2013 IEEE Sixth International Conference on Software Testing, Verification and Validation

Test Case Prioritization Using Requirements-Based Clustering

Md. Junaid Arafeen and Hyunsook Do
Department of Computer Science
North Dakota State University
Fargo, ND
$\{md.arafeen, hyunsook.do\}$*@ndsu.edu*

***Abstract*—The importance of using requirements information in the testing phase has been well recognized by the requirements engineering community, but to date, a vast majority of regression testing techniques have primarily relied on software code information. Incorporating requirements information into the current testing practice could help software engineers identify the source of defects more easily, validate the product against requirements, and maintain software products in a holistic way. In this paper, we investigate whether the requirements-based clustering approach that incorporates traditional code analysis information can improve the effectiveness of test case prioritization techniques. To investigate the effectiveness of our approach, we performed an empirical study using two Java programs with multiple versions and requirements documents. Our results indicate that the use of requirements information during the test case prioritization process can be beneficial.**

***Keywords*-Regression testing, test case prioritization, requirements-based clustering, empirical study**

I. Introduction

Software systems and their environments change continuously. They are enhanced, corrected, and ported to new platforms. These changes can affect a system adversely, thus software engineers perform regression testing to ensure quality of the modified systems. Because regression testing is responsible for a significant percentage of the costs for software maintenance and because the maintenance costs often dominate total lifecycle costs [1], [2], [3], regression testing is one of the largest contributors to the overall cost of software. To improve the cost effectiveness of regression testing techniques, many researchers have proposed and empirically studied various regression testing techniques, such as regression test selection (e.g., [4], [5]), test suite minimization (e.g., [6], [7]), and test case prioritization (e.g., [8], [9]).

Test case prioritization techniques schedule test cases to run more important test cases earlier so that we can detect faults earlier or provide earlier feedback to testers. Most of these techniques depend primarily on software code information, including code coverage or code dependency relations. However, the software systems are built upon the product requirements, and certain requirements are more important (e.g., contain features that are more frequently utilized by users) or error-prone than others. Often, testers have limited knowledge for understanding the problems (errors) with software products. Requirements information could potentially help identify the source of the problems in these cases. Further, by building relationships among requirements, source code, and the faults detected during regression testing, software engineers can maintain software products in a holistic way, thus they can eventually build more reliable products through seamless software development and maintenance practice.

While the importance of incorporating requirements information during the testing phase has been well understood by the requirements engineering community [10], only a few researchers have studied the use of requirements with software testing (black-box testing) [11], [12], [13]. Their work has shown that utilizing requirements information can be useful in improving the effectiveness of test case prioritization. For instance, Srikanth et al. [12] report that prioritized test cases based on the importance and fault proneness of requirements were able to detect severe faults earlier.

Typically, similar or related requirements are implemented in the same class or in classes under the same subsystem, which makes a software product cohesive. This means that test cases associated with a similar or related set of requirements exercise a similar set of classes. As we observed from our previous study [14], test cases with common properties tend to have similar fault detection ability. Reordering test cases considering the relationship between tests and requirements could help improve the regression testing process. (e.g., Developers could detect more faults earlier by running test cases with different properties sooner.) Prior research [11], [12], [13], however, has not attempted this approach.

Thus, in this research, we investigate whether clustering test cases based on requirement similarities could improve the effectiveness of regression testing techniques, particularly focusing on test case prioritization techniques. In this work, we implement new prioritization techniques that utilize requirements information. Our approach uses a text-mining technique that provides a means to cluster relevant requirements; incorporates code complexity for test case prioritization for each cluster; and, finally, creates a set of reordered test cases using the requirements priority. To inves-

978-0-7695-4968-2/13 $26.00 © 2013 IEEE
DOI 10.1109/ICST.2013.12

tigate the effectiveness of our approach, we have designed and performed empirical studies using one open source and one capstone program written in Java with multiple versions and requirements documents. Our results show that prioritized test cases using requirements-based clustering approaches improve the effectiveness of test case orders in terms of early fault detection.

The rest of the paper is organized as follows. Section II describes our new prioritization techniques in detail. Sections III and IV presents our experiment, including design, results, and analysis. Section V discusses our results and their implications. Section VI describes related work relevant to regression testing and test case prioritization techniques. Finally, Section VII presents conclusions and discusses future work.

II. Methodology

In this section, we describe our technical approach to test case prioritization using requirements clustering. Figure 1 summarizes our approach's main activities and how these activities are related to each other. The light gray boxes depict the main activities, and the ovals depict inputs and outputs associated with the activities. The approach consists of five main activities: requirements clustering, requirements-tests mapping, prioritization of test cases for each cluster, cluster prioritization, and test case selection from the clusters. The following subsections describe each activity in detail.

A. Requirements Clustering

To cluster the requirements, we use textual similarity among requirements. Textual similarity has been studied in the field of text mining for clustering documents and information retrieval [15], [16]. We group the requirements into clusters based on the distribution of words that co-occur in the requirements. This process includes three tasks: term extraction, term-document matrix construction, and k-means clustering. We describe these tasks in detail as follows:

