation is applied, positive values indicate that T is beneficial compared to $base$, and negative values indicate otherwise. We used the original technique as a baseline in this experiment.

The major cost components that EVOMO captures are as follows: costs for applying regression testing techniques, costs associated with missed faults, costs for artifact analysis, and costs of delayed fault detection feedback. In addition to these cost components, we are also required to obtain the age of residual defects to investigate our research questions. To do this, we recorded all residual defects at the end of regression testing for each version. Using this information we calculated the cost of fixing residual defects for all versions considering the age value for each residual defect.

Determining the cost of fixing residual defects, however, is much more difficult. Given the many factors that can contribute to these costs, and the long-term nature of these costs, we could not obtain this measure directly. Instead, we estimated them based on data obtained from a survey by Shull et al. [28]. The survey suggests that finding and correcting severe defects after product release is 100 times more expensive than correcting them before product release. The survey also suggests that the effort to find and fixing non-severe defects is about twice as expensive as finding these defects before delivery. These values are also situation dependent. Because our object program is medium size, and we do not wish to inflate the effect of residual defects in the results, we applied the multiplier for non-severe defects to estimate the cost of fixing residual defects.

C. Experiment Setup

To investigate our research questions, we required object programs that contain faults. Thus, we utilize artifacts equipped with mutation faults, which have been created by *ByteME* (Bytecode Mutation Engine), from the SIR repository [26].

However, to utilize such mutants for our experiment, we need to support two things. First, we must add support, to our experimentation tools, for the activity of "carrying faults forward", so that faults not detected by techniques can be placed as residual defects into subsequent versions.

Second, we need to regroup mutants as residual defects propagated into subsequent versions. Table I lists the numbers of mutants for our object program, but in actual testing scenarios, programs do not typically contain as many faults as these numbers of mutants. Thus, to simulate more realistic scenarios, [23] introduced *mutant groups*, which were formed by randomly selecting mutants from the pools of mutants created for each version; each mutant group size varied (randomly) between 1 and 10. In this study, we also performed our experiment considering the mutant groups. For version 1, we used the same mutant groups as those used in [23]. For the remaining of versions, however, we needed to regroup mutants to include residual defects that have not been detected during regression testing for previous versions.

In this section, we describe the experimental setup that accommodates such needs. The following steps describe the overview of our experimental procedure.

1) **Execute tests:** For all mutant groups, we executed the half of reordered test cases obtained by applying test case prioritization techniques, and collect fault-matrices that list which tests detect which faults.
2) **Identify residual defects:** Using information we gathered from step 1, we identified the mutants that were not detected by the test cases. We refer them as "residual defects." We also recorded mutant numbers, the originating version, mutation details, and the age of the mutants that were not detected.
3) **Propagate residual defects:** To propagate residual defects that are not detected from previous versions to the subsequent versions as shown in Figure 1, we used the procedure for carrying residual defects forward described in next subsection. As Figure 1 shows, residual defects that were left undetected in *v1* can be propagated into *v2* and *v3* as well. Next subsection will provide more details.

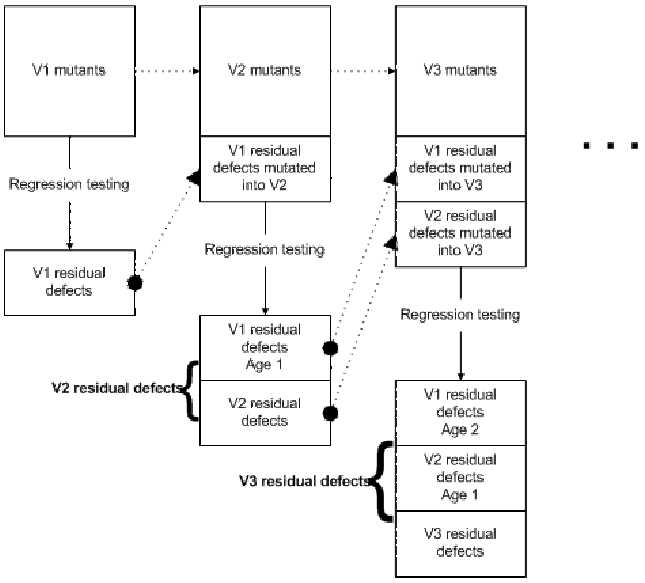

Figure 1. Transferring mutants across versions

4) **Build new mutant groups:** Propagation of residual defects from prior versions led to the creation of new mutant groups in subsequent versions. We build new mutant groups by following the same procedure described in [23] (we also provided a brief description of this procedure in the beginning of this section). Each of these subsequent versions could contain mutants from its own version and any prior versions as residual defects.
5) **Repeat steps 1-4:** Steps 1-4 are repeated across all versions.

