Improving the Effectiveness of Test Suite through Mining Historical Data

Jeff Anderson
Microsoft
North Dakota State U.
Fargo, ND, USA
jeffrey.r.anderson@ndsu.edu

Saeed Salem
North Dakota State U.
Fargo, ND, USA
saeed.salem@ndsu.edu

Hyunsook Do
North Dakota State U.
Fargo, ND, USA
hyunsook.do@ndsu.edu

ABSTRACT

Software regression testing is an integral part of most major software projects. As projects grow larger and the number of tests increases, performing regression testing becomes more costly. If software engineers can identify and run tests that are more likely to detect failures during regression testing, they may be able to better manage their regression testing activities. In this paper, to help identify such test cases, we developed techniques that utilizes various types of information in software repositories. To assess our techniques, we conducted an empirical study using an industrial software product, Microsoft Dynamics AX, which contains real faults. Our results show that the proposed techniques can be effective in identifying test cases that are likely to detect failures.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing & Debugging—*testing tools*

General Terms

Experimentation, Measurement, Verification

Keywords

Test failure prediction, mining software repositories, regression testing, empirical study

1. INTRODUCTION

Regression testing is one of the most common means for ensuring the quality of software products during development cycles [2]. While the benefit of running regression tests is increased quality, there is also a cost to this effort in terms of resources to run the suites and to analyze their results. For instance, we learned that running the full regression tests for Microsoft *Dynamics AX* often takes multiple days spread across a large number of computers. Testing processes by which fewer tests are run while maintaining the same ability to detect defects are therefore beneficial in reducing costs while maintaining quality, and many researchers have proposed various regression testing techniques to achieve this goal (e.g., regression test selection and test prioritization [5, 13]).

By identifying tests which are more likely to detect failures while delaying or skipping tests that are less likely to detect failures, a great deal of the cost of running regression tests can be eliminated. Alternately, the faster run time means regression tests can be run on a more frequent cadence and thereby would be more likely to catch regression faults sooner [3, 4].

While many current regression testing techniques focus on static analysis and explicit relationships among tests and product elements, we believe that there are many additional relationships among software artifacts (e.g., software code, test code and historical test results) that are not obvious or even not visible to the owners of the software artifacts. If these underlying relationships about software systems are able to be discovered, regression testing processes can be improved by selecting and running more important test cases for failure detection.

To investigate this possibility, we apply a repository mining approach to several types of data commonly found in software repositories, or available by performing calculations on the contents of those repositories. These data sets include software test coverage data, prior defect-revealing behavior related to test cases, previous test results, build history information, and smoke tests.

Using these data sets, we propose and develop two techniques. The first approach is referred to as *most common failures* in which test cases that failed the most previously are recommended as test cases that are likely to fail in the future. The second approach is referred to as *failure by association* where failures in certain subsets of tests are used to determine other subsets that are likely to fail. Both of these techniques are then paired with an examination of age of data we refer to as *windowing*. The techniques are run once using all historical data, and then again using only data from the most recent runs.

To investigate the effectiveness of the proposed techniques, we have designed and performed an empirical study using an industrial product, Microsoft *Dynamics AX* that contains real failure information. Our results show that using information about historical failures can better predict future failures. Specifically, using techniques such as frequency of failures and failure by association based on a window of recent builds outperforms the same techniques on the entire historical dataset.

The rest of the paper is organized as follows. Section 2 describes our new regression testing techniques that incorporate failure by association as well as data collection windows. Sections 3 and 4 present our experiment design, the results of the study, and data analysis. Section 5 discusses our results and their implications, and Section 6 presents related work. Finally, Section 7 presents conclusions and discusses possible future work.

MSR'14, May 31 – June 1, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2863-0/14/05...$15.00
http://dx.doi.org/10.1145/2597073.2597084

2. APPROACH

In this section, we describe the frequent failures, failure by association and windowing approaches that were examined. While we describe these approaches with Microsoft *Dynamics AX*, the same approaches could be applied to any system with a similar test infrastructure and tracking system.

The data used in this study is the result of all development done during the *Dynamics AX 2012 R2* release. This was a major release encompassing over a year of development. During the development phase, check-ins of source code to the source control system occur at random intervals, 24 hours a day, 365 days a year. While they may cluster more sparsely or heavily on certain days, there is little cadence to them as shown in Figure 1.