- *Term extraction:* We consider each user requirement as a bag of words or terms. In this process, words are extracted from each requirement. The words that add no meaning the sentence, such as articles and prepositions, have been eliminated. After eliminating these words, all distinct terms across all the requirements are identified and used in the subsequent tasks.
- *Term-document matrix:* We use the distinct terms obtained from the previous step to create a term-document matrix. In this matrix, the rows correspond to the requirements, and the columns correspond to the distinct terms across all requirements. The matrix can be built in many ways. For instance, the matrix cells can list Boolean values indicating whether the terms are present in the corresponding requirements. The matrix can list the frequency of the word in the corresponding requirements or can list the term frequency-inverse document frequency.
Among these, we used the term frequency-inverse document frequency in this work because this approach is more suitable for text retrieval than others [17], [18]. The term frequency, *tf(t,d)*, is the number of occurrences of a term, *t*, in a document, *d*. The inverse document frequency, *idf*, provides a measure of how common the term is across all documents. The *idf* is calculated by taking the logarithm of the ratio for the total number of documents and the number of documents containing the term. We obtain the term frequency-inverse document frequency by multiplying the term frequency by the inverse document frequency (*tf*idf*). The *tf*idf* values of all the terms used in requirement R are utilized in the next step (clustering). Table I shows an example of the term-document matrix we just described. Suppose we have six requirements. After performing term-extraction, we identify a set of distinct terms, including vital, weight, warning, wait, etc. We calculate the *tf*idf* values of every term for each requirement. For instance, the *tf*idf* value of "vital" for Req Id 2, 4, and 5 is 0.210, 0.201, and 0.11, respectively. From these data, we can say that, Req 2 is more similar to Req 4 than Req 5 for the term "vital."

Table I
Term-Document Matrix

Req Id	vital	weight	warning	wait	...
1	0.000	0.112	0.240	0.000	...
2	0.210	0.000	0.100	0.000	...
3	0.000	0.000	0.000	0.220	...
4	0.201	0.132	0.000	0.120	...
5	0.11	0.140	0.140	0.150	...
6	0.000	0.102	0.200	0.040	...

- *k-means clustering:* Many clustering algorithms exist, such as agglomerative hierarchical clustering, k-means clustering,etc. Steinbach et al. [19] show that k-means clustering is suitable for document clustering, so in this work, we use k-means clustering. The k-means clustering approach allows us to specify the number of clusters. We use the "Hartigan and Wong" algorithm to implement k-means. In this algorithm, for n number of requirements and p number of terms, the k-means technique allocates each requirement to one of the clusters to minimize the inter-cluster sum of squares shown in the following equation.

$$Sum(k) = \sum_{i=0}^{n} \sum_{j=0}^{p} (x(i,j) - \overline{x(k,j)})^2 \qquad (1)$$

where $\overline{x(k,j)}$ is the mean variable, j, of all elements in group k.

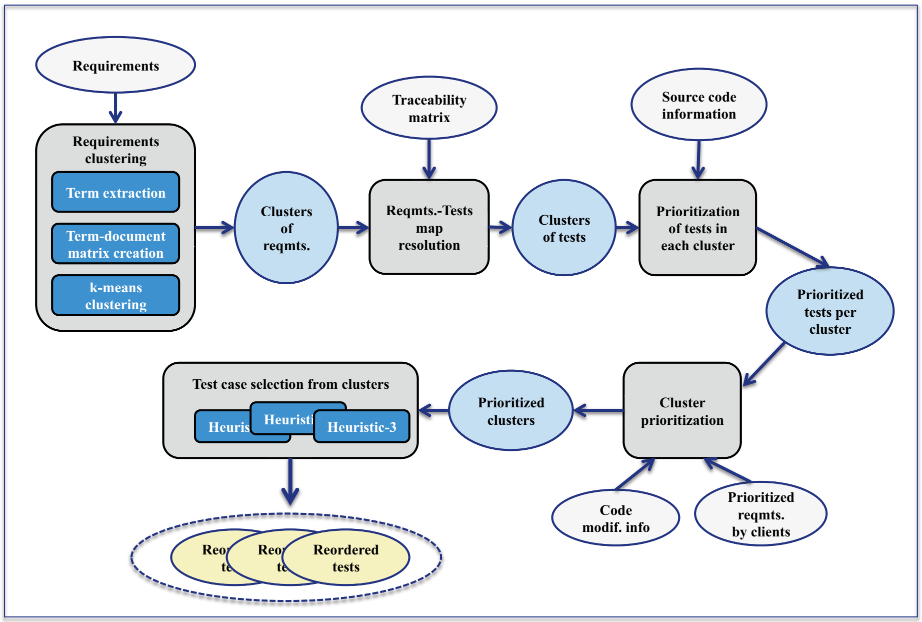

Figure 1. Overview of our approach

B. *Requirements-Tests Mapping Resolution*

Once we obtain the clusters of requirements, we utilize the requirement-test cases traceability matrix to collect test cases that are associated with each requirement cluster. Figure 2 summarizes the process. The two ovals on the left side represent the clusters of requirements. For instance, cluster 1 contains requirements 1, 3, and 4. The figure represents the requirement-tests traceability matrix. There are two cases (TC1 and TC2) associated with requirement 1. The requirements-tests mapping resolution process obtains the clusters of test cases (the ovals on the right side of the figure) by reading the requirements in the clusters and identifying their corresponding test cases from the matrix. For instance, cluster 1 on the right side contains five test cases (1, 2, 6, 7, and 8) that are associated with requirements 1, 3, and 4.

C. *Test Case Prioritization*

Having created clusters, now we apply prioritization techniques to them. There are many ways to prioritize test cases as explained in Section VI. In this work, we consider a code complexity metric.[1]

To calculate a code complexity metric, we used three types of information obtained from source code (Lines of Code, Nested Block Depth, and McCabe Cyclomatic Complexity) because they are considered good predictors for finding error-prone modules [20], [21], [22], and they are defined as follows:

- *Lines of Code (LOC):* It measures the total number of lines in a class. We consider only the executable code; we ignore the comments and the blank line inside the code.
- *Nested Block Depth (NBD):* It measures the number of nested statements in a method.
- *McCabe Cyclomatic Complexity (MCC):* It measures the number of linearly independent paths by analyzing the decision structure of a method.