After completing the procedure we described, we obtained 30 *sequences of mutant groups* by randomly selecting a mutant group for each version of the program. Then we collected all data values described in Section III-B2 for each of the 30 sequences of mutant groups. These resulting data sets, residual defect numbers, age values, and relative cost-benefit values, serve as the data for our subsequent analysis.

Procedure for Carrying Faults Forward

The procedure for forwarding residual defects involves two tasks: (1) Identify faults that are not detected by test case prioritization techniques for a certain version (V_m). (We define these undetected faults as "residual defects"); (2) Replicate residual defects in a later version (V_n). These two tasks, repeatedly applied across all versions, could produce persistent residual defects that were undetected from an early version of the program and last until the final version of the program.

Since mutants we utilized in this study were generated in Java bytecode, we need to manipulate mutants in bytecode to propagate them into the subsequent versions, which is a more complicated procedure than simply handling source level of mutants or faults. To illustrate this procedure, we consider two versions of the *ant* program, V_m (originating version) and V_n (destination version). Suppose that V_m has a set of mutants and some of them are forwarded into V_n. Figure 2 visually illustrates how mutants are propagated from one version to another, and the following steps provide detailed description of this procedure:

1) **Identify residual defects:** To identify and propagate mutants, we are required to record and track details of the individual mutants, and *ByteME* allows us to collect all the details about the mutants, such as the mutation operator applied, the mutant identifier, the class/method name, and the byte code location where mutation applied. For V_m, we record all the information including the version number.
We then execute test cases over V_m to identify which mutants were undetected by test cases (note that we are using 50% of test cases to simulate testing scenario with limited testing time). This step provided us with the undetected mutants that need to be replicated in the next version, V_n. For example, in Figure 2, we identified two mutants that were not detected by test cases in V_m, and now they became residual defect candidates for V_n.
2) **Replicate residual defects:** Next, we need to replicate mutants that were undetected during regression testing for V_m in V_n. Since V_n has its own set of mutants (it is hidden in Figure 2, we assume the gray box contains a set of mutants for V_n), we first check whether mutants for V_n contain mutants identical to the residual defects that came from V_m. If the residual defect is identical to one of mutants for V_n, then we do not replicate the residual defect into V_n, however, we do consider it as the propagated defect from V_m. Otherwise, using *ByteME*, we create new mutants based on information that residual defects carry.

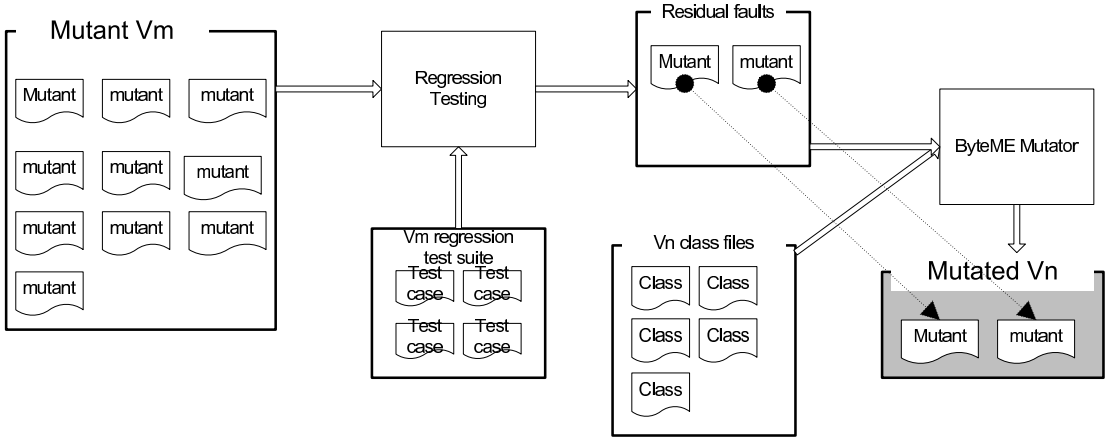

Figure 2. Mutation propagation procedure

Suppose we identified a mutant candidate for residual defect replication for the next version. Based on the mutant information provided by *ByteME*, as shown below, such as mutation operator (in this example, AOC – Argument Operator Change), class/method name to be mutated (ProjectHelper$RootHandler/resolveEntity), and the bytecode offset where mutation will be taken place (in this example, 127).

```
6:AOC:org.apache.tools.ant.ProjectHelper\$RootHandler:
resolveEntity:(Ljava/lang/String;Ljava/lang/String;)
Lorg/xml/sax/InputSource; :127:18:invoke virtual:0,1
```

