A Clustering Approach to Improving Test Case Prioritization: An Industrial Case Study

Ryan Carlson*
* Microsoft
Fargo, ND
ryan.carlson@microsoft.com

Hyunsook Do†
† Department of Computer Science
North Dakota State University
Fargo, ND
hyunsook.do@ndsu.edu

Anne Denton‡
‡ Department of Computer Science
North Dakota State University
Fargo, ND
anne.denton@ndsu.edu

***Abstract*—Regression testing is an important activity for controlling the quality of a software product, but it accounts for a large proportion of the costs of software. We believe that an understanding of the underlying relationships in data about software systems, including data correlations and patterns, could provide information that would help improve regression testing techniques. We conjecture that if test cases have common properties, then test cases within the same group may have similar fault detection ability. As an initial approach to investigating the relationships in massive data in software repositories, in this paper, we consider a clustering approach to help improve test case prioritization. We implemented new prioritization techniques that incorporate a clustering approach and utilize code coverage, code complexity, and history data on real faults. To assess our approach, we have designed and conducted empirical studies using an industrial software product, Microsoft Dynamics Ax, which contains real faults. Our results show that test case prioritization that utilizes a clustering approach can improve the effectiveness of test case prioritization techniques.**

***Index Terms*—Regression testing, test case prioritization techniques, clustering approach, industrial case study**

I. Introduction

Regression testing plays an integral role in maintaining the quality of subsequent releases of software, but it is also expensive, accounting for a large proportion of the costs of software production [1]. We find this to be true when we perform regression testing on Microsoft *Dynamics Ax*, whose entire regression testing process requires several days for executing test cases and several more days for analyzing the results. In order to deal with this huge time and effort requirement, we need to improve the cost-effectiveness of regression testing. For similar reasons, many researchers have proposed various regression testing techniques, such as regression test selection (e.g., [2], [3]), test suite minimization (e.g., [4], [5]), and test case prioritization (e.g., [6], [7]).

Test case prioritization provides a way to run more important test cases earlier so that we can detect faults earlier or provide earlier feedback to testers. Test case prioritization also allows us to use a smaller set of test cases when testing time is limited, such as when nightly or weekly regression testing processes are used instead of the current quarterly process used for *Dynamics Ax*. In this way we can continue to provide lower possibilities that faults will escape into the released system [8], [9].

To date, various prioritization techniques have been proposed and empirically studied [6], [10], [11], [12]. Most of these techniques depend primarily on code coverage information or code complexity metrics. However, the various phases of software development and maintenance produce numerous software artifacts such as specifications, test harnesses, bug reports, and version control databases. We believe that an understanding of the underlying relationships in massive data about software systems, including data correlations and patterns, could provide information that would help improve test case prioritization techniques.

As an initial approach to investigating the relationships in massive data in software repositories, in this paper, we consider a clustering approach, which could simplify test case prioritization processes by dividing test cases into groups that have common properties. It has been conjectured [13], [14] that if test cases have common properties (e.g., having similar code coverage areas), then test cases within the same group may have similar fault detection ability. If this conjecture is correct, engineers may be able to manage regression testing activities more efficiently by using test case prioritization techniques that can utilize clustering approaches. For instance, if an organization does not have enough time to run all the test cases, by running a limited number of test cases from each cluster, they still could have a better chance to catch more faults than otherwise.

Thus, in this paper, we investigate whether the use of a clustering approach can help improve the effectiveness of test case prioritization techniques. In this work, we implement new prioritization techniques that incorporate a clustering approach and utilize code coverage, code complexity and history data on real faults that have been reported by users after the product has been released.

To investigate the effectiveness of our approach, we have designed and performed empirical studies using an industrial software product, Microsoft *Dynamics Ax*, which contains real faults. Our results show that test case prioritization that utilizes a clustering approach can improve the rate of fault detection of test cases, and reduce the number of faults that slip through testing when testing activities are cut short and test cases must be omitted due to time constraints, compared to the techniques that do not utilize clustering.

The rest of the paper is organized as follows. Section II discusses background information and related work relevant to regression testing and prioritization techniques. Section III describes our new prioritization techniques that incorporate a clustering approach. Sections IV and V present our experiments, including design, results, and analysis. Section VI addresses threats to validity, and Section VII discusses our results and their implications. Finally, Section VIII presents conclusions and discusses possible future work.

II. Background and Related Work

As software evolves, software engineers perform regression testing on it to validate new features and detect whether corrections and enhancements have introduced new faults into previously tested code. In practice, engineers often reuse all of the existing test cases to test the modified version of the software system; however, this retest-all approach can be expensive [15].

Researchers have studied various methods for improving the cost-effectiveness of regression testing. Regression test selection techniques (e.g., [2], [3]) reduce testing costs by selecting, from the existing test suite, a subset of test cases to execute on the modified software system. Two recent surveys [16], [17] provide an overview of regression test selection techniques. Test suite minimization techniques try to reduce the size of test suites by identifying and eliminating redundant test cases (e.g., [4], [5]). Both of these techniques reduce costs by reducing testing time and maintenance effort, but unless they are safe they can omit test cases that would otherwise have detected faults.

Test case prioritization techniques offer an alternative approach to improving regression testing cost-effectiveness. Test case prioritization techniques (e.g., [7], [18]) reorder test cases to increase the chance of early fault detection using various types of information available from software artifacts. These techniques help engineers reveal faults early in testing, allowing them to begin debugging earlier than might otherwise be possible. Depending on the types of information available relating to test cases, various test case prioritization techniques can be utilized, and to date, numerous test case prioritization techniques have been proposed. A recent survey by Yoo and Harman [17] provides a comprehensive overview of these techniques. Since we cannot discuss all the existing work here, we briefly introduce techniques that utilize different types of information.

