A CLUSTERING APPROACH TO IMPROVING TEST CASE PRIORITIZATION: AN INDUSTRIAL CASE STUDY

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
Of Agriculture and Applied Science

By

Ryan Curtis Carlson

In Partial Fulfillment of the Requirements
For the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science
Software Engineering

November 2010

Fargo, North Dakota

North Dakota State University
Graduate School

Title

A CLUSTERING APPROACH TO IMPROVING TEST CASE

PRIORITIZATION: AN INDUSTRIAL CASE STUDY

By

RYAN CURTIS CARLSON

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

ABSTRACT

Carlson, Ryan Curtis, M.S., Department of Computer Science, College of Science and Mathematics, North Dakota State University, November 2010. A Clustering Approach to Improving Test Case Prioritization: An Industrial Case Study. Major Professors: Dr. Hyunsook Do, Dr. Anne Denton.

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. 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 history data on real faults and code complexity. To assess our approach, we 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 rate of fault detection of test suites, 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.

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank my advisors, Dr. Hyunsook Do for all her time in helping me prepare this paper and Dr. Anne Denton for all the guidance with regards to the data mining aspects of this paper. I would also like to thank my supervisory committee members: Dr. Dean Knudson and Dr. Roger Green for their time and contributions.

I would also like to thank my beautiful wife for supporting me through this wild endeavor while caring for our four wonderful children, Anika, Austin, Bridger, and Cierra.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Regression testing is an important part of software development to maintain the quality of subsequent releases of a software product. The goal of regression testing is to uncover new faults that have been introduced into the previously tested system. The simplest and safest regression testing approach is to run all existing test cases. However, this process is expensive and time consuming [1, 4, 5] for large scale industrial size software systems. In fact in [9] we learn that one product with roughly 20,000 lines of code takes nearly seven weeks to run the full suite of regression tests. We face the same problem 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 requirement for regression testing, we need a way to improve the cost-effectiveness of regression testing. Prioritization is one way we can accomplish this by moving the tests that can uncover faults sooner in the execution pass as several researchers have proposed [9, 11, 23, 24, 25]. Using an effective prioritization technique will allow using a smaller set of tests when time is limited, such as overnight or each week instead of the current quarterly process. In this way we can continue to provide lower possibilities that faults will escape into the released system [1, 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. Several studies have been done on test case prioritization where a smaller set of tests are selected to be run that still provide a high-degree of confidence that the smaller set of tests will uncover defects [1, 4, 5, 7, 8, 9, 10]. Prioritization allows a decision to be made based on the budget or time constraint imposed on a software project as to how many tests from the entire suite will be run during a particular regression test run. Rothermel et al. [9, 11] present test case prioritization with the goal of greater fault detection which is described as, "a measure of how quickly a test suite detects faults during the testing process". We will also consider fault detection as the primary metric for success when grouping tests in this paper.

To date, various prioritization techniques have been proposed and empirically studied [1, 4, 5, 7, 8, 9, 10, 11, 12, 13, 22]. Most of these techniques depend primarily on code coverage information or code complexity metrics. However, the various phases of the software development process produce several different software artifacts such as specifications, test cases, bug reports, and version control databases. We believe that an understanding of the underlying relationships between these software artifacts can be useful in providing data to assist us in prioritizing tests. There has been some existing work such as in [6] that considers prior version data to improve software quality.

As an initial approach to investigating the relationships in massive data in software repositories, in this paper, we consider a *clustering* approach, which will simplify test case prioritization processes by dividing test cases into groups that have common properties. It has been conjectured [22, 26] 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 could have a better chance to catch more faults than otherwise.

To investigate the effectiveness of our approach, we conduct two empirical studies using an industrial Enterprise Resource Planning software system, Microsoft Dynamics Ax [3]. Enterprise Resource Planning (ERP) [2] is used to define a software package or system that is used by a business to manage the information and business functions that occur throughout a business. The example functions that an ERP system might provide are [2, 3], Manufacturing, Supply Chain Management, Financials, Project Management, Human Resources, Customer Resource Management (CRM), and Service Management. Each of these functions of an ERP system can be considered a module or sub-system. For the scope of this paper we will focus our research on just the financials sub-system of the Microsoft Dynamics Ax ERP software package. We are limiting our research to just the financials sub-system because of two reasons. First, the researcher for this paper is employed by Microsoft as a developer for the maintenance of just the financials sub-system. Second, the sheer size of the Microsoft Dynamics Ax product is quite large and managing the research data for the entire product was simply not conducive to completing research in a timely fashion. The financials sub-system is still quite large and measured in several thousand lines of code. More detail about the Microsoft Dynamics Ax product will be discussed in Section 2.1 below.

1.1. Problem Definition

The current process for regression testing the Microsoft Dynamics Ax product during maintenance is to run the entire test suite once every quarter throughout the year as we prepare to package all the previously released bug fixes in a hotfix rollup [14]. During this regression testing phase, all tests are executed sequentially with little thought given to the priority of which tests should be executed first. The entire process takes several days for executing the tests and several more days to analyze the results of the test runs. The analysis process requires humans to decipher if a failure reported is a real, repeatable failure that exists in the current version of the product, or if the failure is a random environmental issue that is not repeatable. In addition, it is possible a test reports a failure because of an intended change in the software. This would indicate a problem with the test itself. Again, it should be noted that this process only occurs once each quarter and defects in the software often remain for several weeks or months until the defect is uncovered by running the full set of tests. If a smaller set of tests were run more frequently, it is possible that the defect could have been found much sooner.

One important distinction is that we will differentiate between a *defect* found in the software and a *failure* reported from an external source just as found in [6]. The external source is typically a customer currently using the product. A *defect* can be found in the software system before or after the software is released, and a failure is described as an observable error in the software. For our purposes in this study, we will only consider post-release failures reported by an external source.