The following shows a snippet of bytecode for the class file to be mutated, which has two string arguments (bytecode offsets 127 and 130). Note that 127 is the bytecode offset where actual mutation will be made.

```
% Original class-------------------------------------
125:  iload             %4
127:  invokevirtual     java.lang.String.substring
(II)Ljava/lang/String; (15)
130:  invokevirtual     java.lang.StringBuffer.append
(Ljava/lang/String;)Ljava/lang/StringBuffer; (8)
133:  ldc               "#" (17)
135:  invokevirtual     java.lang.StringBuffer.append
(Ljava/lang/String;)Ljava/lang/StringBuffer; (8)
138:  aload_3
139:  iload             %4
-----------------------------------------------------
```

Now we create a mutant by running *ByteME* with the information we obtained. The following shows a snippet of the mutated class file. Since the mutation operator, AOC, changes arguments order, we can see that "swap" instruction has been inserted in the bytecode offset 127.

```
% Mutated class--------------------------------------
125:  iload             %4
127:  swap
128:  invokevirtual     java.lang.String.substring
(II)Ljava/lang/String; (15)
131:  invokevirtual     java.lang.StringBuffer.append
(Ljava/lang/String;)Ljava/lang/StringBuffer; (8)
134:  ldc               "#" (17)
136:  invokevirtual     java.lang.StringBuffer.append
(Ljava/lang/String;)Ljava/lang/StringBuffer; (8)
139:  aload_3
140:  iload             %4
-----------------------------------------------------
```

Note that there are some cases that we cannot replicate: 1) V_n does not include the class file that the mutants associated with; 2) there are drastic changes between two class files from V_m and V_n.

Now, we need to decide whether these propagated mutants are useful for our experiment. Because experimentation with test suites only considers mutants that can be exposed by those suites, we compared the output of V_n run with the tests from our test suites, with and without each mutant present. If the outputs were equivalent for a particular mutant we discarded that mutant. We also discarded mutants that caused "verify" errors during execution, because these represent syntactic errors that would not require testing to be revealed.

3) **Repeat steps 1 and 2:** We repeat the first two steps across all versions considering one originating version at a time. Some destination versions could have mutants propagated from multiple versions. We were able to successfully replicate those mutants by considering one originating version at a time.

Figure 1 shows an example which illustrates how later versions could have residual defects from multiple previous versions: after completing regression testing for *v1*, some faults escaped into *v2* (the lower box in the left column); then, these residual defects are placed in *v2* as a part of *v2* mutants (the upper box in the middle column), and again, we have faults that are escaped after regression testing (the lower box in the middle column). Now we have residual defects from two different sources: one set from *v1*, which we assign "1" as an age value and one set from *v2*. After applying the same process to *v3*, we could have residual defects from multiple sources (thus, having different age values) as shown in the lower box in the last column.

D. Threats to Validity

Internal Validity: The conclusions about the cost benefit values of a regression test technique could have been

affected by the following two factors: errors in the tools we used and the estimates of the values we made to calculate the cost benefit. To address the first threat we used *ByteME* and the *Sofya* Java analysis tool, which have been created for the purpose of experimentation and has been thoroughly tested in previous experiments [23], [26], [29].

The second threat pertains to the values we used in calculating costs and benefits. We estimated the impact of residual defects by assigning weights according to age. The weights were estimated based on prior studies and surveys. However, this value depends on the software development project and the situation. We were careful in assigning weights, to prevent inflating the costs of them.

External Validity: The set of ant programs we used in the experiment are relatively small - average LOC for a version is 71K. Also, the time for regression test execution was small. Industrial software can be comparatively larger and more complex, changing the dynamics of regression testing. These changes from experimental setup to the industry might lead to differences in cost benefit tradeoffs. Since these factors are unknown in the experimental setup, studies of industrial software will have to be conducted to find those factors and relate them to the cost benefit calculation.