Initially, most research utilized code coverage information to implement prioritization techniques [6], [7], [11], [15]. Using code coverage information, test cases can be prioritized in terms of the number of code statements, basic blocks, or methods they executed on a previous version of the software. This is a simple approach, but numerous empirical studies have shown that prioritization techniques that use code coverage information can improve the effectiveness of regression testing [8], [19], [9], [20].

More recently, several prioritization techniques that utilize other types of information have also been proposed. Jeffrey and Gupta [10] present an algorithm that prioritizes test cases based on their coverage of statements in relevant slices of the outputs of test cases, and compare their proposed technique with conventional code coverage techniques. Korel et al. [21] propose prioritization techniques based on system models that are associated with code information. Sherriff et al. [22] utilize change history to gather change impact information and prioritize test cases accordingly. Mirarab and Tahvildari [23] present Bayesian Network-based (BN) prioritization techniques that employ probabilistic inference algorithms with code modification information, univariate measures of fault-proneness, and test coverage information in an attempt to provide improved techniques.

More relevant to our work is work by Leon and Podgurski [13], who present prioritization techniques incorporating sampling methods that select test cases from clusters that are formed based on distributions of test execution profiles. Their techniques utilize clustering in test case prioritization, but the primary difference between their techniques and those we consider here is that they simply apply random selection of test cases from clusters for prioritization. In contrast, our approach applies prioritization within each cluster using code coverage, a code complexity metric, or real fault history information. Yoo et al. [14] study the use of expert knowledge to improve the effectiveness of prioritization techniques by pair-wise comparison of test cases, and propose clustering test cases into similar groups to facilitate the comparison process. By clustering test cases into groups, they reduce the number of pair-wise comparisons that are required for the human judgment process. Unlike our goal in this paper, their primary goal of using clustering is to help reduce the cost of human effort for pair-wise comparisons.

As we introduced briefly, there are many types of information available in implementing test case prioritization techniques. Among these, in this work, we utilize three different types of information. First, we consider a code coverage-based technique; we chose this technique because it is one of the most widely used techniques and a relatively simple but an effective approach. Second, we consider a code complexity-based technique; we chose this technique because it uses more complicated information than the first one, and it could have a good chance to improve the effectiveness of regression testing according to several prior studies [10], [22], [23]. Third, we consider a fault history-based technique because some researchers [11], [24] have conjectured that the use of fault history information could improve the effectiveness of test case prioritization techniques. However they have not used real faults in their experiments (they used hand-seeded faults), so it is difficult to conclude that the techniques that use fault history information can actually be effective in improving the effectiveness of test case prioritization. Since the object program we study in this work contains real faults, it makes it possible to investigate whether the use of fault history information can actually help improve the effectiveness of test case prioritization techniques.

III. PRIORITIZATION WITH CLUSTERING

We now describe the clustering and prioritization approaches that we use in this work. While we describe these in terms of steps used on Microsoft *Dynamics Ax*, the approach could be applied to any system for which the required information is available.

Our test case prioritization techniques require two main steps. First, we cluster test cases by retrieving code coverage and test case information from the version control system for *Dynamics Ax*. Second, using clustered test cases, we prioritize test cases based on software metrics we consider. The following subsections describe each of these steps in detail.

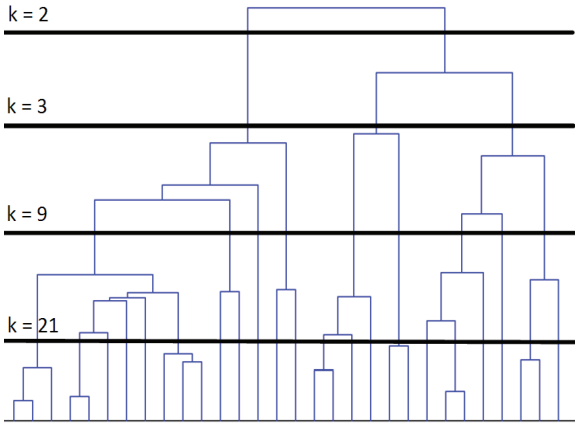


Fig. 1. Hierarchical Tree

A. Clustering Approach

For clustering, we use an agglomerative hierarchical clustering method [25], which is based on the pair-wise distances between test cases. The closest two test cases in terms of code coverage similarity are merged into a cluster first. Then the table of pair-wise similarities is recomputed with the new cluster being considered as a single element. The distance between the cluster and any of the remaining test cases is determined using average linkage, i.e., the distance is derived by averaging the distances between each of the elements of the cluster and the test case.

The algorithm then proceeds by using the new similarity table. At later stages, a test case may be merged with another test case, or with a cluster, or two clusters may be merged, based on the average distance between test cases that define the clusters. As a distance function we use Euclidean distance. The result of this algorithm is a tree of test cases that returns one clustering with k clusters for every k from 1 to the total number of test cases.

Figure 1 shows an example of a hierarchical tree. The numbers on the vertical lines in the figure indicate the number of clusters at the particular level in the tree. This approach provides a flexible way to adjust the number and size of clusters depending on the particular test case prioritization strategy we consider.

B. Prioritization Techniques

Having created clusters, now we apply prioritization techniques to them. To do so, first, we reorder test cases within each cluster using specific software metrics. Second, we generate the complete prioritized list of test cases by selecting test cases from each cluster. To perform the first step in our studies, we considered four different test case prioritization techniques that utilize the three types of information we mentioned in Section II:

- Code coverage
- A code complexity metric
- Fault history information
- Combination of code complexity metric and fault history information: We consider this technique to see whether utilizing two forms of sophisticated information can help improve the effectiveness of prioritization.