Builds of the source control occur on a regular nightly cadence, regardless of how many check-ins have occurred since the previous build. While not all builds are successful, the vast majority are usable, deployed, and used on a regular basis by developers.

As shown in Figure 1, the full regression test suite is not executed on each nightly build due to the time the regression test suite takes to execute. Approximately 100 test machines are utilized for a regression test run, and the full run takes approximately 3 days to complete. Thus, full regression test runs are executed approximately weekly, although that cadence may shift forward or backward by a few days based on build quality, holiday schedules, infrastructure or other issues.

The goal of this research is to select effective regression tests that are likely to detect failures and to run them, giving development extra time to fix any issues discovered while the remaining tests are run. If a more useful set of tests runs earlier, problems found by such tests can be fixed up to two days sooner based on a three day test run.

Figure 2 demonstrates the mechanism by which test case selection occurs. At the start of any test run, such as the one shown on 2/4/2013 of Figure 1, the test cases for that regression test run are selected. This is accomplished by running a selection algorithm which reads from all test artifacts as well as historical test results from previous builds. The output of that algorithm is a set of tests deemed more likely to fail in the current regression test run. That set of tests runs first on the pool of test computers. Once execution of the selected tests completes and the test machines are free again, then the remaining tests are executed.

A final note in Figure 2 is that a small random set of tests out of the pool of all tests are pre-executed as part of the build process. These are referred to as "smoke tests" and are used to help determine the general quality of the build and whether or not it can be used for a full regression test run. The selection algorithm may also consider the results of those smoke tests in determining the selected tests deemed most likely to fail for that test run.

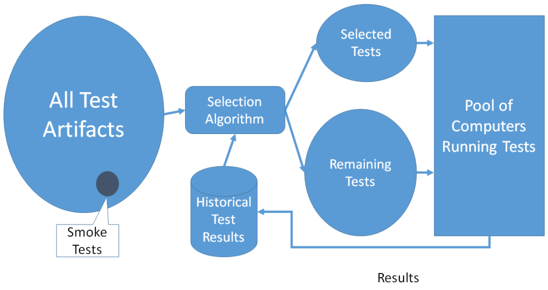


Figure 2: Test Selection Process

This study can be described abstractly as follows. Consider a set of n regression tests labeled T_1, T_2 through T_n. We define A_i to be the set of all tests runs in the i^{th} regression suite run:

$$A_i = \{T_{i1}, T_{i2}, ...T_{in}\}$$

The union of all tests from all test runs is defined as A. Note that A differs from A_i since a given test may or may not be run in a given regression test run due to infrastructure issues, time pressure, active development causing the test to be temporarily disabled, or various other reasons.

Let F_i be the set of all failed tests in a given regression test run. So if m failures existed in regression test run i, we would define:

$$F_i = \{T_{i1}, T_{i2}, ...T_{im}\}$$

Only tests which were run in a given regression test run may actually fail, i.e., $F_i \subseteq A_i$.

The goal of this study is to predict F_i as accurately as possible for a given regression test run. We will therefore define the set of k predicted failures for regression test run i as P_i such that:

$$P_i = \{T_{i1}, T_{i2}, ...T_{ik}\}$$

Only tests that have failed in previous regression test runs can be predicted to fail in the current test run. So, we have that:

$$P_i \subseteq \bigcup_{j=1}^{i-1} F_j$$

There is however no guarantee that P_i and F_i overlap in any way. If we were able to perfectly predict the failures in regression test run i, then we would have a case where $P_i = F_i$, but this perfect prediction is likely not possible. We can define the set of correctly predicted test failures C_i as the intersection between the predicted tests and the test that actually failed, i.e., $C_i = P_i \cap F_i$.

Increasing the set of correctly predicted tests is important, as it will increase the bug finding abilities of regression test run i. However, it should be noted that by increasing the size of P_i, the size of C_i can also be increased, to the point where if $P_i = A_i$, then we would have $C_i = F_i$. While all failed tests would have been correctly predicted, this is not a good situation, since $|P_i|$ would be significantly larger than $|F_i|$. These extra predicted failures that did not occur would require additional test resources to run, but would not find any bugs. Therefore what we actually want to do is maximize $|C_i|$ while minimizing $|P_i|$.