Residual defects are characterized in the general scientific literature as the hardest to find faults, and the defects that reside in the least exercised code. This indicates that residual defects are a special type of software defects and may possess characteristics unique to them. Our faults are derived through code mutation, and although there is some evidence that mutation faults can be representative of real faults for purposes of experimental evaluation of the effectiveness of testing techniques [26], [30], further studies are needed to investigate whether mutated faults can be representative of residual faults. We addressed this threat by selecting a variety of mutation operators to seed faults.

We have considered only one program in this experiment. The results obtained by this experiment might be skewed by program specific characteristics. Further studies using more programs will yield results that could be generalized with more confidence than just the results from this experiment considering ant program only.

IV. Data and Analysis

In this section, we present the results of our study, and data analyses, for each of our research questions in turn. (We discuss further implications of the data and results in Section V.)

For our statistical analysis, we followed a process well established in prior studies of test case prioritization (e.g., [26], [7], [4]): we used the Kruskal-Wallis non-parametric one-way analysis of variance followed by Bonferroni's test for multiple comparisons [31]. We used the Kruskal-Wallis test because our data did not meet the assumptions for using ANOVA: our data sets do not have equal variance, and some have severe outliers. For multiple comparisons, we used the Bonferroni method for its conservatism and generality. We used the SPSS statistics package to perform the analysis.

A. The effectiveness of prioritization in reducing the occurrence of residual defects and their age (RQ1)

Table III shows the number of residual defects undetected by each prioritization technique for all versions. From this table, we observed that there are three clear patterns in the data when we applied the original technique: for the first two versions (*v2* and *v3*), a relatively large number of residual defects has been undetected from previous versions; for the next three versions (*v4*, *v5*, and *v6*), the number of residual defects undetected decreased rapidly; for the last three versions (*v7*, *v8*, and *v9*), only one residual defect was undetected.

In the case of heuristics, they do not show a particular trend as we observed from the original technique, but overall, they produced relatively small numbers of residual defects across all versions. This indicates that heuristics are effective in reducing the occurrence of residual defects over the system lifetime. In particular, in the case of the feedback technique, only 2 residual defects (version 6) were undetected across all versions.

Table III
THE NUMBER OF RESIDUAL DEFECTS PER VERSION FOR TECHNIQUES

Version	Residual defects		
	To	Ttot	Tadd
v2	23	3	0
v3	14	0	0
v4	4	2	0
v5	5	0	0
v6	3	3	2
v7	1	0	0
v8	0	0	0
v9	0	0	0

Figure 3 presents boxplots that show the distribution of the average of age values for the residual defects undetected across versions for the three prioritization techniques (the number of data points represented by each boxplot is 30). The horizontal axis corresponds to techniques, and the vertical axis corresponds to the sum of the ages for all the residual defects undetected. The leftmost boxplot presents data for the control technique and the other two for the heuristics. Higher values indicate that residual defects persisted through more versions undetected. Examining the box plots for each technique, the original ordering yield higher age values compared to both heuristics, and has more spread of data. However, the heuristics performed very similar to each other.

Next, to formally address one of the issues raised in our first research question (the effect of the age of residual defects on prioritization), we performed statistical analysis.

The hypothesis associated with the age values in RQ1 is: The age values of residual defects undetected by test case

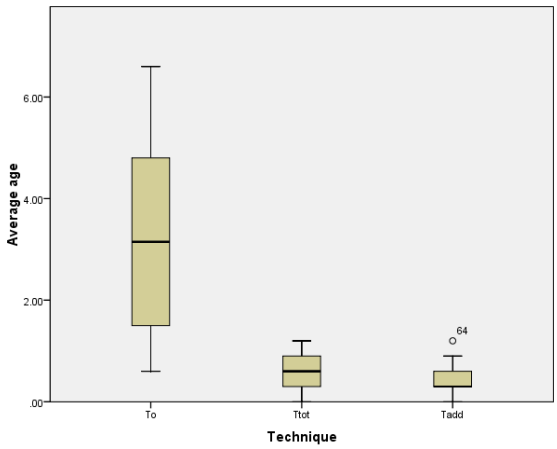

Figure 3. Boxplots for average age values of residual defects (all techniques, and versions)