1.2. Significance of this Research

The ability of the Microsoft Dynamics Ax Sustained Engineering team to find defects more quickly by running fewer tests provides a huge benefit. This will help prevent defects from being released to customers systems and presenting themselves as failures in live customer systems. Also, it is well known that the cost of a software error carries a much greater cost when found later in the development cycle especially after released and found by a customer using the product [20]. If we are able to uncover defects more quickly, the overall cost of the defect will be much lower.

As mentioned in Section 1.1, the current regression testing process used by the Dynamics Ax Sustained Engineering team takes several days to run all the existing tests in a regression test run. This is followed by the analysis of any reported failures to understand if the failure is real and needs to be addressed, or if the failure was caused by an environmental issue or a problem with the test itself. This study can provide a new process for the team to reduce the number of tests that you would need to run and still provide a way to uncover defects in the system. Today there is no way for the team to accomplish this. For our study we will also look at various time constraints imposed on the regression testing process. This can provide a way for a very limited set of tests to be run overnight or a moderate set of tests to be run weekly that will allow more regression test runs to occur instead of waiting for the quarterly roll-up and having to process a larger set of defects reported from the run.

Lastly, the work from this paper will apply a new idea of using a clustering technique to the prioritization of test cases in order to improve regression testing process. The cost of clustering test cases was basically negligible in the entire process as it only required a few minutes to run the clustering algorithm. More detail about the clustering and related work can be found in sections 2.2 and 2.3. Prioritization of test cases is also an interesting research area where several studies have already proven the importance of doing so, but they have largely been focused on using code coverage metrics for prioritizing tests. Our work is important with respect to test case prioritization because the clustering technique can improve the prioritization process. The prioritized tests will allow us to select a smaller set of tests to run on a more frequent basis while being able to uncover defects more quickly.

1.3. Organization of the Paper

The remainder of this paper is organized as follows. Section 2 introduces the background and related work. Section 3 will discuss the study design and how the prioritization technique was applied. Section 4 presents the first empirical study. Section 5 presents the second empirical study. Section 6 presents the threats to validity. Section 7 discusses our results. Section 8 presents conclusions and discusses future work.

2. BACKGROUND AND RELATED WORK

2.1. Microsoft Dynamics Ax

Microsoft Dynamics Ax is an ERP product produced by Microsoft Corporation to provide businesses the ERP functions previously mentioned in Section 1. All of the various ERP functions can be accessed from a simple user interface displayed in a similar fashion as Microsoft Outlook. The navigation pane (Figure 1) of the application allows a user to select a functional area such as General Ledger and perform activities related to that area.