We measure these competing interests through the use of two aspects, precision and recall. Precision is a measure of the how well we were able to predict the tests that have actually failed, expressed as the ratio of correctly predicted failures to the number of predictions made. More formally, we define:

$$Precision = \frac{|P_i \cap F_i|}{|P_i|} = \frac{|C_i|}{|P_i|}$$

Recall on the other hand measures how well we were able to cover all the failed tests with our prediction. It is the ratio of correctly predicted failures in the regression test run. This can be defined as:

$$Recall = \frac{|P_i \cap F_i|}{|F_i|} = \frac{|C_i|}{|F_i|}$$

With these two measures we have a good indication as to how well a given heuristic meets the goals. But these two measures need to somehow be combined as increasing one measure will often

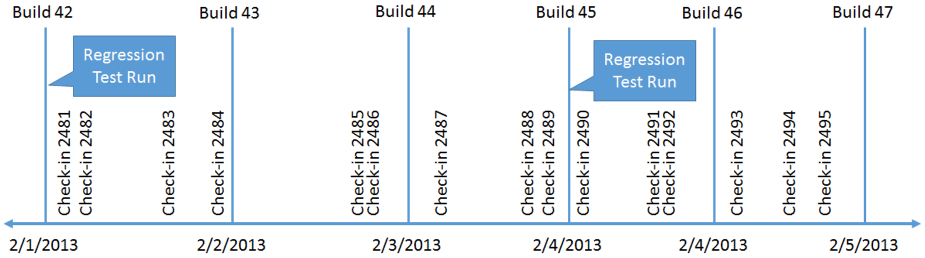


Figure 1: Build and Test Cadence

decrease the other. For that we compute the harmonic mean of the two values, often referred to as F-measure.

$$F - measure = \frac{2 \times Precision \times Recall}{(Precision + Recall)}$$

The values of these measures range between 0 and 1.

Figure 3 depicts a concrete example of the method in which the empirical study was performed. This diagram presents the full historical results of which tests passed and failed in regression test runs 1 through 9. As an example, we will consider performing the selection algorithm to predict the tests that are likely to fail in test run number 9. This prediction will be based on the historic results from builds 1 through 8, together with the smoke tests of T_2 and T_{10} which were pre-executed on test run 9. Those predicted failures will then be compared against the actual failures seen in the historic results for rest run 9 and analyzed.

Note that Figure 3 shows both passes and fails of tests as being part of the data set. In this study, only failures are examined, as failed tests are what is being predicted. So for example, the actual dataset for test run number 4 would only contain 4 failed tests, $\{T_4, T_5, T_7, T_{14}\}$. These represent the failed tests in test run number 4 and would be expressed as F_4. The rest of the tests are only shown in this diagram to aid in understanding.

Once the set of failures has been predicted, we analyze the quality of the prediction based on precision, recall and F-measure. For example, consider a case where we predict for test run number 9 that 8 test cases would fail. Of those, 3 actually did fail. And two additional tests that actually failed were not predicted to fail.

As previously defined, precision is the percentage of the predicted failure that actually did fail. So in this example, the precision would be $3/8 = 0.375$. As the total percentage of all actual failures that were predicted, recall in this example would be $3/5 = 0.600$. Combining precision and recall, the F-measure is 0.462.

2.1 Most Frequent Failures

We refer to the first algorithm for predicting failures as *Most Frequent Failures*. In this approach, a threshold is used, and any test which failed at least as many times as the threshold in previous test runs is predicted to be likely to fail again.

As an example, consider using this approach with a threshold of 50 percent using the results depicted in Figure 3 to predict failures in test run number 9. Since there are 8 previous runs any test that fails 4 or more times in test runs 1 through 8 exceeds the threshold and will be predicted to be likely to fail. Based on that prediction it would be selected above other test cases.

In the case of Figure 3, test T_1 failed 5 times in test runs 1 through 8, and therefore exceeds the threshold and is selected. Similarly, T_5, T_6, T_{10}, and T_{14} are also selected. Using our previous notation, we have:

$$P_9 = \{T_1, T_5, T_6, T_{10}, T_{14}\}$$

These are then compared with the 5 actual failures in build 9, $F_9 = \{T_1, T_3, T_{10}, T_{13}, T_{14}\}$.

Out of the 5 predicted failures, 3 were correctly predicted:

$$C_9 = P_9 \cap F_9 = \{T_1, T_{10}, T_{14}\}$$

This yields a precision of $|C_9|/|P_9| = 3/5 = 0.600$, recall of $|C_9|/|F_9| = 3/5 = 0.600$, and F-measure of 0.600.