The following subsections describe each of the prioritization techniques in detail.

1) Prioritization Using Code Coverage: Our first prioritization technique utilizes code coverage information. The *Dynamics Ax* development environment allows us to collect method-level coverage information. For each test case, we collected the number of methods in the program that were exercised by that test case. Using this method coverage information, we reordered test cases in each cluster according to the total number of methods they cover simply by sorting them in order of total method coverage achieved.

To obtain a complete set of reordered test cases across clusters, we visited each cluster using a round robin method. We used the order of clusters that has been generated by the clustering tool, Matlab. Starting from the first cluster that the clustering tool generated, we picked the first test case in the cluster, added it to the prioritized list of test cases (initially an empty list), and removed the added test case from the cluster. Then, we moved to the next cluster, and repeated the same process until we had added all the test cases to the prioritized list. In cases where a cluster ran out of test cases due to the fact that the number of test cases varied with each cluster, we skipped that cluster and moved to the next cluster.

For instance, suppose we have five clusters (this means k=5 in Figure 1) and 15 test cases (T1, T2, ..., T15). Using our prioritization technique, we reordered test cases for each cluster as shown in Figure 2. Then, we created the prioritized list, *[T7, T2, T11, T15, T1, T3, T6, T5, T13, T10, T9, T8, T14, T12, T4]*, using the process that we just described.

2) Prioritization Using a Code Complexity Metric: Our second prioritization technique utilizes a code complexity metric. To calculate this metric, we collected two data sets, the number of lines of code (LOC) for a class and the method dependency count. To obtain LOC, we retrieved the project repository that stores all source code information. The repository provides an easy way to collect a simple count of non-blank lines for each class file.

We also collected dependency information directly from the Microsoft *Dynamics Ax* system. The Microsoft *Dynamics Ax*

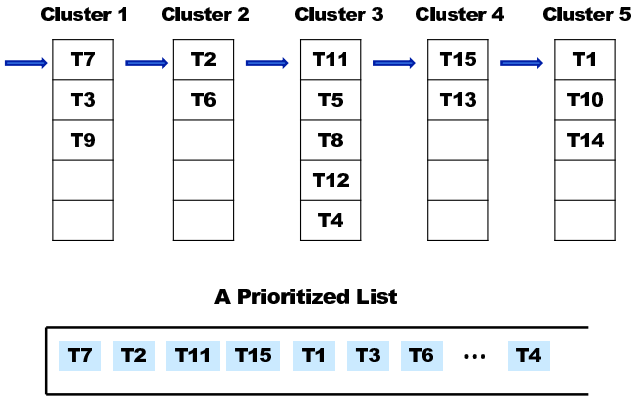


Fig. 2. An Example of Prioritized Test Cases in Clusters

system provides a cross reference mechanism that maintains the relationships between objects (e.g., methods or classes) in the *Dynamics Ax* system. This information is stored in internal SQL tables. We retrieved dependency information for each method, and recorded the number of invocations from other methods, which we refer as the number of dependency relationships.

Using these two data sets, first we calculated the LOC ratio and the number of dependency relationships ratio by dividing LOC and the number of dependency relationships by the largest number of LOC among classes and the largest number of dependency relationships among methods, respectively. Then, we calculated a code complexity metric (CC) by averaging these two ratios as shown in the following equation:

$$CC = \frac{\frac{LOC}{Max(LOC)} + \frac{DC}{Max(DC)}}{2}$$

where LOC is the number of lines of code, DC is the number of dependency relationships, Max(LOC) is the largest number of LOC among classes, and Max(DC) is the largest number of dependency relationships among methods. The code complexity metric value (CC) ranges between zero and one.

Similar to our first prioritization technique, we reordered test cases in each cluster in an order that puts the highest CC values earlier, and then selected test cases across entire clusters until all the test cases had been added to the prioritized list.

3) Prioritization Using Fault History Information: Test cases that have detected faults in previous versions could have high fault detection power when they are reused to test the current version of the program. If this is true, we can improve the test case prioritization process by running test cases that have fault detection history earlier than those that do not. Since the *Dynamics Ax* program comes to us with real faults and information relating these to test cases that detect them, we can explore this possibility by implementing prioritization techniques that use fault detection history of test cases.

To do this, first, we counted the number faults detected by each test case. Then, we calculated the fault detection ratio by dividing the number of detected faults by the total number of faults. The fault detection ratio ranges between zero and one, and we used these values to reorder test cases.

Similar to our first prioritization technique, we reordered test cases in each cluster in an order that puts the highest fault count ratios earlier, and then selected test cases across entire clusters until all the test cases had been added to the prioritized list.

4) Combined Technique: In the combined prioritization technique, we use the arithmetic mean of the code complexity value and the fault detection ratio value to prioritize the final test case list.

Again, similar to the second prioritization technique, we reordered test cases in each cluster in an order that puts the highest average values earlier, and then selected test cases across entire clusters until all the test cases have been added to the prioritized list.

IV. Study 1

As stated in Section I, in this paper, we investigate whether the use of a clustering approach can help improve the effectiveness of test case prioritization techniques, in application to the *Dynamics Ax* product. Thus, we conducted two empirical studies. This section describes the first empirical study, and the following section describes the second study.

In our first study, we investigate the following research question:

RQ1: Can a clustering approach help improve the effectiveness of test case prioritization techniques?