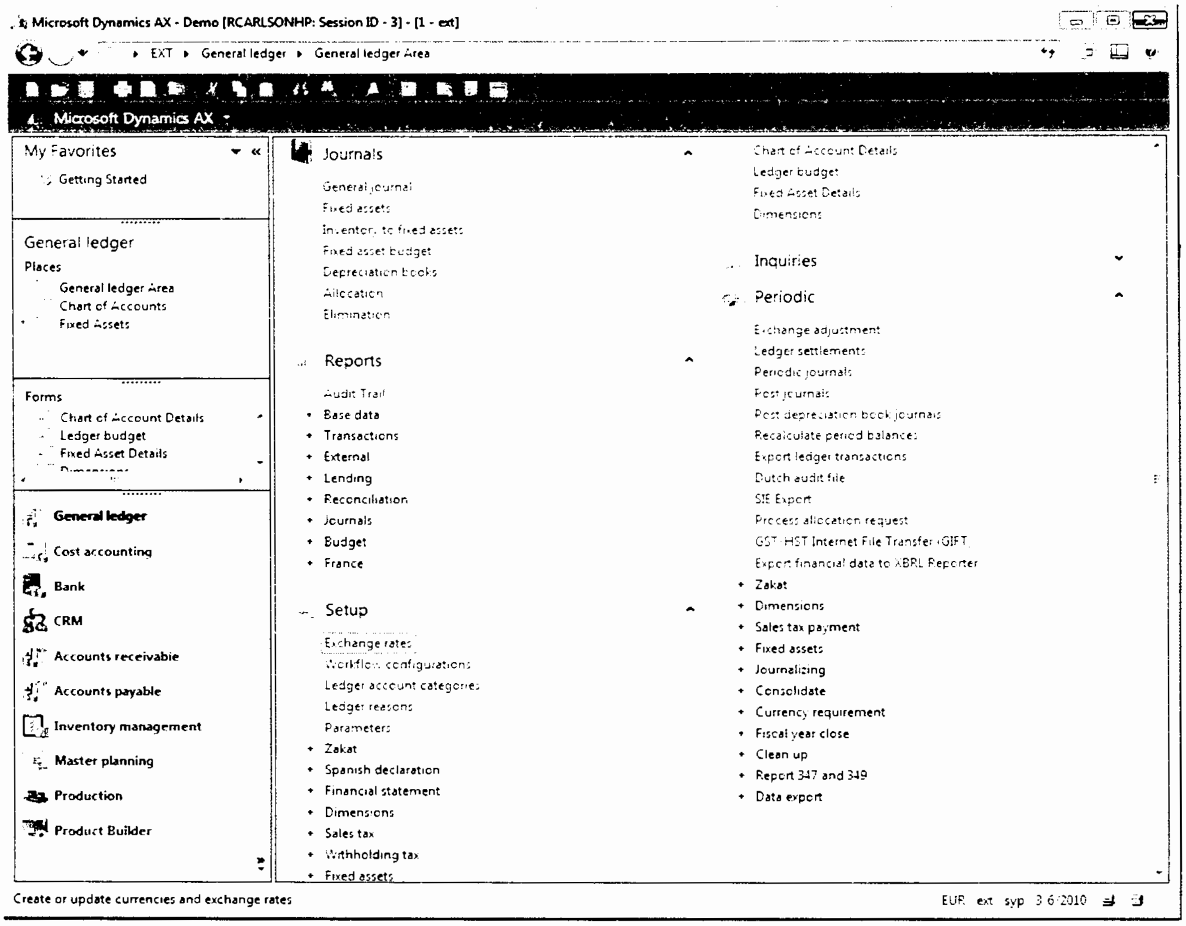


Figure 1. Microsoft Dynamics Ax

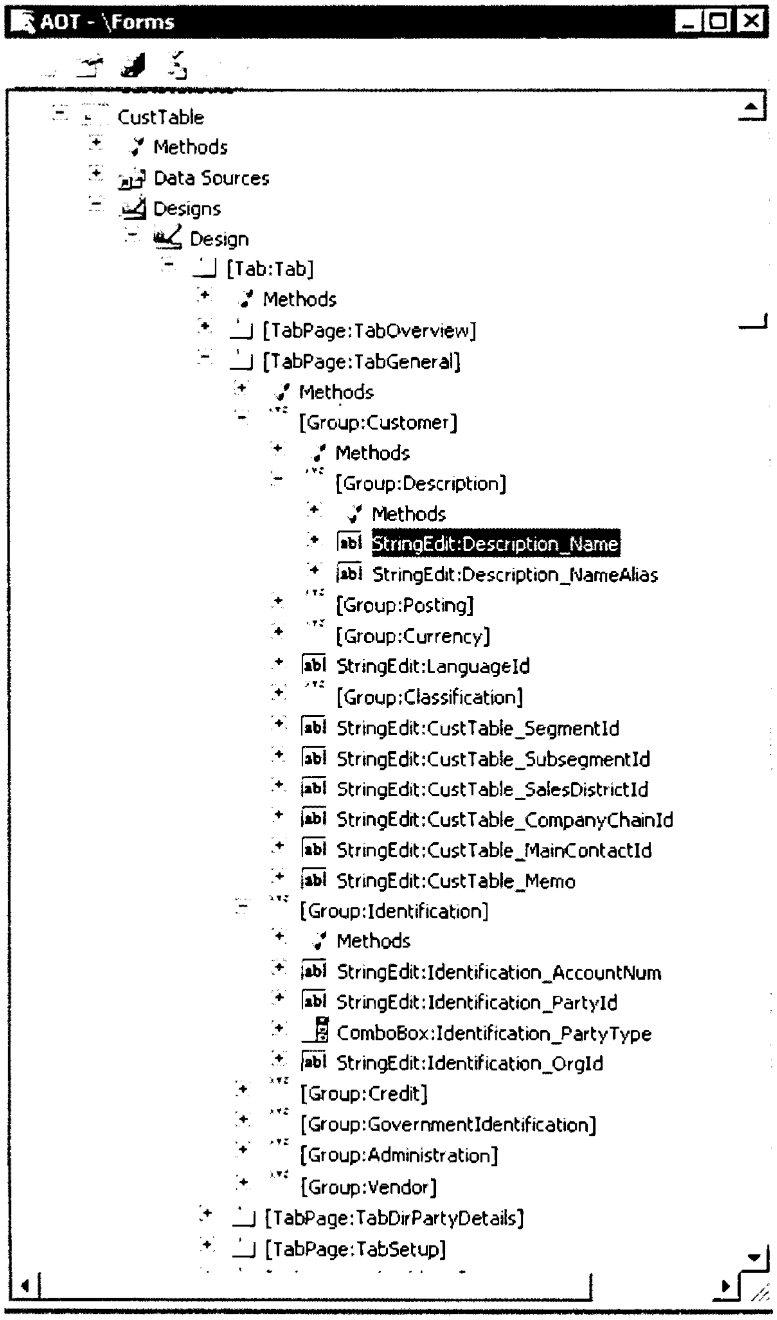
This paper will focus on the tests and code from only the financials sub-system which includes the General ledger, Accounts Receivable, Accounts Payable, Fixed assets, and Banking areas.

The Microsoft Dynamics Ax product is comprised of two different code bases. The foundational or system level code base is the runtime, compiler and related system frameworks that are written in C++. The application code to implement specific business application logic is written in a proprietary language known as X++. X++ is very similar in syntax and usage to Java. More information about X++ and its beginnings can be found in [18]. It is important to note that development in Microsoft Dynamics Ax happens both by modeling the design of the data structure, form layout, and report layout as well as by writing X++ code. Figure 2 shows the design of a customer form where an entire form can be modeled without writing a line of application code.

2.2. Test Case Prioritization.

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 [27].

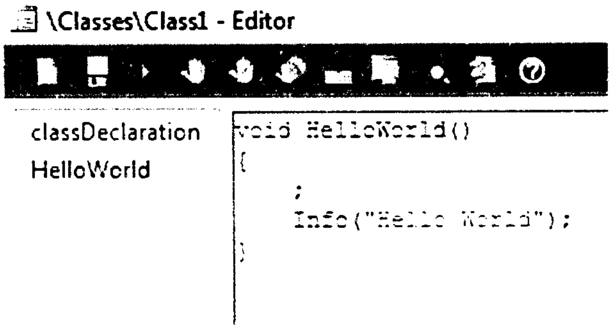
Researchers have studied various methods for improving the cost-effectiveness of regression testing. Regression test selection techniques (e.g., [23, 25]) reduce testing costs by selecting, from the existing test suite, a subset of test cases to execute on the modified software system. Test suite minimization techniques try to reduce the size of test suite by identifying and eliminating redundant test cases to reduce the maintenance cost [24, 28]. 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.



```
AOT - \Forms
CustTable
  Methods
  Data Sources
  Designs
    Design
      [Tab:Tab]
        Methods
        [TabPage:TabOverview]
        [TabPage:TabGeneral]
          Methods
          [Group:Customer]
            Methods
            [Group:Description]
              Methods
              StringEdit:Description_Name
              StringEdit:Description_NameAlias
            [Group:Posting]
            [Group:Currency]
            StringEdit:LanguageId
            [Group:Classification]
            StringEdit:CustTable_SegmentId
            StringEdit:CustTable_SubsegmentId
            StringEdit:CustTable_SalesDistrictId
            StringEdit:CustTable_CompanyChainId
            StringEdit:CustTable_MainContactId
            StringEdit:CustTable_Memo
          [Group:Identification]
            Methods
            StringEdit:Identification_AccountNum
            StringEdit:Identification_PartyId
            ComboBox:Identification_PartyType
            StringEdit:Identification_OrgId
          [Group:Credit]
          [Group:GovernmentIdentification]
          [Group:Administration]
          [Group:Vendor]
        [TabPage:TabDirPartyDetails]
        [TabPage:TabSetup]
```

Figure 2. Design of Customer Form

X++ code can be added or written by using the built in editor and can be seen in Figure 3.