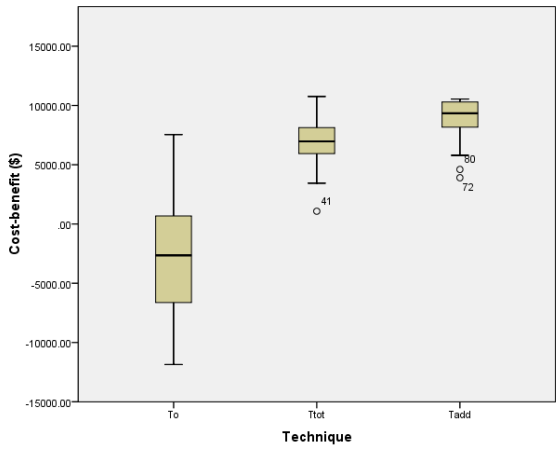

Figure 4. Boxplots for relative cost benefit values (all techniques, and versions)

prioritization techniques differ. To test this hypothesis, we performed the Kruskal-Wallis test (df = 2) for each technique, at a significance level of 0.05. The test results indicate that there is a significant difference between techniques (p-value < 0.0001), and thus, the hypothesis is supported.

We next performed multiple pair-wise comparisons using the Bonferroni tests. Table IV presents the results of the Bonferroni tests. The results show that differences between all heuristics and the control technique were statistically significant (p-value < 0.0001, at a significance level of 0.05), meaning the heuristics were significantly effective in reducing the age of residual defects compared to the control technique. The results, however, show that there is no significant difference between two heuristics.

Comparison	Estimate	Lower	Upper	p-value
To-Ttot	2.6	1.9	3.3	< 0.0001
To-Tadd	2.8	2.1	3.5	< 0.0001
Ttot-Tadd	0.14	-0.5	0.8	1

Table IV
RQ1: BONFERRONI ANALYSIS, THE AVERAGE AGE VALUES OF RESIDUAL DEFECTS.

B. Effects of residual defects on the cost-benefits of prioritization (RQ2)

Figure 4 presents boxplots that show the distribution of the relative cost benefits for all three techniques (the number of data points represented by each boxplot is 30). In this box plot, the horizontal axis is the prioritization technique and the vertical axis is the cost benefit in dollar. The higher the values, the greater the cost benefits are for a given technique. This box plot allows us to compare the cost benefit of the technique employed. Examining the boxplots for the three techniques, the heuristics are more beneficial than the control technique.

Next, to formally address this research question, we performed statistical analysis.

The hypothesis associated with RQ2 is: The relative cost-benefits that are associated with the age of residual defects between test case prioritization techniques differ. To test this hypothesis, we performed the Kruskal-Wallis test (df = 2) for each technique, at a significance level of 0.05. The test results indicate that there is a significant difference between techniques (p-value < 0.0001), and thus, the hypothesis is supported.

Since the hypothesis is supported, we further performed multiple pair-wise comparisons on the data using the Bonferroni test. The results are shown in Table V. The results show that differences between all heuristics and the control technique were statistically significant (p-value < 0.0001, at a significance level of 0.05), meaning the heuristics were significantly cost-beneficial compared to the control. The result for comparing the two heuristics was inconclusive (p-value = 0.049, at a significance level of 0.05).

Comparison	Estimate	Lower	Upper	p-value
To-Ttot	-9165	-11257	-7072	< 0.0001
To-Tadd	-11262	-13354	-9170	< 0.0001
Ttot-Tadd	-2097	-4189	-5	0.049

Table V
RQ2: BONFERRONI ANALYSIS, COST BENEFIT VALUES.

V. DISCUSSION

We now draw on the results of our analyses, together with additional consideration of our data, to derive practical implications of these results.

Prioritization techniques are effective in reducing the occurrence of residual defects and their age.: Our results indicate that heuristics were effective in reducing the occurrence of residual defects across entire system lifetimes.

In this study, we observed that the original technique was inferior to heuristics: a large number of residual defects were carried over to subsequent versions undetected. In the case of the original technique, 50 residual defects have been undetected across all versions, however, in the cases of the

non-feedback and feedback heuristics, only eight and two residual defects have been undetected, respectively. Comparing the two heuristics, the feedback technique performed slightly better than the non-feedback technique; in particular, the feedback technique performed well for the first three versions of program.