A. Object of Analysis

We used the Microsoft *Dynamics Ax* 2009 product including the initial release and the SP1 release of the product. The entire Microsoft *Dynamics Ax* product contains several million lines of application code written in multiple programming languages, such as C++ and X++ (a proprietary language that Microsoft developed) [26]. This size does not account for the kernel system code that provides the runtime engine, the development interface allowing application code to be written, the compiler and numerous other system pieces allowing tasks such as interfacing with the database.

There are several maintenance teams involved with Microsoft *Dynamics Ax*. One of the authors is involved in maintaining the financials subsystem. Therefore, due to the accessibility of software artifacts, in this study, we focus our study on the financials subsystem of Microsoft *Dynamics Ax*, which contains about 827 classes and 705 KLOCs (the latest version).

Like other products developed at Microsoft, the *Dynamics Ax* repository maintains all the software artifacts that have been produced during development and maintenance phases including fault information reported by the users. Through this repository we collected software artifacts required for our study as shown in Table I.

Table I lists, for each version of *Dynamics Ax*, data on its associated "Classes" (the number of class files), "Size (KLOCs)" (the number of lines of code, "Test Cases" (the

TABLE I
EXPERIMENT OBJECTS AND ASSOCIATED DATA

Objects	Classes	Size (KLOCs)	Test Cases	Faults
Version 0 (Dynamics Ax 4.0)	600	650	500	254
Version 1 (Dynamics Ax 2009)	787	687	656	139
Version 2 (Dynamics Ax 2009 SPI)	827	705.8	908	221

number of test cases), and "Faults" (the number of faults reported by users). Here, "faults" indicates system failures that have been detected and reported by users after the product has been released.

Test cases used in this study are functional test cases, which are the major type of test cases that have been used for measuring the quality of the *Dynamics Ax* system. These test cases were created by test engineers at Microsoft.

B. Variables and Measures

In this section, we describe independent and dependent variables.

1) Independent Variable: Given our research question, this study manipulated one independent variable: prioritization technique. Since we want to investigate whether a clustering approach can improve the effectiveness of prioritization techniques, we consider four control techniques that do not use clustering and their corresponding four heuristic prioritization techniques that use clustering. Table II summarizes the techniques.

- Control (prioritization without clustering): We consider four control techniques.
  - Code coverage (Tcov): This technique orders test cases based on the code coverage information without using clustering.
  - Code complexity (Tcc): This technique uses a code complexity metric to prioritize the tests without using clustering.
  - Fault history-based (Tfb): This technique orders test cases based on the fault history information without using clustering..
  - Combined (Tcb-clst): This technique combines Tcc and Tfb.
- Heuristics (prioritization using clustering): We consider four heuristics that utilize clustering for each corresponding control technique.

2) Dependent Variable: We consider one dependent variable: Average Percentage of Fault Detection (APFD). APFD [19] is defined as the average of the percentage of faults detected during the execution of a test suite. APFD provides us with a value between 0 and 100 that indicates how successful the prioritization technique is. The closer the value is to 100

TABLE II
TEST CASE PRIORITIZATION TECHNIQUES

Group	Technique	Description
Control	Tcov	Code coverage without clustering
	Tcc	Code complexity without clustering
	Tfb	Fault history-based without clustering
	Tcb	Combination of Tcc and Tfb
Heuristics	Tcov-clst	Code coverage using clustering
	Tcc-clst	Code complexity using clustering
	Tfb-clst	Fault history-based using clustering
	Tcb-clst	Combination of Tcc-clst and Tfb-clst

the better the prioritization technique is. More formally, Let T be a test suite containing n test cases, and let F be a set of m faults revealed by T. Let TF_i be the first test case in ordering T' of T which reveals fault i. The APFD for test suite T' is given by the equation:

$$APFD = 1 - \frac{TF_1 + TF_2 + \cdots + TF_m}{nm} + \frac{1}{2n}$$

Examples and empirical results illustrating the use of this metric are provided in [6].

C. Data Collection and Study Setup

To perform prioritization, our control techniques and heuristics required four data sets: code coverage information, the number lines of code, the number of dependency relationships, and fault history data, as outlined in Section III. Data collection required three steps as follows:

1) **Collect code coverage:** We collected the code coverage data for test cases executed against the System Under Test (SUT) using the code coverage recorder that *Dynamics Ax* provides as a part of its framework. We stored the code coverage data in the SQL database that lists information about which tests exercised which methods in the program. For each class and method, unique identifier values were assigned. This not only provides a key for the class or method tables that easily map to foreign keys in the related tables, but also provides a way to hide the actual names, protecting Microsoft's intellectual property and sensitive data. A partial data set we collected is shown in Table III.

   In Table III, the first two columns, "ClassID" and "MethodID", show the unique identifier values assigned to each class and method in the system. "No. of Dep." is the number of dependency relationships for a given method, "LOC" is the number of lines of code for a given class, and "CC" is the code complexity metric we defined in Section III-B. The "Fault" columns list Boolean values indicating whether the fault occurred for a given method: 0 indicates no faults occurred, and 1 indicates faults occurred. The "TestID" columns also list Boolean values indicating whether test cases exercised a given method or not: 0 indicates that test cases are not exercised, and 1 indicates that test cases are exercised.