\Classes\Class1 - Editor

```
void HelloWorld()
{
    ;
    Info("Hello World");
}
```

Figure 3. Example of X++ Code

Prioritizing test cases has been studied previously [1, 7, 8, 9, 11] in attempts to order tests that need to be executed with hopes that defects will be more quickly uncovered as the more important tests will be executed first. 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. Initially, most techniques utilized code coverage information to implement prioritization techniques [8, 9, 11, 27] and numerous studies have shown that prioritization can improve the effectiveness of regression testing [1, 4, 5, 7].

More recently, several prioritization techniques that go beyond the use of code coverage information have also been proposed. Jeffrey and Gupta [29] present an algorithm that prioritizes test cases based on their coverage of statements in relevant slices of the outputs of test cases. Korel et al. [30] propose prioritization techniques based on coverage of system models. Hou et al. [31] and Sampath et al. [32] study prioritization of test cases for testing web services software and web applications. Qu et al. [21] consider prioritization in the context of configurable systems, presenting algorithms for prioritization of configurations. Walcott et al. [19] present a technique that combines information on test execution times with code coverage information, and utilize a genetic algorithm to obtain test case orderings. Mirarab and Tahvildari [15] 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 [22], who present prioritization techniques incorporated 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 real fault history information and code complexity metrics. Yoo et al. [26] study the use of expert knowledge for prioritization by pair-wise comparison of test cases and propose clustering test cases into similar groups to facilitate the comparison 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 comparison.

2.3. Clustering

Clustering has recently been used to analyze the voluminous amounts of data generated from version control systems or fault reporting systems [6, 12, 16]. In [6] there was a considerable amount of research on the data retrieved from the version control system and the bug database for five major software projects developed at Microsoft.

Their approach for mining metrics was similar to those used in this paper as a bug database was linked to the version control system to provide the relationship between a bug and a software artifact. In [12,13] a considerable amount of data mining work was done to find related changes that may also need to happen or errors that may have been introduced based on code patterns from prior errors.

As found in [22] we see that a clustering technique has been used to group tests together based on similarity in the tests. The grouping of tests is used to assist in the prioritization of tests. This allows a reduced set of regression tests to be run by selecting a sub-set of tests from each cluster. In [22] they eventually used a combined approach to prioritize tests. They found that using a combination of 2 different techniques was more effective than each technique alone. Along with clustering tests together, they used a maximum code coverage technique in the selection or prioritization of their tests. The maximum code coverage technique is an effort to cover as much of the program to be tested as possible with the smallest number of tests. The idea of clustering tests together is similar to the approach we used in this paper. In this paper we will use clustering based on code coverage to improve on a basic technique that only looks at code complexity to prioritize tests.

3. PRIORITIZATION TECHNIQUES 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.

To implement our proposed test case prioritization techniques, we required 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.

3.1. Clustering Approach

We use agglomerative hierarchical clustering [11], 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 algorithm proceeds iteratively through the remaining test cases. The distances between a cluster and a test case, and between two clusters, are determined using average linkage, i.e., the distances between the elements of clusters are averaged. 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 4 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.