2.2 Failure by Association

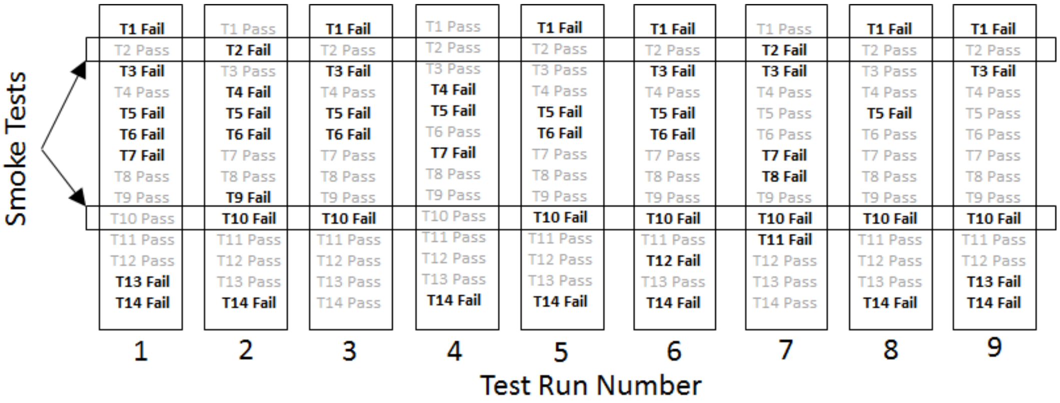
The second algorithm used in predicting likely failures is referred to as *Failure by Association*. In this technique, we use concepts from association rule mining to predict failures. Association rule mining is a technique by which a database of historical transactions is analyzed, and a set of rules are determined which indicates associations between items in the transactions [1]. Association rules mined from previous transactions can be utilized for predicting future associations. For simplicity we often refer to failure by association by the acronym ARM, since it relies on association rule mining concepts.

Our failure by association approach is similar to association rule mining, with one major difference. Instead of determining the rules ahead of time, we run the rule mining algorithms separately for each test run being analyzed. We did this for two reasons. First, the number of transactions in our system is relatively small, being based on only 64 total regression test runs. So if a static database were used such as starting at halfway through the project, this greatly reduces both the number of test runs we can analyze as well as the number of transactions being considered in the rules. Also importantly, association rule mining techniques are very computationally intensive. The running time of association rule mining mainly depends on the number of items in the dataset and the minimum support and confidence. The number of tests cases which correspond to the number of items in association rule mining is over 65,000 test cases to analyze.

In *Failure by Association*, relationships between test failures are found and then used to predict failures in a given regression test run. Unlike the most frequent failures approach in the previous section, the *Failure by Association* approach requires some information from the current regression test run in addition to historical data. Fortunately, this information is available in the form of smoke tests. As described by Kaner et all [7], smoke tests are a small sample of tests that are run after each build to determine whether or not the build is high enough quality to continue running additional tests. We define the set of failed smoke tests in regression test run i as the set of regression tests which were run in i, but failed. We define S_i as:

$$S_i = \{T_{i1}, T_{i2}, ...T_{ir}\}$$

The association rules are of the form $LHS \implies RHS$. The support of a rule is the number of transactions in which the left hand side items appear along the right high side items. Support