TABLE III
AN EXAMPLE OF COLLECTED DATA

ClassID	MethodID	No. of Dep.	LOC	Code Complexity	Fault1	Fault2	...	Fault139	TestID1	...	TestID656
3519	8	2	149	0.008314	1	0		0	1		0
3519	9	108	149	0.055341	0	0		0	1		0
3519	10	9	149	0.011419	0	0		1	0		0
3519	11	9	149	0.011419	0	0		0	1		1
1815	12	4	50	0.004266	0	0		0	0		0
1815	13	1	50	0.002935	0	0		0	0		1
1815	14	4	50	0.004266	0	0		0	1		0

2) **Collect fault history:** To collect fault history information, we queried the fault history database of the prior version to find all the related fault history information for the financials subsystem. Similar to the test coverage recording process just explained, the code change information associated with each fault was traced using method names. We use this information to populate the fault columns found in Table III. For methods that have been changed due to fault corrections, we assigned 1, on others we assigned 0.
3) **Collect LOC and dependency information:** For LOC, we collected the number of non-blank lines including comments. Typically comments should also be excluded but since most class files only have a set of comments in each class file's header, known as XML Documentation [26], we had a consistent enough comparison across class files when leaving comments in the count. Dependency information was retrieved from the existing cross reference tables that exist as part of the Microsoft *Dynamics Ax* system as explained in Section III-B.

Once we collected all the required data, we formatted the data to make it readily usable for the clustering tool. We used Matlab, which provides the agglomerative hierarchical clustering method that we explained in Section III-A. In this study, we created ten clusters, and the median cluster size is 14. We chose the number of clusters based on the number of feature areas in *Dynamics Ax*, and each code coverage-based cluster was formed based on areas test cases exercise. Next, we obtained reordered test cases by applying the eight different prioritization techniques described in Section III-B. Then, APFD values were obtained from the reordered test suites, and the collected values were analyzed to determine whether the proposed prioritization techniques improved the rate of fault detection.

D. Data and Analysis

Our research question (RQ1) considers whether a clustering approach can help improve the effectiveness of test case prioritization techniques. To answer this question, we compare techniques based on the results shown in Table IV. The table contains two subtables, and each subtable corresponds to the results of each version of the program. Within each subtable, the first column lists control techniques, the second column lists their APFD scores, the third column lists heuristic techniques, and the fourth column lists their APFD scores. The last column shows the percentage improvement attained by the given heuristic over its corresponding control technique. For example, in the first subtable (version 1), the APFD value for Tcov is 50.78, and the APFD value produced by Tcov's corresponding heuristic technique, Tcov-clst, is 73.82. Tcov-clst's improvement rate over Tcov is 46%.

TABLE IV
APFD VALUES AND IMPROVEMENT RATES

Version 1				
Control techniques (Ctrl)	APFD	Heuristics (Hrst)	APFD	Improvement over control (Hrst/Ctrl)
Tcov	50.78	Tcov-clst	73.82	46%
Tcc	48.59	Tcc-clst	73.92	52%
Tfb	50.37	Tfb-clst	64.67	29%
Tcb	48.25	Tcb-clst	71.86	50%
Version 2				
Control techniques (Ctrl)	APFD	Heuristics (Hrst)	APFD	Improvement over control (Hrst/Ctrl)
Tcov	46.55	Tcov-clst	66.97	44%
Tcc	45.77	Tcc-clst	67.09	47%
Tfb	48.05	Tfb-clst	56.01	17%
Tcb	46.05	Tcb-clst	67.31	46%

The results indicate that all heuristic techniques outperformed their corresponding control technique. For version 1, on average, the heuristics improved the rate of fault detection by 45% over the control techniques (ranging from 29% to 52%). For version 2, the average improvement rate over the control techniques was 39% (ranging from 17% to 47%).

Overall, the technique that used code complexity with clustering (Tcc-clst) produced the best improvement rates over the corresponding control technique (52% and 47% for versions 1 and 2, respectively). In the case of Tfb-clst (fault history-based with clustering), the improvement rates were worst compared to other heuristics (29% and 17% for versions 1 and 2, respectively). In the cases of Tcov-clst and Tcb-clst, the improvement rates were a few points less than those of Tcc-clst.

When we compared the heuristic techniques against each other in terms of the fault detection rate (APFD scores), we can see that three heuristics (Tcov-clst, Tcc-clst, and Tcb-clst) produced similar APFD values. In particular, the values obtained by Tcov-clst and Tcc-clst were very close to each other for both versions. In the case of Tfb-clst, however, the

APFD values were several points less than those of other heuristics (64.67 and 56.01 for versions 1 and 2, respectively).

When we compared the control techniques, we can see that the differences between the APFD values were very small (raging from 48.25 to 50.78 for version 1, and ranging from 45.77 to 48.05 for version 2). For version 1, Tcov produced a slightly better result than others, and for version2, Tfb produced a slightly better result than others.

V. Study 2

The results of Study 1 suggest that a clustering approach can help improve the effectiveness of prioritization. While these results provide insights into the potential usefulness of clustering in test case prioritization, we do not know whether this observation holds when time constraints have been imposed. Considering the situations under time constraints is particularly important for industry because, in practice, software development processes often impose time constraints on regression testing. For instance, the Microsoft *Dynamics Ax* SE team often faces this situation. Thus, a better understanding of the effects of time constraints could lead to improved testing processes.

Our prior studies [8], [9] show that time constraints can affect assessments of the effectiveness of prioritization techniques, and also demonstrate that prioritization heuristics can be beneficial under time constraints. In this study, we further explore the findings from our prior studies by utilizing an industrial software product equipped with real faults.

Therefore, in the second study, we investigate the following research question:

RQ2: Do observations drawn from Study 1 hold when time constraints applied?

This study utilizes the same object of analysis, study setup, data sets, and prioritization techniques as those used in our first study. We thus do not repeat discussion of these here. Instead, we describe only the differences between this study and the prior one.

A. Variables and Measures

In this section, we describe independent and dependent variables.