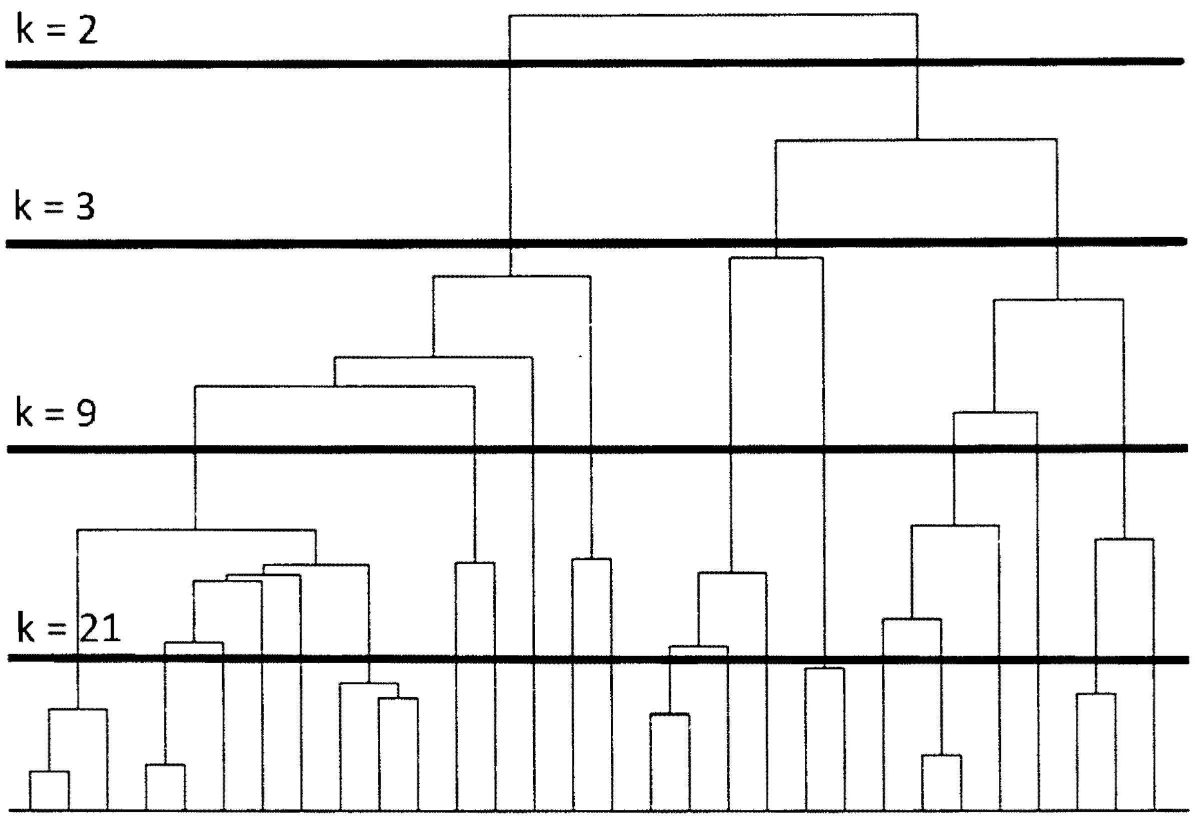


Figure 4. Hierarchical Cluster Tree

3.2. Prioritization Techniques

Having created clusters, now we applied new prioritization techniques to them. To do so, first, we reordered test cases within each cluster using specific software metrics. Second, we generated the complete prioritized list of test cases by selecting test cases from each cluster. To perform the first step, we considered three different test case prioritization techniques that utilize the following information:

- Historical fault information
- Code complexity
- Combination of code complexity and historical fault information

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

Prioritization Using a Code Complexity Metric

Our first 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 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 relationship 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:

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

The code complexity metric value (CC) ranges between zero and one. Using this code complexity metric, CC, we reordered test cases in each cluster in an order that puts the highest CC values earlier.

To obtain a complete set of reordered test cases across clusters, we visited each cluster using a round robin method. 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 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 (see Table 1), we skipped that cluster and moved to the next cluster.

For instance, suppose we have five clusters and 15 test cases. Using our prioritization technique, we reordered test cases for each cluster as shown in Table 1. 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.

Table 1. Example of Prioritized Test Cases in Clusters

Cluster 1	**Cluster 2**	**Cluster 3**	**Cluster 4**	**Cluster 5**
T7	T2	T11	T15	T1
T3	T6	T5	T13	T10
T9		T8		T14
		T12		
		T4		

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 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, 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 of 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.

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 first 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.

4. EMPIRICAL STUDY 1

As stated in Section 1, we wish to 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. The following subsections describe the first of these studies in detail. Section 5 discusses the second study in detail.

4.1. Research Question

In this study, we investigate the following research question:

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

4.2. Object of Analysis

For this study, we analyze the Microsoft Dynamics Ax 2009 product including the initial release and the SP1 release of the product. We will also look back at the previous version, Microsoft Dynamics Ax 4.0, as well to provide a set of defect data as an input to one of our calculations for prioritizing data. The Microsoft Dynamics Ax product contains several million lines of application code written in X++ (a proprietary language that Microsoft Developed) This 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 in Microsoft Dynamics Ax. The author is involved in maintaining the financials subsystem. Therefore, due to accessibility of software artifacts, in this study we focus 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 2.

Table 2. Experiment Objects and Associated Data

Objects	Classes	Size (KLOC)	Test Cases	Defects
Version 0 (Dynamics Ax 4.0)	~600	~650.0	n/a	254
Version 1 (Dynamics Ax 2009)	787	687.0	656	139
Version 2 (Dynamics Ax 2009 SP1)	827	705.8	908	221

Table 2 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 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.

For version 1, the number of faults per class varies from 0 to 23 (average: 0.08), and for version 2, it varies from 0 to 16 (average: 0.05).

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.

4.3. Variables and Measures

4.3.1. Independent Variable

Given our research question, this study manipulated one independent variable: prioritization technique. We consider one control technique and three heuristic prioritization techniques. Table 3 summarizes the techniques.

Table 3. Prioritization Techniques

Technique	Label	Description
Control	Tc	Conventional code complexity without clustering technique
Code Complexity	Tcc	Code complexity using clustering technique
Fault based	Tfb	Historical fault feedback prioritization using clustering
Combined	Tcb	Combination of Tcc and Tfb

- **Control (*Tc*)**: This technique serves as a control technique. The control technique (*Tc*) is one that simply uses a conventional code complexity metric to prioritize the tests without using clustering.
- **Heuristics (prioritization using clustering)**: As we described in Section 3.2, we consider three heuristics that utilize clustering:

  -**Code complexity (*Tcc*)**: This technique orders test cases based on the code complexity metric and clustering.

  -**Fault history based (*Tfb*)**: This technique orders test cases based on the fault history information and clustering.

  -**Combined (*Tcb*)**: This technique combines *Tcc* and *Tfb*.

4.3.2. Dependent Variable

We consider one dependent variable: Average Percentage of Fault Detection (APFD). APFD [5] 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 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 TFi 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:

$$APFD = 1 - (TF1 + TF2 + \ldots + TFm)/nm + 1/2n$$

4.4. Data Collection and Study Setup

To perform prioritization, our control technique 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 3.

Data collection required three steps which can be seen in the process diagram found in Figure 5. Each of these steps were automated, however manual intervention was needed to run each automated task to collect data. Collecting data from each system required a few minutes to a few hours at most from each system. First, 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 r· lated 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 4.

Table 4. An Example of Collected Data

ClassID	MethodID	#Dep	LOC	CC	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

In Table 4, the first two columns, "ClassID" and "MethodID", show the unique identifier values assigned to each class and method in the system. "#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 3.2. 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.

A second step was to query 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 queried from the bug database by method names. We use this information to populate the fault columns found in Table 4. For methods that have been changed due to fault corrections, we assigned 1, on others we assigned 0.

A third step was to collect the number of lines of code (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 [17], 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 3.2.

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 3.1. In this study, we created ten clusters, and the median cluster size is 14. Then, we obtained reordered test cases by applying the four different prioritization techniques described in Section 4.3.1. 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.