Smoke Tests

T1 Fail	T1 Pass	T1 Fail	T1 Pass	T1 Fail	T1 Fail	T1 Pass	T1 Fail	T1 Fail
T2 Pass	T2 Fail	T2 Pass	T2 Pass	T2 Pass	T2 Pass	T2 Fail	T2 Pass	T2 Pass
T3 Fail	T3 Pass	T3 Fail	T3 Pass	T3 Pass	T3 Fail	T3 Fail	T3 Pass	T3 Fail
T4 Pass	T4 Fail	T4 Pass	T4 Fail	T4 Pass	T4 Pass	T4 Pass	T4 Pass	T4 Pass
T5 Fail	T5 Fail	T5 Fail	T5 Fail	T5 Fail	T5 Fail	T5 Pass	T5 Fail	T5 Pass
T6 Fail	T6 Fail	T6 Fail	T6 Pass	T6 Fail	T6 Fail	T6 Pass	T6 Pass	T6 Pass
T7 Fail	T7 Pass	T7 Pass	T7 Fail	T7 Pass	T7 Pass	T7 Fail	T7 Pass	T7 Pass
T8 Pass	T8 Pass	T8 Pass	T8 Pass	T8 Pass	T8 Pass	T8 Fail	T8 Pass	T8 Pass
T9 Pass	T9 Fail	T9 Pass	T9 Pass	T9 Pass	T9 Pass	T9 Pass	T9 Pass	T9 Pass
T10 Pass	T10 Fail	T10 Fail	T10 Pass	T10 Fail	T10 Fail	T10 Fail	T10 Fail	T10 Fail
T11 Pass	T11 Pass	T11 Pass	T11 Pass	T11 Pass	T11 Pass	T11 Fail	T11 Pass	T11 Pass
T12 Pass	T12 Pass	T12 Pass	T12 Pass	T12 Pass	T12 Fail	T12 Pass	T12 Pass	T12 Pass
T13 Fail	T13 Pass	T13 Pass	T13 Pass	T13 Pass	T13 Pass	T13 Pass	T13 Pass	T13 Fail
T14 Fail	T14 Fail	T14 Pass	T14 Fail	T14 Fail	T14 Fail	T14 Pass	T14 Fail	T14 Fail
1	2	3	4	5	6	7	8	9

Figure 3: Sample data

is used to filter out weak rules that show the association in a very small number of transactions. The confidence of a rule is the ratio of the number of times the right hand side items appear when the left side items appear. Confidence is the likelihood that the right hand side items appear given that the left hand side items appear. Confidence is used to filter out rules where there is not a strong correlation between the left and right hand side items appearing. For a given dataset and support and confidence threshold, a set of 'interesting' rules is mined, such that:

$$Rules = \{(LHS_1 \implies RHS_1), ...(LHS_x \implies RHS_x)\}$$

The LHS is the left hand side of each rule, also referred to as the antecedent. Given the failed tests in previous regression test runs, we can mine all the rules whose support and confidence exceed user-specified thresholds. However, we only have a small set of tests for the regression test run for which we are trying to predict the tests that are likely to fail. Therefore, we are not interested in all the rules that are valid in the previous test runs. We are only interested in the rules in which the LHS tests are part of the smoke tests, i.e., $LHS_j \subseteq S_i$.

To mine only the sought-after rules which are applicable for the current test run, we propose a different method for mining these rules. For a given support threshold, $minsup$, we mine the frequent tests in previous test runs. Only these frequent tests can be on the left hand side of any interesting rules since by definition the tests in an interesting rule have to appear in at least $minsup$ transactions. The transactions that have at least one of these frequent tests are retained and other transactions are pruned.

Using Figure 3 as an example, let us consider using a support of 50 percent. Since we are predicting test failures for regression test run 9, the first step is to find all itemsets which occur in test runs 1 through 8 more than 50 percent of the time, i.e., 4 or more runs. So any test that occurs 4 or more times is considered to be frequent. Since the LHS consists only of tests from the smoke tests S, only T_2 and T_{10} need to be examined as being part of the LHS.

Based on this simplified example with only two smoke tests (T_2, T_{10}), only T_{10} is frequent (appears in test runs $2, 3, 5, 6, 7$ and 8) and thus it is the only smoke test that can appear in the LHS of any interesting rule.

The second step is to mine for associations between tests and the frequent smoke tests in these transactions. Once all the antecedents have been determined based on support, the next step is to find the associated RHS or right hand side of the rule, also referred to as the consequent. Similar to how a support level is used in determining the frequent LHS itemsets, a threshold called the *confidence* is used to determine which itemsets occur in the RHS. Unlike the LHS which examines all previous test runs looking for when that test occurs, the RHS only looks at test runs in which the LHS also occurs.

Continuing the example based on Figure 3, consider a confidence of 50 percent. Remember we have a single LHS, so $LHS_9 = \{T_{10}\}$. In this case, $|LHSRuns_9| = 6$, meaning there are six previous test runs containing the $LHSRuns_9 = \{2, 3, 5, 6, 7, 8\}$. So a confidence of 50 percent means any itemsets from the test failures in these test runs occurring more than $6 \times 0.50 = 3$ times are considered frequent.