Using these three data sets for each class, we calculated a code metric (*cm*) using the following equation:

$$cm = \frac{\frac{NBD}{Max(NBD)} + \frac{MCC}{Max(MCC)} + \frac{LOC}{Max(LOC)}}{3} \quad (2)$$

D. *Cluster Prioritization*

Not all requirements are equally important to clients. Therefore, certain software components associated with requirements that are more important to clients could be more frequently utilized by users when software is deployed. It means that certain requirements need to get more attention from testers. Thus, we prioritize the requirement clusters so that we can utilize their priority information when we select test cases from each cluster to obtain a complete set of reordered test cases. That is, we can visit the cluster with the higher priority earlier or select more tests from it.

Often, companies prioritize requirements and implement them incrementally based on a client's need and the product delivery schedule. From one of the projects we used in our experiment, we observed that, before each iteration of de-

[1]We could not use traditional code coverage-based techniques because we have mixed test cases (existing and new test cases) to prioritize, and this means code coverage for new test cases is not available. However, under different circumstances (e.g., using solely existing test cases or being able to use real fault-history information), we can apply various test case prioritization techniques.

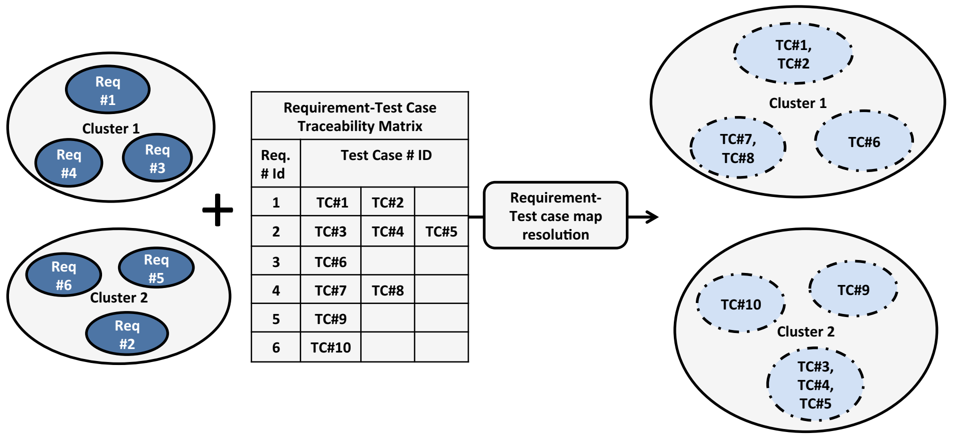

Figure 2. Requirement-tests mapping resolution

velopment, software developers categorize the requirements based on importance as follows:

- Commit (C): Developers will implement the given requirements (High Priority).
- Target (T): Developers will strive to implement the given requirements, but they will not guarantee to do so (Medium Priority).
- No-Commit (NC): Developers will implement the given requirements if they have time (Low Priority).

To prioritize requirement clusters, we use the commitment level defined above, and we also use code modification information that can be a good indicator for the presence of faults [23]. Using these two cluster prioritization levels, we construct the scale of weights shown in Table II.

Table II
SCALE OF WEIGHTS

Weight	Definition of Weight
1	No-Commit and Unmodified
2	Target and Unmodified
3	Commit and Unmodified
4	No-Commit and Modified
5	Target and Modified
6	Commit and Modified

The rank of cluster k is obtained using the following equation:

$$R(k) = \frac{1}{x(k)} \sum_{i=0}^{x(k)} w(i,k) \qquad (3)$$

where $x(k)$ is the number of requirements in cluster k and $w(i,k)$ is the weight of requirement i for cluster k.

E. Test Case Selection

Once we have clusters and prioritized tests for each cluster, we need to create a complete set of reordered test cases across clusters. To do so, we visit each cluster to select test cases using three different selection methods as follows:

- Original order of clusters: This method visits clusters in the order they were generated by the clustering tool, picks the first test case in the cluster, moves to the next cluster, and repeats the same process using a round robin method until all test cases have been picked.
- Random order of clusters: This method visits clusters in a random order and applies the same process as the first method.
- Prioritized order of clusters: This method visits clusters in the prioritized cluster order described in Section II-D. When we pick tests from the clusters, unlike the first two methods, we select more test cases from the higher-priority clusters. To do so, we calculate the average number of test cases per cluster, *t*. Then, we take *t* test cases from the highest-ranked cluster, and the number of selected test cases will go down gradually to 10% of *t* for the subsequent clusters. This process is repeated until all test cases have been picked.

III. EMPIRICAL STUDY

In this study, we address the following research question:

RQ: Does clustering test cases based on requirements improve the effectiveness of test case prioritization?

A. Objects of Analysis

Table III lists, for each of our objects of analysis, data on its associated "Req." (the number of requirements in the latest version of the program), "Ver." (the number of versions of the object program), "Classes" (the number of class files in the latest version of that program), "Size (KLOCs)" (the number of code lines in the latest version of the program), and "Test Cases" (the number of test cases available for the latest version of the program). Our experiment focuses on regression faults, so we generated mutation faults that only involve code modified in moving from one version of a system to a subsequent version using

ByteME (Bytecode Mutation Engine) [24]. The rightmost column, "Mutation Faults," is the total number of mutation faults for the program (summed across all versions).