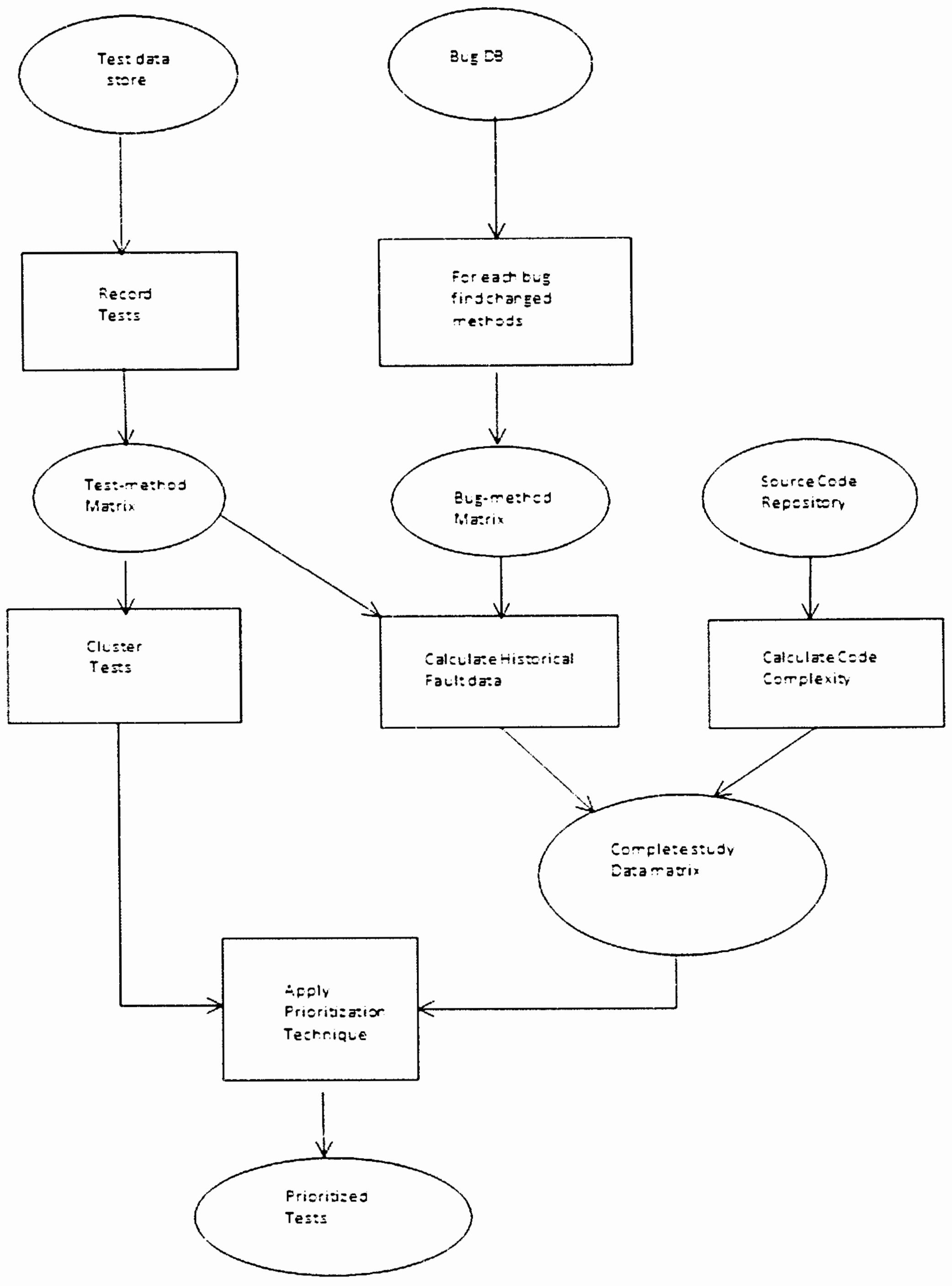

Test data store
Bug DB
Record Tests
For each bug find changed methods
Test-method Matrix
Bug-method Matrix
Source Code Repository
Cluster Tests
Calculate Historical Fault data
Calculate Code Complexity
Complete study Data matrix
Apply Prioritization Technique
Prioritized Tests

Figure 5. Study Process

4.5. 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 5. The table shows APFD values for each technique and version, and the percentage of the improvement over the control technique.

The results indicate that all heuristic techniques outperformed the control technique. For version 1, on average, the three prioritization heuristics improved the rate of fault detection by 44% compared to that of the control technique (*Tc*), and for version 2, the average improvement was 39%.

In particular, for version 1, the technique that used code complexity with clustering (Tcc) produced the best results, which shows a 52% improvement over the control technique. In version 2, both heuristics, Tcc (code complexity) and Tcb (combined), produced the best results, which show a 47% improvement over the control.

When we compared heuristic techniques against each other, the fault history based technique (Tfb) performed the worst. The APFD values (64.67 and 56.01 for version 1 and version 2, respectively) were several points less than those of the code complexity (Tcc) technique. The combined approach (Tcb) falls between these two techniques.

Table 5. APFD Results

Technique	Version 1		Version 2	
	APFD	Percentage improvement	APFD	Percentage improvement
Tc	48.59	0%	45.77	0%
Tcc	73.92	52%	67.09	47%
Tfb	64.67	33%	56.01	22%
Tcb	71.86	48%	67.31	47%

5. EMPIRICAL 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 because the current regression testing process that they use is a lengthy and time consuming process. Thus, a better understanding of the effects of time constraints could lead to improved testing processes.

Prior studies [1, 7] 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.

5.1. Research Question

For the second study in this paper 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.

5.2. Variables and Measures

5.2.1. Independent Variables

Variable 1: Prioritization Techniques

We will use the same prioritization techniques found in the study 1 above for this study.

Variable 2: Time constraints

Time constraints imposed on regression testing processes are very realistic 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 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 [1]: TCL-25, TCL-50, and TCL-75. They represent situations in which time constraints shorten the testing process by 25%, 50%, and 75%, respectively. The different levels are displayed in Table 6 below.