There are some interesting patterns in data for the original technique as we observed in Section IV: the number of residual defects decreased as the version number increased. In particular, for the first two versions (*v2* and *v3*), a large number of residual defects has been undetected, and some of them have existed in more than one version. One possible reason for this trend is the location of faults in those versions, the amount of changes between versions, and the code coverage ability of their test suites that reveal those faults. Another reason for this result could be the construction of the JUnit test cases supplied with the objects used in this study. It is typical in practice for developers to add new test cases at the end of a test suite. Since newer test cases tend to exercise new code, these new test cases may be more likely to be fault-revealing than previous test cases. Omitting those test cases could cause a large number of defects undetected and, as a consequence, those defects could become residual defects.

Our analysis of results also showed that prioritization heuristics were effective in reducing the age of residual defects across system lifetimes, which means heuristics could be effective in preventing defects from slipping through undetected into subsequent versions. In particular, some types of mutants were more persistent than others. For example, relational mutation operators were a dominant type for residual defects. None of residual defects contained object-oriented specific operators (e.g., argument order change, or access flag change) and logical operators. Several replicated studies could provide further insights into the relationship between residual defects and mutation operators.

Prioritization techniques are cost-effective when we accounted for the effect of residual defects.: Our analysis of results also showed that prioritization heuristics were highly cost-beneficial when we accounted for the costs associated with the age of residual defects. The higher the number of residual defects and their age value are involved, the higher costs in testing incur. Thus, prioritization heuristics, which produced a small number of residual defects and small age values, were significantly cost-beneficial compared to the control technique. Both heuristics did not allow residual defects to exist beyond one version. The feedback technique performed slightly better than the non-feedback technique, but the statistical result was inconclusive.

Practical implications of the results: From prior empirical studies of prioritization (studies that investigated cost-benefits of prioritization techniques [6], [23]), we learned that prioritization heuristics are more cost-effective compared to control techniques when organizations have limited regression testing resources (e.g., available testing time). In this study, we further considered additional cost factors (the number of residual defects and their age) that can be affected by the choice of regression testing techniques. We found that the empirical results of this study even more strongly advocate use of prioritization heuristics under the environment with limited testing resources, which would be a likely case for the majority of companies.

As addressed in Section I, several factors contribute to causing residual defects, such as inadequate test suites, time-to-market pressure, regression testing techniques used, an organization's business goals, or the type of system being built. One also could argue that residual defects occur simply because code segments that contain defects may be rarely exercised for a long period of time. While this conjecture makes sense, one clear consequence of allowing residual defects is that we have to pay high costs if residual defects eventually do arise. Without clear understanding of how factors affect the cause of residual defects and how serious the damages of residual defects can be, we cannot support such a claim. Building dependable software is the best practice to avoid such defects, but practicing effective testing processes will also play a significant role in reducing the occurrence of residual defects in the long run as we observed in our study. Furthermore, a focus on faults escaping testing and the costs of residual faults would have a greater chance of leading to techniques that might have high impact in terms of improving regression testing cost-effectiveness.

VI. Conclusions and Future Work

We have presented a study of assessing the effectiveness of test case prioritization techniques in reducing both the number of residual defects and their age. The results of our experiment show that heuristics can be effective in reducing both the occurrence of residual defects and their age. Our results also indicate that residual defects and their age have a strong impact on the cost-benefits of test case prioritization techniques.

The results of our studies suggest several avenues for future work. First, we intend to perform additional controlled experiments using several additional software systems considering different types of faults (e.g., real faults and hand-seeded faults) and additional mutation operators. Second, we intend to develop new regression testing techniques so that we can improve our chances of detecting faults that would otherwise become residual defects, beyond results that could be achieved using coverage or change information alone. Third, in this study, we considered only single level of time constrained testing environment. We plan to investigate interactions between regression testing technique's performance and different time constraint levels which can be affected by organizations' budget and testing processes.

REFERENCES

[1] Y. Chen, D. Rosenblum, and K. Vo, "TestTube: A system for selective regression testing," in *Int'l. Conf. Softw. Eng.*, May 1994, pp. 211–220.

[2] J. Offutt, J. Pan, and J. M. Voas, "Procedures for reducing the size of coverage-based test sets," in *Proc. Int'l. Conf. Testing Comp. Softw.*, Jun. 1995, pp. 111–123.