Table III
EXPERIMENT OBJECTS AND ASSOCIATED DATA

Objects	Req.	Ver.	Classes	Size (KLOCs)	Test Cases	Mutation Faults
Capstone	21	2	67	6.82	42	118
iTrust	107	4	1029	30.30	142	200

The *Capstone* and *iTrust* systems were developed by college students as part of a class project. The *Capstone* project was developed by a team of graduate students. The students collaborated with a local software company and developed an online testing system which automates their examination procedure. The *iTrust* was developed and maintained by the RealSearch Research Group at North Carolina State University. *iTrust* is an open-source medical application that manages patients' medical records. For both programs, the test cases used in this study are functional test cases associated with requirements and written by software developers.

B. Variables and Measures

1) Independent Variable: To investigate our research question, we manipulate one independent variable: test case prioritization. We consider two control techniques that do not use clustering and six heuristic techniques as follows:

- Control (prioritization without clustering)
  - Original (Torig): Original ordering utilizes the order in which test cases are executed in the original testing scripts provided with the object programs.
  - Code metrics (Tcm): This technique uses a code metric to prioritize the tests without clustering.
- Heuristics (prioritization with clustering): We consider two heuristic groups that utilize clustering for each corresponding control technique. For each heuristic group, we consider three test selection methods as described in Section II-E.
  - Cluster-based original
    - ∗ Original (Tcl-orig-orig): This technique uses the original test case order for prioritization and the original cluster order for selection.
    - ∗ Random (Tcl-orig-rand): This technique uses the original test case order for prioritization and the random cluster order for selection.
    - ∗ Priority (Tcl-orig-prior): This technique uses the original test case order for prioritization and the prioritized cluster order for selection.
  - Cluster-based code metric
    - ∗ Original (Tcl-cm-orig): This technique uses the code metric for prioritization and the original cluster order for selection.
    - ∗ Random (Tcl-cm-rand): This technique uses the code metric for prioritization and the random cluster order for selection.
    - ∗ Priority (Tcl-cm-prior): This technique uses the code metric for prioritization and the prioritized cluster order for selection.

2) Dependent Variable and Measures: Our dependent variable is Average Percentage of Fault Detection (APFD). APFD [25] is the average for the percentage of fault detection during the execution of a test suite. The APFD value ranges between 0 and 100, and the closer the value is to 100, the better the prioritization technique is. (See reference [25] for the formal definition of APFD.)

C. Experiment Setup and Procedure

To perform prioritization, we require several types of information: requirements-traceability matrix, priorities of requirements, requirements-modification history, clusters of test cases, and code metric information.

In the case of *iTrust*, a traceability matrix was created by the developers. Two graduate students created priorities for requirements and the requirements-modification history. In the case of *Capstone*, the software developers and clients created requirements priorities and a graduate student created the traceability matrix and the requirements-modification history.

To collect the test case clusters, we utilized the k-means clustering method as explained in Section II. We had a large number of descriptive requirements for the *iTrust* program, so we used five different cluster sizes (10, 15, 20, 25, and 30) to see whether the cluster size affects the results. We chose these cluster sizes considering the number of components in *iTrust*, which is 28 for the final version. In the case of *Capstone*, we only had 21 requirements for 6 components, so we used one cluster size, 6, for this case.

To obtain the Line of Code (LOC), McCabe Cyclomatic Complexity (MCC) and Nested Block Depth (NBD), we utilize Eclipse IDE. Eclipse is the most popular integrated development environment for the Java programming language. While calculating the LOC, we ignore the comments and blank lines in the source code.

To obtain the fault data required to investigate our research questions, we follow a similar approach to our earlier studies [26]. We create mutant groups by randomly choosing n mutation faults (between 1 and 10) from those available with the particular version of the program. We repeat this process for each program version and obtain a sequence of mutant groups for that sequence of versions. Thus, we construct 12 sequences of mutant groups for each program.

After collecting all the required data, we perform prioritization techniques, and calculate the APFD value for each of the test cases obtained from the techniques.

D. Threats to Validity

This section describes the threats to the validity of our study and the approaches we used to limit the effects of these threats.

Internal Validity: The measures we have utilized to calculate a code metric in this study have alternatives, and the choice of the measures could have affected the outcome of our study. However, we chose those metrics because they are considered good predictors for finding error-prone modules according to previous empirical investigations done by other researchers. The number of clusters chosen could also have affected the results of our study, but we chose them considering the number of components for the software systems and used multiple cluster sizes for *iTrust*. These limitations can be addressed through additional studies with different code metrics and different cluster sizes.

External Validity: We performed our study using two object programs equipped with requirements documents. The object programs were small and medium in size. While the results from our study cannot be interpreted in the context of industrial applications, we used different types of applications that came from various sources: one was an open-source application, and the other was an industrial application. Again, this limitation can be addressed through additional studies with a wider population.

IV. Data and Analysis

Our research question considers whether requirements-based test case clustering improves the effectiveness of test case prioritization. To answer this question, we compare techniques based on the results shown in Tables IV to IX. (We discuss further implications of the data and results in Section V.)

Tables IV, V, and VIII show the experimental results for the *iTrust* program. Tables VI, VII, and IX represent the results for the *Capstone* project. All values shown in the tables are average values (APFD values and improvement rates) for 12 datasets.

In Tables IV and VI, the first column represents the size of clusters, and the second column shows the heuristic techniques with the cluster-based original. Subsequent columns show the average APFD values for the heuristics and their improvement rates (as percentages) over the control technique (the original order without clustering (*Torig*)) for each version. Tables V and VII show the results for the techniques with a cluster-based code metric following the same structure as Tables IV and VI, respectively. The heuristics' improvement rates were measured over the code-metric-based without clustering (*Tcm*).