Table 6. Time Constraint Levels

ID	**Name**	**Description**
TC-25	25% reduction	25% of tests were omitted
TC-50	50% reduction	50% of tests were omitted
TC-75	75% reduction	75% of tests were omitted

To implement time constraint levels, we followed the approach used in [1] - 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).

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 over a weekend or spread across several days.

5.2.2. 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.

5.3. 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 7. Examining the table, we have observed strong similarities with the results from Study 1. Overall, compared to the control technique, 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) at TCL-75. In particular, the code complexity with clustering (Tcc) technique produced the best results, which is consistent with the results from Study 1.

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) compared to the control technique (11 and 9). At TCL-50, the gap between the heuristics and the control slightly widened, but still the difference between the heuristics (ranging from 3-18) and the control (14 and 27) is outstanding. At TCL-75, however, the gap between these two groups became smaller. In the case of the two heuristics (Tcc and Tcb), the numbers of missed faults (ranging from 12 to 22) were smaller than those of the control (17 and 29). In the case Tfb, however, the results were not better than the control.

Table 7. Results with Time Constraints Applied

Technique	Version 1			Version 2		
	TCL	# of Missed Faults	% found	TCL	# of Missed Faults	% found
Tc	25	11	56%	25	9	78%
	50	14	44%	50	27	33%
	75	17	32%	75	29	28%
Tcc	25	2	92%	25	2	95%
	50	3	88%	50	8	80%
	75	13	48%	75	22	45%
Tfb	25	1	96%	25	2	95%
	50	5	80%	50	18	55%
	75	19	24%	75	35	13%
Tcb	25	2	92%	25	3	93%
	50	6	76%	50	10	75%
	75	12	52%	75	22	45%

6. 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. Our choice, 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. 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, 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 be applied 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.

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 these additional factors could affect our results.

7. DISCUSSION

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

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, then test cases within the same group may have similar fault detection ability.

One interesting result we observed was that the technique that used fault detection history did not perform as well compared to other heuristics. This result was surprising because we expected that test cases that have fault detection history can provide high fault detection power. One possible reason for this is that since new test cases have been added as the system is upgraded (see Table 2), newly added or modified components could be more error-prone and thus these new test cases may be more likely to be fault-revealing than previous test cases.

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 5.3, 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 technique, but when we omitted test cases further (75% cut), the differences were not outstanding and in the case of *Tfb*, the results were reversed. We can see this detailed in Figure 6 below. The trends we observed in this study are different from those in study [1]. The reason for this is that the two studies used different control techniques: in this study, we used one type of prioritization techniques (code complexity based) as a control technique, but in [1], we utilized the original test order as a control.

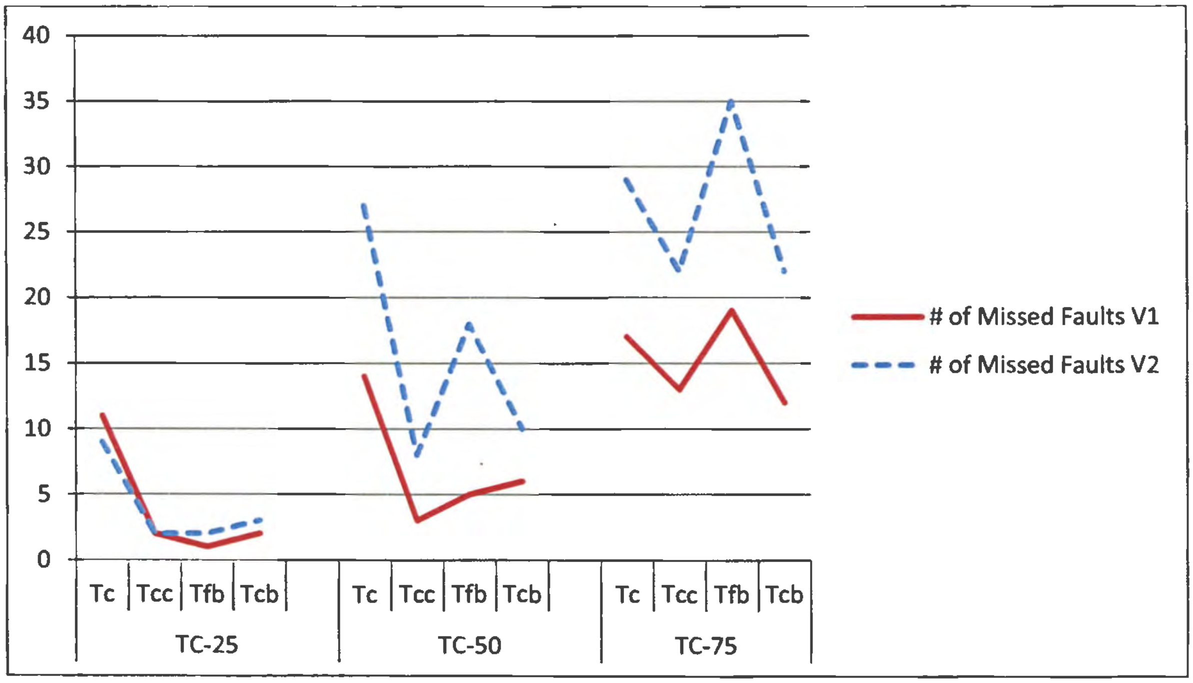


Figure 6. Missed Faults Under Time Constraints - [TC-25 = 25% tests removed; TC-50 = 50% tests removed; TC-75 = 75% tests removed; Tc = control technique; Tcc code complexity technique; Tfb = fault base technique; Tcb = combined technique]

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 complexity metrics 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.

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.

8. CONCLUSIONS AND FUTURE WORK

We have presented two empirical studies of 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.

Our studies are constrained by a few limitations that can be considered for future work. One of the constraints in these studies is the feature area of the software system. As noted in section 4.2, we only examined the financial sub-system for Microsoft Dynamics Ax product. This constraint could easily be expanded to the entire application code for Microsoft Dynamics Ax, but expanding it to include the kernel or system level API's would require additional tooling not readily available.