[3] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Trans. Softw. Eng. Meth.*, vol. 6, no. 2, pp. 173–210, Apr. 1997.

[4] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold, "Test case prioritization," *IEEE Trans. Softw. Eng.*, vol. 27, no. 10, pp. 929–948, Oct. 2001.

[5] A. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. Roos, "Time-aware test suite prioritization," in *Int'l. Symp. Softw. Test. Anal.*, Jul. 2006, pp. 1–12.

[6] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, "An empirical study of the effect of time constraints on the cost-benefits of regression testing," in *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, Nov. 2008, pp. 71–82.

[7] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 159–182, Feb. 2002.

[8] J. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Int'l. Conf. Softw. Eng.*, May 2002, pp. 119–129.

[9] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," in *Proceedings of the International Symposium on Foundations of Software Engineering*, Nov. 2004.

[10] N. Fenton and S. Pfleeger, "Science and substance: A challenge to software engineers," *IEEE Software*, pp. 86–95, Jul. 1994.

[11] N. Fenton, M. Neil, and D. Marquez, "Using bayesian networks to predict software defects and reliability," *Journal of Risk and Reliability*, vol. 222, pp. 701–712, May 2008.

[12] Y. Malaiya and J. Denton, "Estimating the number of residual defects," *IEEE International Volume*, pp. 98–105, 1998.

[13] K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma, "Regression testing in an industrial environment," *Comm. ACM*, vol. 41, no. 5, pp. 81–86, May 1988.

[14] W. Evanco, "Poisson analysis of defects for small software components," *Journal of Systems Software*, vol. 38, pp. 27–35, Jul. 1997.

[15] G. Holzmann, "Economics of software verification," in *Proc. Int'l. Workshop on Program Analysis for Software Tools and Engineering*, Jun. 2001, pp. 80–89.

[16] D. Leon and A. Podgurski, "A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases," in *Int'l. Symp. Softw. Rel. Eng.*, Nov. 2003, pp. 442–453.

[17] J. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test application frequency," in *Proceedings of the International Conference on Software Engineering*, Jun. 2000, pp. 126–135.

[18] X. Qu, M. Cohen, and R. G., "Configuration-aware regression testing: An empirical study of sampling and prioritization," in *Proceedings of the International Conference on Software Testing and Analysis*, Jul. 2008, pp. 75–86.

[19] H. K. N. Leung and L. White, "A cost model to compare regression test strategies," in *Proceedings of the Conference on Software Maintenance*, Oct. 1991.

[20] A. Malishevsky, G. Rothermel, and S. Elbaum, "Modeling the cost-benefits tradeoffs for regression testing techniques," in *Conf. Softw. Maint.*, Oct. 2002, pp. 204–213.

[21] J. Munson and T. Khoshgoftaar, "The detection of fault-prone programs," *IEEE Transactions on Software Engineering*, pp. 423–433, May 1992.

[22] T. Ostrand, E. Weyuker, and R. Bell, "Automating algorithms for the identification of fault-prone files," in *Int'l. Symp. Softw. Test. Rel.*, Jul. 2007, pp. 219–227.

[23] H. Do and G. Rothermel, "An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models," in *Found. Softw. Eng.*, Nov. 2006, pp. 141–151.

[24] ——, "Using sensitivity analysis to create simplified economic models for regression testing," in *Int'l. Symp. Softw. Test. Anal.*, Jul. 2008, pp. 51–61.

[25] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Int'l. J. Emp. Softw. Eng.*, vol. 10, no. 4, pp. 405–435, 2005.

[26] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Trans. Softw. Eng.*, vol. 32, no. 9, pp. 733–752, 2006.

[27] N. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 675–689, Jul. 1999.

[28] F. Shull *et al.*, "What we have learned about fighting defects," in *Int'l. Softw. Metrics Symp.*, Jun. 2002, pp. 249–258.

[29] A. Kinneer, M. Dwyer, and G. Rothermel, "Sofya: A flexible framework for development of dynamic program analysis for Java software," University of Nebraska–Lincoln, Tech. Rep. TR-UNL-CSE-2006-0006, Apr. 2006.

[30] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Int'l. Conf. Softw. Eng.*, May 2005, pp. 402–411.

[31] F. L. Ramsey and D. W. Schafer, *The Statistical Sleuth*, 1st ed. Duxbury Press, 1997.