From the results for *iTrust* (Table IV), we observe that the heuristics (cluster-based original) outperformed the control technique (*Torig*) across all versions and cluster sizes except for version 1 with a cluster size 10. In particular, for version 1, at a cluster size 20, the heuristics produced the best results. All three heuristics produced improvement rates ranging from 55.52% to 65.37%. In the cases of versions 2 and 3, the heuristics produced the best results at a cluster size of 15.

The results for the techniques that used code metric (cluster-based code metric (Table V)) show similar trends: all heuristics outperformed the control technique (*Tcm*) across all versions and cluster sizes except for version 1 with the cluster size of 10. For version 1, at cluster size 20, the heuristics produced the best results. (The improvement rates range from 43.29% to 64.84%.) In the case of version 2, the improvement rates are not very different across the cluster sizes compared to version 1. For version 3, overall, all heuristics produced high improvement rates (ranging from 42.20% to 72.95%) compared with other versions.

In the case of *Capstone*, the heuristics using the original order (cluster-based original (Table VI)) outperformed the control technique (*Torig*), but the heuristics using the code metric (cluster-based code metric (Table VII)) are not better than the control technique (*Tcm*).

When we compare two heuristic groups (cluster-based original vs. cluster-based code metric), we can see that results from the two programs show different trends. In the case of *iTrust* (Table VIII), we observe that the results vary with the size of the clusters and version. With version 1, for the cluster sizes of 10, 15, and 20, the cluster-based original group is better than the cluster-based code metric group except for one case. For the cluster sizes of 25 and 30, the results are reversed. For version 2, the cluster-based code metric group is better than the cluster-based original group for all but three cases. In the cases of *Capstone* (Table IX), the cluster-based code metric group is better than the cluster-based original group for all cases. For version 3, the cluster-based code metric group outperforms the cluster-based original group for all cases.

To show our results visually, we present them in boxplots. (Due to space limitations, we show the boxplots for *iTrust* only.) Figure 3 presents the boxplots that show APFD values for the control techniques and heuristics for all versions of *iTrust*. The figure contains 15 subfigures. The three columns of subfigures present the results for three versions. The five rows present results for five different sizes of clusters: C-10, C-15, C-20, C-25, and C-30, respectively.

Each subfigure contains boxplots for eight prioritization techniques, showing the distribution of APFD values for those techniques. The two leftmost boxplots (*Torig* and *Tcm*) present data for the control techniques (without clustering), and the rest present data for the six heuristics. (Due to space limitations, we abbreviate names for the heuristics, and Table X presents a legend for the heuristics.) The horizontal axis corresponds to techniques, and the vertical axis corresponds to APFD values. Because we measure results across 12 sequences of mutant groups, the number of data points in each boxplot is 12.

Table IV
APFD Comparison Chart (iTrust): Improvement over Torig

Cluster	Technique	$iTrust - v1$		$iTrust - v2$		$iTrust - v3$	
		APFD	Improvement over Torig (%)	APFD	Improvement over Torig (%)	APFD	Improvement over Torig (%)
10	Tcl-orig-orig	42.96	-1.83	62.38	30.47	60.81	27.19
	Tcl-orig-rand	43.35	-0.95	62.42	30.55	60.10	25.70
	Tcl-orig-prior	43.59	-0.38	57.28	19.81	70.60	47.67
15	Tcl-orig-orig	48.13	9.98	62.54	30.82	69.77	45.92
	Tcl-orig-rand	48.69	11.27	62.71	31.17	69.17	44.68
	Tcl-orig-prior	51.68	18.11	64.12	34.11	76.49	59.98
20	Tcl-orig-orig	68.06	55.52	61.59	28.82	63.60	33.03
	Tcl-orig-rand	68.66	56.90	62.16	30.02	64.54	35.00
	Tcl-orig-prior	72.37	65.37	63.42	32.64	72.06	50.72
25	Tcl-orig-orig	59.45	35.86	57.03	19.28	61.71	29.07
	Tcl-orig-rand	60.40	38.03	57.82	20.93	63.49	32.80
	Tcl-orig-prior	63.76	45.69	58.50	22.37	68.50	43.28
30	Tcl-orig-orig	54.58	24.73	59.78	25.04	64.70	66.88
	Tcl-orig-rand	54.71	25.02	61.07	27.74	64.36	25.70
	Tcl-orig-prior	56.61	29.36	62.54	30.80	70.54	47.54

Table V
APFD Comparison Chart (iTrust): Improvement over Tcm

Cluster	Technique	$iTrust - v1$		$iTrust - v2$		$iTrust - v3$	
		APFD	Improvement over Tcm (%)	APFD	Improvement over Tcm (%)	APFD	Improvement over Tcm (%)
10	Tcl-cm-orig	38.46	-15.98	68.26	39.87	68.48	52.73
	Tcl-cm-rand	38.88	-15.06	68.30	39.95	64.54	43.92
	Tcl-cm-prior	39.51	-13.68	57.78	18.40	75.31	67.94
15	Tcl-cm-orig	47.67	4.15	59.58	22.10	72.34	61.33
	Tcl-cm-rand	48.18	5.27	59.93	22.80	71.73	59.98
	Tcl-cm-prior	49.71	8.62	57.56	17.94	77.55	72.95
20	Tcl-cm-orig	65.58	43.29	62.67	28.41	66.69	48.73
	Tcl-cm-rand	66.22	44.69	63.65	30.43	67.68	50.94
	Tcl-cm-prior	75.45	64.84	65.33	33.87	72.41	61.48
25	Tcl-cm-orig	61.47	34.29	57.67	18.18	63.76	42.20
	Tcl-cm-rand	61.31	33.95	58.94	20.78	65.71	46.54
	Tcl-cm-prior	66.10	44.41	58.81	20.51	70.56	57.36
30	Tcl-cm-orig	56.81	24.13	65.00	33.20	67.49	50.51
	Tcl-cm-rand	57.03	24.60	66.80	36.88	67.06	49.55
	Tcl-cm-prior	60.10	31.31	63.63	30.38	72.35	61.34