In this example, the frequent tests, that could be part of the RHS of interesting rules, are $\{T_1, T_3, T_5, T_6, T_{14}\}$. Each of these tests appeared with the left hand side in at least 3 test runs. With these tests we can generate multiple rules with different tests in the RHS. Unlike with traditional association rule mining though, we do not need all combinations of items to occur in the RHS of the rules, since all items in any RHS will be predicted as test failures. Therefore we do not need to examine the RHS items in combinations, rather we can just examine the number of failures of each test individually to determine if it meets the minimum confidence level. The union of all these tests which meet the confidence level is then our predicted set of failures.

With a single LHS in this example, the set of rules is:

$$\{T_{10} \implies T_1, T_{10} \implies T_3, T_{10} \implies T_5, ...T_6, T_{14}\}$$

This means that for the failure by association approach for test run 9:

$$P_9 = \{T_1, T_3, T_5, T_6, T_{14}\}$$

As described earlier, the actual failures for run excluding the smoke tests T_2 and T_{10} are:

$$F_9 = \{T_1, T_3, T_{13}, T_{14}\}$$

And the set of correctly predicted failures:

$$C_9 = P_9 \cap F_9 = \{T_1, T_3, T_{14}\}$$

So applying the same analysis for precision, recall and F-measure, this yields a precision of $|C_9|/|P_9| = 3/5 = 0.6$, recall of $|C_9|/|F_9| = 3/4 = 0.75$, and F-measure of 0.667.

2.3 Test Age

The final approach examined in this research is the applicability of test age on the accuracy of the predictions. The age of a test is the number of test runs that have occurred since that test run until now. We define a window of data to be the set of all test runs with $age \leq window$.

This approach is accomplished by examining only a window of previous data, rather than all previous data when applying the most frequent failures and failure by association approaches. Consider the example from Figure 3 while predicting failures for test run number 9. If a window size of 5 is used, then instead of using test runs 1 through 8 as historic data, only the 5 most recent runs, 4 through 8, would be examined.

Applying this approach to the most frequent failures example previously described leads to different results. The 50 percent threshold for frequent failures is now applied to 5 runs, meaning a failure only needs to occur 3 times instead of 4 to be considered frequent. Similarly, only runs 4 through 8 are examined for these failures. In these runs, T_1 fails 3 times and is frequent. Similarly T_5, T_{10} and T_{14} each fail three or more times in runs 4 through 8. These tests are then predicted to fail.

Of the 4 predicted failures and 5 actual failures, 3 were correctly predicted. This yields a precision of $3/4 = 0.75$, recall of $3/5 = 0.6$, and F-measure of 0.667

Note this resulting F-measure of 0.667 is slightly better than the F-measure of 0.600 found when examining all previous test runs. The same approach is also applied to the failure by association when mining mining for applicable rules..

3. EMPIRICAL STUDY

As stated in Section 1, in this research, we investigate whether the use of failure by association, most frequent failures, and test result age can help better predict the likely test failures in a given test run. These methods are applied to an industrial product. This section describes the empirical study performed.

In our study, we investigate the following research questions:

RQ1: Can learning from previous test runs improve the effectiveness of selecting tests in terms of fault prediction?

RQ2: Can restricting the set of previous test runs based on age help increase the effectiveness of selecting tests in terms of fault prediction?

3.1 Object of Analysis

We used the Microsoft *Dynamics AX 2012 R2* 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) [10]. 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.

Table 1: Dynamics AX R2 Product

Lines of Product Code	5,584,753
Lines of Metadata Code	3,986,849
Lines of Regression Test Code	4,821,215
Regression Test Computers	100
Number of Regression Tests	65,512
Code Check-ins	6667
Product builds	340
Test runs	64
Distinct test results	3,444,634
Total failed tests	309,697

Table 1 describes the attributes of the product which was studied in this research. As previously mentioned, the product contains millions of lines of code, written in a proprietary language called X++. This language allows much of the business logic to be expressed as metadata, such as table definitions, queries, and user interface constructs. As such, the lines of metadata code is also an interesting metric, as metadata can have bugs similar to other product code. In the case of this product, the metadata is similar in size to the X++ code in the product. Metadata code is product code expressed in ways other than a traditional programming language. This includes XML files which define form and table structures as well as macros and other elements.