1) Independent Variables: Given our research question, our experiment manipulated two independent variables: *prioritization technique* and *time constraints*.

Variable 1: Prioritization Technique: We utilize the same prioritization techniques used in Study 1 (Section IV).

Variable 2: Time Constraints: Time constraints imposed on regression testing processes are very common in practice, in particular, when the development cycle implements a nightly build-and-test approach that imposes a limited number of hours allowed before development changes begin again the next day as the Microsoft *Dynamics Ax* SE team does. Thus, to assess the effects of time constraints, our second independent variable controls the amount of regression testing.

In this study, we consider three different time constraint levels that our prior study utilized [2]: TCL-25, TCL-50, and TCL-75. They represent situations in which time constraints shorten the testing process by 25%, 50%, and 75%, respectively.

To implement time constraint levels, we followed the approach used in [8] - we assume that all of the test cases for the *Dynamics Ax* financials subsystem have equivalent execution times (this assumption is reasonable for the *Dynamics Ax* financials subsystem). We then manipulate the number of test cases executed to obtain results for different time constraint levels. For example, in the case of TCL-25, for each version and for each prioritized test suite, we halt the execution of the test cases as soon as 75% of those test cases have been run (thus omitting 25% of the test cases).

The time constraint levels we consider can be interpreted in light of the practicality of testing processes. TCL-75, which omits 75% of the tests, may represent a nightly build and test scenario where we only have a few hours each night to run as many tests as possible before changes will continue to be made the following day. The levels TCL-25 and TCL-50 may represent a weekly test run where more time is allotted to run tests, such as a weekend or a few days.

2) Dependent Variable: By omitting test cases due to time constraints, we could have faults that escape into the released system. Thus, as our dependent variable, we count the number of missed faults for each time constraint level.

B. Data and Analysis

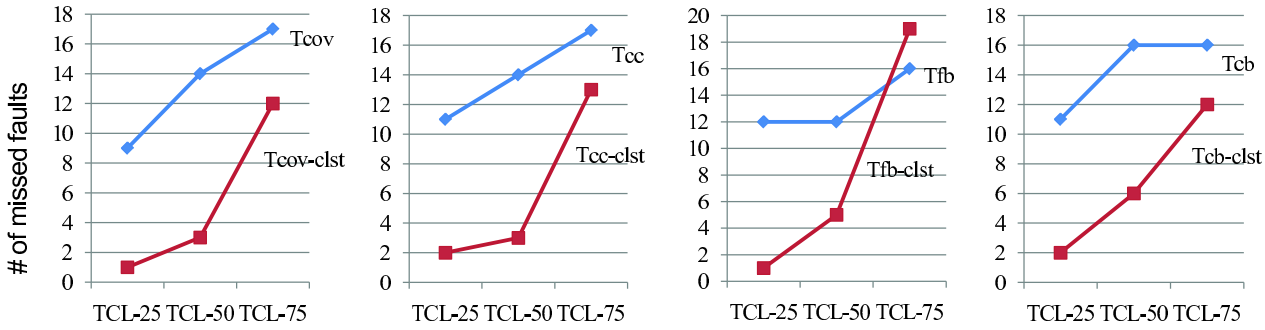
Our research question (RQ2) considers whether the observations we draw from Study 1 still hold when time constraints are applied. To answer this question, we compare techniques based on the results shown in Table V. The table shows the results of each version. For each version, data is organized per time constraint level (rows). For each time constraint level, the second column lists control techniques, the third column lists the number of missed faults produced by each control technique, the fourth column lists corresponding heuristics, and the fifth column lists the number of missed faults produced by each heuristic technique.

To show our results visually, we present them in lineplots as shown in Figure 3. Figure 3 presents lineplots of the number of missed faults for each pair of the control technique and its corresponding heuristic technique, at each time constraint level. The upper lineplots shows the results of version 1 and the lower lineplots show the results of version 2.

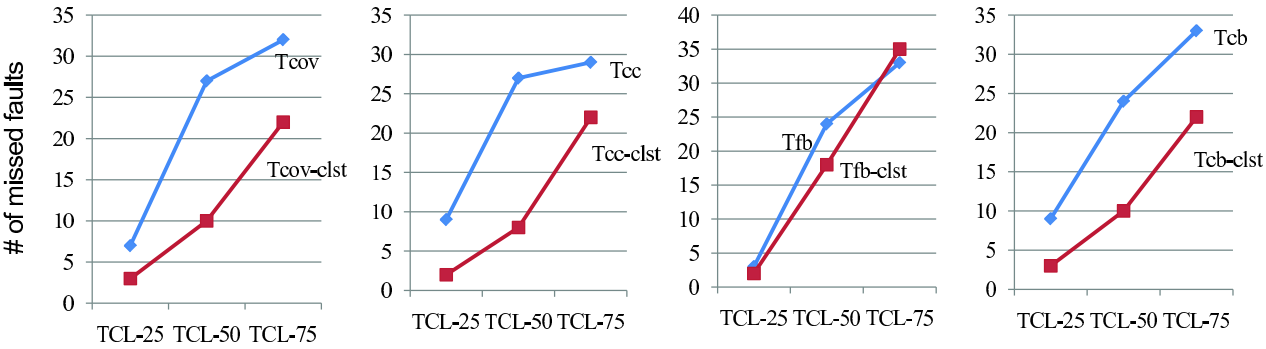
Examining the table and the lineplots, we have observed strong similarities with the results from Study 1. Compared to corresponding control techniques, the heuristic techniques missed relatively small numbers of faults across all time constraint levels for both versions with the exception of the fault history based technique (Tfb-clst) at TCL-75. Overall,the code complexity with clustering (Tcc-clst) technique produced the best results in terms of improvement over the control technique, which is consistent with the results from Study 1: For version 1, Tcc-clst reduced the number of missed faults compared to that of Tcc by 9, 11, and 4 at TCL-25, TCL-50 and TCL-75, respectively; For version 2, the number of