Table VI
APFD Comparison Chart (Capstone): Improvement over Torig

Cluster	Technique	$capstone - v1$	
		APFD	Improvement over Torig (%)
6	Tcl-orig-orig	50.71	15.96
	Tcl-orig-rand	51.62	18.05
	Tcl-orig-prior	50.25	14.90

Table VII
APFD Comparison Chart (Capstone): Improvement over Tcm

Cluster	Technique	$capstone - v1$	
		APFD	Improvement over Tcm (%)
6	Tcl-cm-orig	61.19	-9.83
	Tcl-cm-rand	61.47	-9.41
	Tcl-cm-prior	67.86	0.00

Examining the boxplots for each cluster in the first column (version 1) of Figure 3, we see that the trends observed from the tables (average values) are similar to those of the boxplots. Overall, the results for version 1 show a wider distribution of data points than the other two versions. In particular, for all three cluster-based original techniques (*Tcoo*, *Tcor*, and *Tcop*), the differences between the best and worst APFD values are noticeable. In the cases of versions 2 and 3, except for a couple cases, the results for all techniques show similar data-distribution patterns. For version 3, across all cluster sizes, the heuristics consistently show great benefits over the control techniques.

V. Discussion and Implications

The results of our study indicate that clustering test cases based on requirements can improve the effectiveness of test case prioritization. Thus, test cases associated with a similar set of requirements could have similar fault-detection ability.

By examining the results, we drew the following observations. First, overall, the clustering approach was effective regardless of the selection method we utilized, although the results varied slightly across selection methods.

Second, our results varied across object programs, and in the case of *iTrust*, the results varied across the cluster sizes and versions. For *Capstone*, which is a relatively small program with a small number of tests and requirements, the improvement rates for the heuristics were low compared to the results for *iTrust*. One possible explanation for this result is that there version 1 underwent a major refactoring process and the requirements that are associated with the old code components have not been updated to reflect the

Table VIII
APFD Comparison Chart (iTrust): Heuristics based on code metrics vs original order

Version	Cluster	Tcl-cm-orig	Tcl-orig-orig	Tcl-cm-rand	Tcl-orig-rand	Tcl-cm-prior	Tcl-orig-prior
v1	10	38.46	42.96	38.88	43.35	39.51	43.59
	15	47.67	48.13	48.18	48.69	49.71	51.68
	20	65.58	68.06	66.22	68.66	75.45	72.37
	25	61.47	59.45	61.31	60.40	66.10	63.76
	30	56.81	54.58	57.03	54.71	60.10	56.61
v2	10	68.26	62.38	68.30	62.42	57.78	57.28
	15	59.58	62.54	59.93	62.71	57.56	64.12
	20	62.67	61.59	63.65	62.16	65.33	63.42
	25	57.67	57.03	58.94	57.82	58.81	58.50
	30	65.00	59.78	66.80	61.07	63.63	62.54
v3	10	68.48	60.81	64.54	60.10	75.31	70.60
	15	72.34	69.77	71.73	69.17	77.55	76.49
	20	66.69	63.60	67.68	64.54	72.41	72.06
	25	63.76	61.71	65.71	63.49	70.56	68.50
	30	67.49	64.70	67.06	64.36	72.35	70.54

Table IX
APFD Comparison Chart (Capstone): Heuristics based on code metrics vs original order

Version	Cluster	Tcl-cm-orig	Tcl-orig-orig	Tcl-cm-rand	Tcl-orig-rand	Tcl-cm-prior	Tcl-orig-prior
v1	6	61.19	50.71	61.47	51.62	67.86	50.25

Table X
Prioritization Techniques

Label	Technique
Tcoo	Tcl-orig-orig
Tcor	Tcl-orig-rand
Tcop	Tcl-orig-prior
Tcco	Tcl-cm-orig
Tccr	Tcl-cm-rand
Tccp	Tcl-cm-prior

changes. In the case of *iTrust*, except for the results of cluster size 10, all heuristics produced high improvement rates compared with those of *Capstone*. In particular, for version 3, the improvement rates were the best among other versions. When we examined the software artifacts for *iTrust* version 3, we observed that a large portion of the source code associated with new requirements had been modified and that only three old requirements were relevant to version 3. We speculated that these factors contributed to the outcome of version 3 because we used code-modification information when we prioritized the requirement clusters.

The findings of the study provide significant implications for software companies. By prioritizing test cases based on the requirements information, companies can improve their regression testing process, and also by understanding the relationships among the test cases, requirements, and source code, software engineers can identify the source of the errors more easily. Further, our approach provides a way to maintain the clear relationship between the test cases and software artifact, so software companies can manage their products in a holistic way. Therefore, they can build more reliable and dependable products.

VI. Related Work

There are different types of regression testing techniques, such as regression test selection, test case prioritization, and test suite minimization, but in this section, we discuss test case prioritization, the main focus of our work.