Second, while the prioritization heuristics showed improvement over the control, still we could improve the effectiveness of prioritization by utilizing additional information. For instance, the current code complexity based technique utilized two sets of data (the mean value of number of lines of code and dependency relationships normalized to a value between 0 and 1). 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 1, 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.

REFERENCES

1. H. Do, S. Mirarab,L. Tahvildari,G. Rothermel, *An Empirical Study of the Effect of Time Constraints on the Cost-Benefits of Regression Testing*. Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, *2008*, Atlanta, Georgia
2. Wikipedia Contributors. *Enterprise resource planning*. Retrieved Feburary 5, 2010, from http://en.wikipedia.org/wiki/Enterprise_resource_planning.
3. Microsoft Corporation (2010). *Microsoft Dynamics Ax 2009*. Retrieved February 5, 2010, from http://www.microsoft.com/dynamics/en/us/products/ax-capabilities.aspx.
4. G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu, 2004. *On test suite composition and cost-effective regression testing*. ACM Transactions on Software Engineering Methodology, Vol. 13, No 3, July 2004, Pages 277-331.
5. A. Malishevsky, G. Rothermel, S. Elbaum, *Modeling the Cost-Benefits Tradeoffs for Regression Testing Techniques*. ICSM. 2002
6. N. Nagappan, T. Ball, A. Zeller, *Mining metrics to predict component failures*. ICSE. 2006
7. H. Do, G. Rothermel, *An Empirical Study of Regression Testing Techniques Incorporating Context and Lifetime Factors and Improved Cost-Benefit Models*. SIGSOFT – Foundations of Software Engineering, 2006, pages 141–151
8. J. M. Kim, A. Porter, *A History-Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments*, ICSE 2002. pages 119-129, 2002

9. G. Rothermel, R. Untch, C. Chu, M. Harrold, M. *Test Case Prioritization: An Empirical Study*, ICSM, 1999, pages 179-188

10. T. L. Graves, M. J. Harrold, J. M. Kim, A. Porter, and G. Rothermel. *An Empirical Study of Regression Test Selection Techniques.* ACM Transactions on Software Engineering and Methodology, Vol. 10 No. 2, April 2001, pages 184–208

11. S. Elbaum, A. Malishevsky, and G. Rothermel. *Test Case Prioritization: A Family of Empirical Studies.* IEEE TSE, Vol. 28, No. 2, 2002.

12. T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, *Mining version histories to guide software changes.* ICSE 2004, pages 563–572.

13. B. Livshits, T. Zimmermann, *DynaMine: Finding Common Error Patterns by Mining Software Revision Histories.* ACM SIGSOFT Software Engineering Notes, Vol. 30 No. 5, September 2005.

14. Microsoft Corporation (2010). *Dynamics AX Sustained Engineering.* Retrieved Feb 13 2010, from http://blogs.technet.com/dynamicsaxse/archive/2009/09/30/dynamics-ax-2009-rollup-3-has-been-released-to-partner-source-and-customer-source.aspx.

15. S. Mirarab and L. Tahvildari. A prioritization approach for software test cases on Bayesian Networks. *In Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, LNCS 4422-0276, pages 276–290, 2007.

16. C. J. Bose, S. H Srinivasan, *Data Mining Approaches to Software Fault Diagnosis*, IEEE Computer Society, RIDE-SDMA 2005, pages 45-52.

17. Microsoft Corporation (2010). *XML Documentation Tags.* Retrieved Feb 14, 2010, from http://msdn.microsoft.com/en-us/library/cc607340.aspx.

18. A. Greef, M. F. Pontoppidan, and L. D. Olsen, (2006). *Inside Dynamics Ax 4.0.* Redmond: Microsoft Press.

19. A. Walcott, M. L. Soffa, G. M. Kapfhammer, and R.S. Roos. Time-aware test suite prioritization. *In Proceedings of the International Symposium on Software Testing and Analysis*, pages 1–12, 2006.

20. NIST (2010). *Software Errors Cost U.S. Economy $59.5 Billion Annually.* Retrieved March 6, 2010 from http://www.nist.gov/public_affairs/releases/n02-10.htm.

21. X. Qu, M.B. Cohen, and K.M. Woolf. *Combinatorial interaction regression testing: A study of test case generation and prioritization.* In Proceedings of the International Conference on Software Maintenance, pages 255–264, October 2007.

22. D. Leon, A. Podgurski, *A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases.* Proceedings of the 14th International Symposium on Software Reliability Engineering, 2003.

23. 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, 27(3):248–263, 2001.

24. J. Jones and M. J. Harrold. *Test suite reduction and prioritization for modified condition/decision coverage.* IEEE Transactions on Software Engingeering, 29(3):193–209, 2003.

25. G. Rothermel and M. J. Harrold. *Analyzing regression test selection techniques.* IEEE Transactions on Software Engineering, 22(8):529–551, 1996.

26. 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 Symposium on Software Testing and Analysis, pages 201–212, 2009.

27. A. Srivastava and J. Thiagarajan. *Effectively prioritizing tests in development environment.* ACM SIGSOFT Software Engineering Notes, 27(4):97–106, 2002.

28. 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., pages 111.123, June 1995.

29. D. Jeffrey and N. Gupta. *Test case prioritization using relevant slices*. In Proceedings of the Annual International Computer Software and Applications Conference, pages 411–420, 2006.

30. 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, pages 247–256, September 2008.

31. S. Hou, L. Zhang, T. Xie, and J. Sun. *Quota-constrained test case prioritization for regression testing of service-centric systems.* In Proceedings of the International Conference on Software Maintenance, pages 257–266, September 2008.

32. S. Sampath, R. Bryce, G. Viswanath, V. Kandimalla, and A. Koru. *Prioritizing user-session-based test cases for web applications testing.* In Proceedings of the International Conference on Software Testing, Verification, and Validation, pages 141–150, April 2008.