There are many regression tests for the product, again involving nearly as many lines of code as the core X++ code in the product. These 65,512 tests cannot be run on every check-in or build, as previously discussed. They take multiple days to run, and therefore tests are run on a regular cadence, generally once every few builds. The R2 product release contained 340 nightly builds, usually one each night. Of these builds, 64 of them had a regression test suite run, or about 18 percent of the builds. Each of these regression tests runs were scaled out across approximately 100 physical computers during the run, with the results aggregated over the multi-day run.

Throughout the release, a total of 3,444,634 distinct test results were captured, of which 309,697 were failures. That number represents an average of 9 percent of tests failing in any given regression test run. Note that not all 65,512 regression tests were run every time. Occasionally a failure of a computer or other test infrastructure would cause a test not to run. Sometimes, a set of tests may be temporarily disabled during active development. Other tests were added or removed throughout the release. On average approximately 53,822 of the 65,512 were executed each time though, representing 82 percent of the total number of regression tests. For this research, any test that was not executed in a given test run was ignored and not included in the analysis.

Test code refers to the code used to create the regression tests themselves. Since there are 65,512 regression tests and 4,821,215 lines of regression test code, this indicates each regression test averages around 74 lines in length.

3.2 Variables and Measures

3.2.1 Independent Variables

This study manipulated two independent variables: selection technique and test result age window. We consider two *control* technique and four *heuristic* techniques as follows:

- Control: A random set of tests is selected out of the set of previously failing tests and used as the prediction. When recommending the tests in the control method, the number of recommended failures needs to be determined as well. In this study, we predicted the same number of failures as the average number of failures seen in previous test runs.
  - $Trandom, full$: This technique randomly selects a set of tests from the full set of previous failure data.
  - $Trandom, recent$: This technique randomly selects a set of tests from the most recent previous failure data.
- Heuristics: We consider four heuristics representing the different combinations of failure by association and data recency.
  - $Tfrequent, full$: This technique predicts failures based on most frequent previous failures, examining the full set of previous build data.
  - $Tfrequent, recent$: This technique predicts failures based on most frequent previous failures, examining only the most recent previous test runs.
  - $Tarm, full$: This technique predicts failures based on failure by association of previous failures, examining the full set of previous build data.

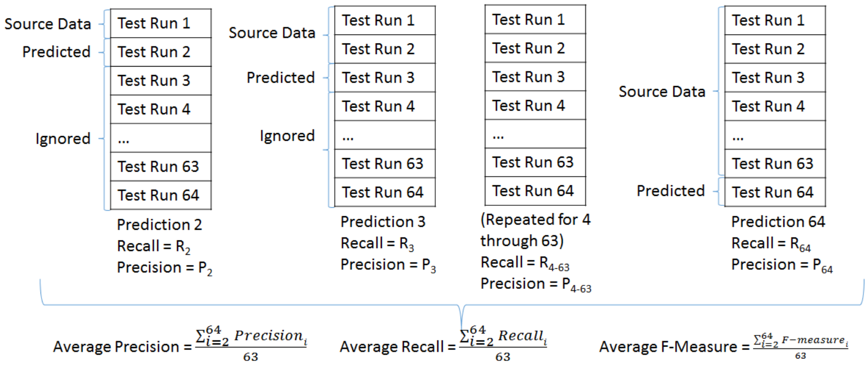


Figure 4: Experiment Process

- $Tarm, recent$: This technique predicts failures based on failure by association of previous failures, examining only the most recent previous test runs.

3.2.2 *Dependent Variables*

We consider two dependent variables, the precision and recall of the predicted test failures. Precision refers to the percentage of predicted failures that actually failed. Recall refers to the percentage of actual failures that were predicted. To measure the overall effectiveness across both precision and recall, we also compute the F-measure which is the harmonic mean of the precision and recall.

The formulas and more complete explanations of precision and recall are presented in Section 2.

3.3 Experiment Process

This experiment was performed on historical data from the *Dynamics AX 2012 R2* release. As previously mentioned, this release contained 64 regression test runs. The techniques being studied were applied to each test run, simulating what would have happened if they had been applied during the actual development cycle.

As shown in the first box of Figure 4, we started with regression test run 2 since the first regression test run had no historical data which could be used for prediction. At the time regression test run two was about to begin, only regression test run 1 had previously happened, so only data from regression test 1 could be used for prediction. Based on that data, we predicted the failures for regression test run 2. Once those failures were predicted, we then compared them with the actual results for regression test run 2 and calculated the precision and recall for that regression test run, P_2 and R_2.