Test case prioritization techniques (e.g., [9], [27]) reorder test cases to increase the chance of early fault detection. The techniques help software engineers reveal faults early in testing, thus allow them to begin debugging earlier. A wide range of prioritization techniques have been proposed and empirically studied, and a recent survey by Yoo and Harman [28] provides an overview of these techniques. Depending on the types of information available, various test case prioritization techniques can be utilized, but most test case prioritization research has used source code information to implement prioritization techniques.

For instance, many researchers utilized code coverage information to implement prioritization techniques [9], [8], [29], and recent prioritization techniques used other types of code information, such as slices [30], change history [31], or code modification information and fault proneness of code [32]. Further, numerous empirical studies showed that prioritization techniques that use source code information can improve the effectiveness of regression testing [25], [26], [33].

As addressed in Section I, the software systems are built upon the product requirements, thus utilizing requirements could potentially help testing and maintenance activities. However, to date, only a few researchers have studied the use of requirements during software testing [11], [12], [13]. Srikanth et al. [12] present an approach to prioritizing test cases at the system level using system requirements, and they report that prioritized test cases based on the importance and fault proneness of requirements were able to detect severe faults earlier. Similar to work done by Srikanth et al. [11], [12], Srivastva et al. [13] utilize requirements information and consider risk factors involving the requirements to improve test case prioritization. Unlike these studies,

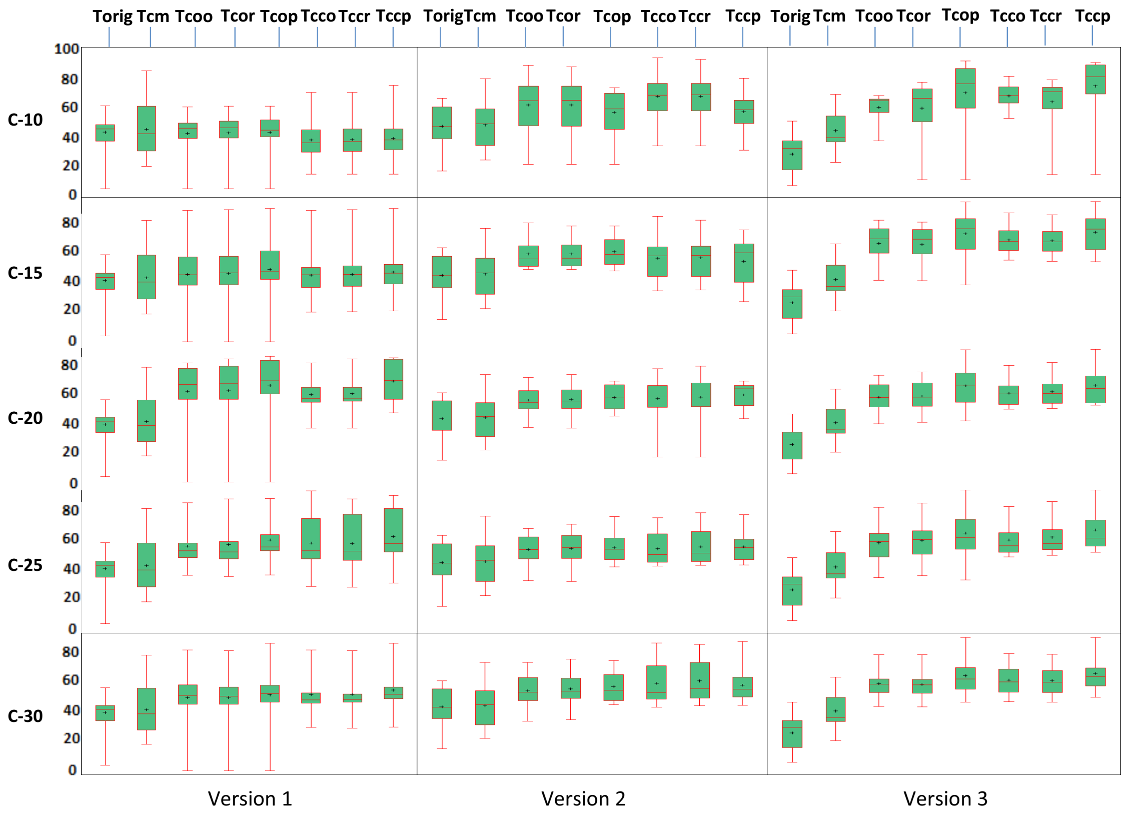

Figure 3. APFD Boxplots for all versions of *iTrust*

in this work, we present a requirements-based clustering approach that incorporates code information to improve the effectiveness of test case prioritization.

Another class of related work is clustering techniques. Leon and Podgurski [34] present a clustering technique that applies to test case prioritization. They use a test execution profile to cluster test cases and then randomly select test cases from clusters. Yoo et al. [35] cluster test cases using expert knowledge and then perform pairwise comparisons of the test cases in each cluster. They use expert knowledge for the pairwise comparison. Carlson et al. [14] also present clustering-based test case prioritization techniques that utilize code coverage information. Unlike these studies, in our work, we cluster test cases based on the requirements using a document-clustering approach that has been utilized in text-mining areas [15], [19].

VII. Conclusions and Future Work

We presented an empirical study that assesses requirement-based clustering technique in test case prioritization using one open-source and one Capstone project. Although numerous studies of test case prioritization have been conducted previously, most studies focused on utilizing source code information. Our study, in contrast, used requirements to group test cases in order to improve test case prioritization techniques.