TABLE V
RESULTS WITH TIME CONSTRAINTS

Version 1					Version 2				
Time constraint level (TCL)	Control techniques	No. of missed faults	Heuristics	No. of missed faults	Time constraint level (TCL)	Control techniques	No. of missed faults	Heuristics	No. of missed faults
TCL-25	Tcov	9	Tcov-clst	1	TCL-25	Tcov	7	Tcov-clst	3
	Tcc	11	Tcc-clst	2		Tcc	9	Tcc-clst	2
	Tfb	12	Tfb-clst	1		Tfb	3	Tfb-clst	2
	Tcb	11	Tcb-clst	2		Tcb	9	Tcb-clst	3
TCL-50	Tcov	14	Tcov-clst	3	TCL-50	Tcov	27	Tcov-clst	10
	Tcc	14	Tcc-clst	3		Tcc	27	Tcc-clst	8
	Tfb	12	Tfb-clst	5		Tfb	24	Tfb-clst	18
	Tcb	16	Tcb-clst	6		Tcb	24	Tcb-clst	10
TCL-75	Tcov	17	Tcov-clst	12	TCL-75	Tcov	32	Tcov-clst	22
	Tcc	17	Tcc-clst	13		Tcc	29	Tcc-clst	22
	Tfb	16	Tfb-clst	19		Tfb	33	Tfb-clst	35
	Tcb	16	Tcb-clst	12		Tcb	33	Tcb-clst	22



(a) The number of missed faults for version 1



(b) The number of missed faults for version 2

Fig. 3. The number of missed faults for all techniques and versions

missed faults were reduced by 7, 19, and 7 at TCL-25, TCL-50 and TCL-75, respectively. Tcov-clst and Tcb-clst also showed similar results and Figure 3 clearly shows this trend: all three set's lineplots (Tcov vs Tcov-clst, Tcc vs Tcc-clst, and Tcb vs Tcb-clst) show a large gap between the control and heuristics techniques. The gap between the controls and heuristics is wider for version 1 than for version 2.

When we examined the results at each time constrain level, we observed different trends between time constraints among techniques. At TCL-25, for all heuristics, the numbers of missed faults were very small (ranging from 1 to 3; on average, 2) compared to the control techniques (ranging from 3 to 11; on average, 9). At TCL-50, the range of the number of missed faults is widened for both the heuristics (ranging from 3 to 18; on average, 8) and the control techniques (ranging from 12 to 27; on average, 20), and the difference between the heuristics

and the control techniques is more outstanding. At TCL-75, however, the gap between these two groups became smaller. In the case of the three heuristics (Tcov-clst, Tcc-clst, and Tcb-clst), the numbers of missed faults (ranging from 12 to 22) were smaller than those of the control (17 and 29). In the case Tfb-clst, however, the results were not better than the control techniques. As the lineplots show, the control technique, Tfb, produced smaller numbers of missed faults compared to Tbf-clst for both versions.

VI. Threats to Validity

In this section, we describe the threats to the validity of our empirical studies, and explain how we tried to reduce the chances that those threats affect the validity of our conclusions.

Internal Validity: The outcome of our studies could be affected by the choice of the number of clusters and the order of clusters. Our choice of the number of clusters, however, was based on the number of feature areas in *Dynamics Ax*, and each code coverage-based cluster was formed based on areas test cases exercise. Also, our choice of the order of clusters was based on the order that clustering tool generated, and different cluster orders could affect our results. Control for this threat can be achieved only through additional studies with different orders of clusters. Another factor involves the use of LOC at the class level. Our choice was based on the availability of information that can be extracted from the project repository. This choice could affect our results, but we believe that the difference would be minimal because except for several methods, the method size does not vary widely.

External Validity: We have studied an industrial software system with real faults reported by real customers. We have, however, focused on the financials subsystem for the Microsoft *Dynamics Ax* product, thus our findings might not apply to the entire *Dynamics Ax* product and products developed by other companies. Control for this threat can be achieved only through additional studies with the entire product and other industrial software products.

Construct Validity: Our dependent variables, APFD and the number of missed faults, do not account for the possibility that faults and test cases may have different costs. Accounting for these factors could improve the accuracy of our results.

VII. Discussion

We now discuss our results, together with additional consideration of our data, to derive practical implications.

Clustering can help improve prioritization

Our results strongly support the conclusion that the use of clustering in test case prioritization can improve the effectiveness of prioritization in terms of increasing the rate of fault detection. The results of our study also show that the conjecture we raised holds: if test cases have common properties (having similar code coverage areas for our case), then test cases within the same group may have similar fault detection ability.

One interesting thing we observed was that the technique that used fault detection history produced somewhat surprising results. When we examined the results of the control techniques, the fault history-based technique (Tfb) performed slightly better than others (Tcc and Tcb for version 1, and all others for version 2), but not as much as we expected or other researchers claimed [11], [24]. We also expected that the use of two types of sophisticated information would provide more improvement than the use of one type of information, but the results (Tcb and Tcb-clst) were not better than (or similar to) those of the techniques used one type of information.

Clustering is effective under time constraints

Our results also strongly support the conclusion that the use of clustering in test case prioritization can improve the effectiveness of prioritization in terms of reducing the number of missed faults when time constraints are imposed.