The results show that the requirements-based clustering approach which incorporates traditional code analysis information can improve the effectiveness of test case prioritization techniques, but the results vary by the cluster sizes. The results suggest that, by grouping test cases associated with a similar or related set of requirements, we can manage regression testing processes more effectively.

We discussed the limitations of our study in Section III-D. To address those limitations, we intend to conduct additional empirical studies that consider different types of code metrics (e.g., inter-method communication and depth of inheritance) as well as various clustering and test case prioritization approaches, with a wider population of applications.

Acknowledgments

This work was supported, in part, by NSF Awards CNS-0855106 and CCF-1050343, and NSF CAREER Award CCF-1149389 to North Dakota State University.

References

[1] B. Beizer, *Black-Box Testing*. New York, NY: John Wiley and Sons, 1995.

[2] R. Binder, *Testing Object-Oriented Systems*. Reading, MA: Addison Wesley, 2000.

[3] M. J. Harrold and A. Orso, "Retesting software during development and maintenance," in *ICSM: Frontiers of Software Maintenance*, Sep. 2008, pp. 88–108.

[4] M. J. Harrold, D. Rosenblum, G. Rothermel, and E. Weyuker, "Empirical studies of a prediction model for regression test selection," *IEEE TSE*, vol. 27, no. 3, pp. 248–263, Mar. 2001.

[5] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE TSE*, vol. 22, no. 8, pp. 529–551, Aug. 1996.

[6] J. Jones and M. Harrold, "Test suite reduction and prioritization for modified condition/decision coverage," *IEEE TSE*, vol. 29, no. 3, pp. 193–209, 2003.

[7] 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.

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

[9] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE TSE*, vol. 27, no. 10, pp. 929–948, Oct. 2001.

[10] J. Bach, "Risk and requirements-based testing," *IEEE Computer*, vol. 32, no. 6, pp. 113–114, 1999.

[11] H. Srikanth and L. Williams, "On the economics of requirements-based test case prioritization," in *Int'l Workshop on EDSER*, May 2005, pp. 1–3.

[12] H. Srikanth, L. Williams, and J. Osborne, "System test case prioritization of new and regression test cases," in *ESE*, Aug. 2005, pp. 64–73.

[13] P. Srivastva, K. Kumar, and G. Raghurama, "Test case prioritization based on requirements and risk factors," *Software Engineering Notes*, vol. 33, no. 4, pp. 1–5, Jul. 2008.

[14] R. Carlson, H. Do, and A. Denton, "A clustering approach to improving test case prioritization: An industrial case study," in *ICSM*, Sep. 2011, pp. 382–391.

[15] N. Slonim and N. Tishby, "Document clustering using word clusters via the information bottleneck method," in *RDIR*, 2000, pp. 208–215.

[16] P. Willett, "Recent trends in hierarchic document clustering: A critical review," *Information Processing and Management*, vol. 24, no. 5, pp. 577–597, Aug. 1988.

[17] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information Processing and Management*, vol. 24, no. 5, pp. 513–523, Aug. 1988.

[18] D. Hiemstra, "A probabilistic justification for using tf.idf term weighting in information retrieval," *International Journal on Digital Libraries*, vol. 3, no. 2, pp. 131–139, 2000.

[19] M. Steinbach, G. Karypis, and V. Kumar, "A comparison of document clustering techniques," in *In KDD Workshop on Text Mining*, 2000.

[20] K. An, D. Gustafson, and A. Melton, "A model for software maintenance," in *ICSM*, Sep. 1987, pp. 57–62.

[21] N. Schneidewind and H.-M. Hoffman, "An experiment in software error data collection and analysis," *IEEE TSE*, vol. 5, no. 3, pp. 276–286, May 1979.

[22] W. Zage and D. Zage, "Evaluating design metrics on large-scale software," *IEEE TSE*, vol. 10, pp. 75–81, 1993.

[23] T. Graves, "Predicting fault incidence using software change history," *IEEE TSE*, vol. 26, pp. 653–661, Jul. 2000.

[24] P. Nagahawatte and H. Do, "The effectiveness of regression testing techniques in reducing the occurrence of residual defects," in *ICST*, Apr. 2010, pp. 79–88.

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

[26] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, "The effects of time constraints on test case prioritization: A series of controlled experiments," *IEEE TSE*, vol. 26, no. 5, Sep. 2010.

[27] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *ISSRE*, Nov. 1997, pp. 230–238.

[28] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritisation: A survey," *JSTVR*, pp. 67–120, Mar. 2010.

[29] J. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *ICSE*, May 2002, pp. 119 – 129.

[30] D. Jeffrey and N. Gupta, "Test case prioritization using relevant slices," in *Int'l Comp. Softw. and Appl. Conf.*, Sep. 2006, pp. 411–420.

[31] M. Sherriff, M. Lake, and L. Williams, "Prioritization of regression tests using singular value decomposition with empirical change records," in *ISSRE*, Nov. 2007, pp. 81–90.

[32] S. Mirarab and L. Tahvildari, "A prioritization approach for software test cases on Baysian Networks," in *FASE*, Mar. 2007, pp. 276–290.

[33] X. Qu, M. Cohen, and G. Rothermel, "Configuration-aware regression testing: An empirical study of sampling and prioritization," in *ISSTA*, Jul. 2008, pp. 75–86.

[34] D. Leon and A. Podgurski, "A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases," in *ISSRE*, Nov. 2003, pp. 442–453.

[35] S. Yoo, M. Harman, P. Tonella, and A. Susi, "Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge," in *ISSTA*, Jul. 2009, pp. 201–212.