As noted in Section V-B, the results also show that the trend changed as time constraint level changed. In particular, up to 50% of test case omission, the heuristics performed far better than the control techniques, but when we omitted test cases further (75% cut), the differences were not outstanding and in the case of Tfb-clst, the results were reversed. The trends we observed in this study are different from those in our prior study [8]. The reason for this is that the two studies used different control techniques: in this study, we used prioritization heuristics as control techniques, but in [8], we utilized the original test order as a control.

Practical implications for software industry

The results from our two empirical studies lead to very significant practical implications for software industry because unlike our prior studies and a vast majority of other empirical studies on prioritization, these studies investigated prioritization problems in the industrial context by utilizing an industrial software product and its byproducts, such as code coverage, a code complexity metric, and real faults reported by users.

In particular, the findings from these studies are directly related to the Microsoft *Dynamics Ax* SE team. Thus, techniques we developed can be easily applied to their regression testing process so that they can produce a better quality product with less effort and costs. For instance, if the *Dynamics Ax* regression testing team detects more faults by using our proposed approach than otherwise during the limited regression testing process, such as weekly regression testing, then they can provide early feedback to developers so that developers can fix faults before the entire regression testing process begins. This means that fewer faults are found during the final regression testing stage, and thus the time and effort that are required for regression testing are reduced, and eventually the product quality and release schedule can be controlled.

Further, the fact that software development processes often impose time constraints on regression testing is true for almost all software companies. Thus, we hope that the results from these studies can provide useful insights into how software organizations manage their regression testing processes cost-

effectively considering their organizational and process contexts.

VIII. Conclusions and Future Work

We have presented two empirical studies assessing the use of clustering in test case prioritization in the context of an industrial software product. Our results show that our new test case prioritization techniques that utilize a clustering approach can improve the rate of fault detection of test cases and reduce the number of faults that slip through testing when testing activities are cut short.

As we discussed in Section VI, our studies have some limitations, which suggest several avenues of future work. First, in this study, we limited our attention to a subsystem of the *Dynamics Ax* product, the financials system. Future studies can be expanded by using the entire Microsoft *Dynamics Ax* product and possibly other application products.

Second, while we used three different types of information when we implemented test case prioritization techniques, other types of information can be utilized to further improve the effectiveness of prioritization. For instance, the current code complexity based technique utilized two sets of data (the number of lines of code and dependency relationships). Alternative to this technique is to utilize other types of software metrics, such as the number of child classes, the depth of inheritance, the age of classes, or change-proneness of classes. Considering these possibilities, we intend to develop new prioritization techniques so that we can further improve regression testing processes.

Third, as we discussed in Section I, we utilized a clustering approach as an initial approach to investigating the relationships in massive data in software repositories. Since this approach provides promising results, the next natural step is to investigate other alternative approaches, such as data mining (e.g., classification and association), that can deal with massive, complex, and heterogeneous software artifacts that software companies produce over time.

Acknowledgments

This work was supported in part by NSF under Awards CNS-0855106 and CCF-1050343 to North Dakota State University. We thank Microsoft for making Dynamics Ax available for empirical studies.

References

[1] M. J. Harrold and A. Orso, "Retesting software during development and maintenance," in *Proceedings of the International Conference on Software Maintenance: Frontiers of Software Maintenance*, Sep. 2008, pp. 88–108.

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

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

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

[5] 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.

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

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

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

[9] H. Do and G. Rothermel, "An empirical study of regression testing techniques incorporating context and lifecycle factors and improved cost-benefit models," in *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, Nov. 2006.

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

[11] J. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the International Conference on Software Engineering*, May 2002.

[12] A. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Time-aware test suite prioritization," in *Proceedings of the International Conference on Software Testing and Analysis*, Jul. 2006, pp. 1–12.

[13] D. Leon and A. Podgurski, "A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases," in *Proceedings of the International Symposium on Software Reliability Engineering*, Nov. 2003, pp. 442–453.

[14] S. Yoo, M. Harman, P. Tonella, and A. Susi, "Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge," in *Proceedings of the International Conference on Software Testing and Analysis*, Jul. 2009, pp. 201–212.

[15] A. Srivastava and J. Thiagarajan, " Effectively prioritizing tests in development environment," in *Proceedings of the International Symposium on Software Testing and Analysis*, Jul. 2002, pp. 97–106.

[16] E. Engstrom, P. Runeson, and M. Skoglund, "A systematic review on regression test selection techniques," *Information and Software Technology*, vol. 52, no. 1, pp. 14 – 30, 2010.

[17] S. Yoo and M. Harman, " Regression testing minimisation, selection and prioritisation : A survey," *Software Testing, Verification, and Reliability*, Mar. 2010.

[18] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *Proceedings of the International Symposium on Software Reliability Engineering*, Nov. 1997, pp. 230–238.

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

[20] X. Qu, M. Cohen, and G. Rothermel, "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.

[21] B. Korel, G. Koutsogiannakis, and L. Tahat, "Application of system models in regression test suite prioritization," in *Proceedings of the International Conference on Software Maintenance*, September 2008, pp. 247–256.

[22] M. Sherriff, M. Lake, and L. Williams, "Prioritization of regression tests using singular value decomposition with empirical change records," in *Proceedings of the International Symposium on Software Reliability Engineering*, November 2007, pp. 81–90.

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

[24] S. Kim and J. Baik, "An effective fault aware test case prioritization by incorporating a fault localization technique," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Sep. 2010.

[25] P. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*. Addison-Wesley, 2006.

[26] Microsoft Corporation, "XML Documentation Tags," http://msdn.microsoft.com/en-us/library/cc607340.aspx, Feb. 2010.