We then applied the same process to examine regression test run 3 (the second box in Figure 4). Again, simulating the data available at the time regression test run 3 was about to begin, we only had historical data from regression test runs 1 and 2. So that data was used to predict the failures in test run 3. We then compared those predicted failures from test run 3 against the actual failures in test run 3 and again calculated precision and recall, P_3 and R_3.

Following this same process, the failures for test run 4 were predicted based on the results of runs 1 through 3 and used to calculate P_4 and R_4. Test run 5 was predicted based on results of runs 1 through 4, yielding P_5 and R_5. This continued all the way up to predicting the failures in test run 64 based on the results of test runs 1 through 63, yielding P_{64} and R_{64} (the rightmost box in Figure 4).

The only difference in this technique when applying test age to examine only a window of data was only the most recent test runs were examined. So with a window size of 10, the predictions for test run 37 would be based on the results of test runs 27 through 36.

Once precision and recall had been calculated for each test run using each technique being studied, the results were then averaged across the 63 results for each technique.Using the average precision and recall for each technique, we then calculated the F-measure of the technique. (See the equations below.) Averaging the precision, recall, and F-measure across all of the test runs is important, as there is high variability in the precision and recall values found in each test run across the data set. Based on the average values, we can determine if one technique is more effective than another in general, ignoring local variability.

$$AveragePrecision = \frac{\sum_{i=2}^{n} \frac{|P_i \cap F_i|}{|P_i|}}{n-1}$$

$$AverageRecall = \frac{\sum_{i=2}^{n} \frac{|P_i \cap F_i|}{|F_i|}}{n-1}$$

$$AverageF-Measure = \frac{\sum_{i=2}^{n} F-measure_i}{n-1}$$

4. DATA AND ANALYSIS

In this section we will discuss the result based on the research questions presented in Section 3.

4.1 Failure by Association and Frequent Failures

The first research question (RQ1) addressed was whether or not learning from previous test runs can improve the effectiveness of test case selection in future runs. Figure 5 shows precision, recall and F-measure for the control and heuristic techniques (failure by association (ARM) and frequent failures). Both the frequent failures and ARM techniques are much more effective at predicting failures than the control technique, the control having an F-measure of 0.179 and the frequent and ARM approaches at 0.441 and 0.460 respectively.

It is also of note that while the ARM approach yielded a better result than frequent failures, the difference was negligible. Overall, it was only 0.04 more effective based on F-measure. Based on the individual analysis of the association rules, it was found that very few rules contained more than a single test on the RHS. This means that the confidence measure in failure by association becomes very similar to the confidence measure in the frequent failures approach for determining predicted tests, leading to very similar prediction effectiveness. In systems where there is higher coupling between tests, the size of the RHS in the rules would be expected to increase, and therefore the effectiveness of the ARM approach should also increase compared to that of the frequent failures approach.

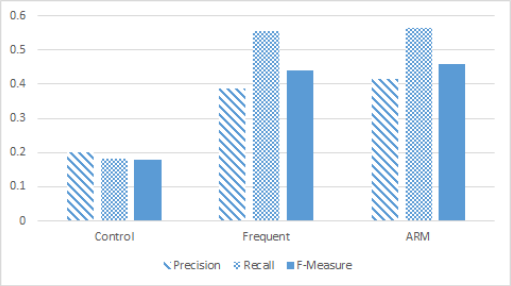


Figure 5: Effectiveness of ARM and Frequent Failures

4.2 Test Age and Analysis Windows

The second research question (RQ2) was whether restricting the set of previous test runs based on age help increase the effectiveness of the techniques we investigated in RQ1.

Figure 6 shows precision, recall, and F-measure across all three approaches both with and without restricting the set of previous test runs. It is clear from this data set that using recent test runs as opposed to all historical test runs greatly increases the effectiveness of fault prediction of the identified tests. This even applies in the case of the control, which randomly selected tests from any that had previously failed.

The cost of applying this windowed approach is also negligible. Unlike approaches such as failure by association which required several data processing steps to make predictions, applying a window based on age actually reduces the amount of data to be processed. The fact that reducing the amount of data sets to be processed increases the effectiveness of the approach is important because it means this is a very valid technique that can be applied with no additional cost to other methods.

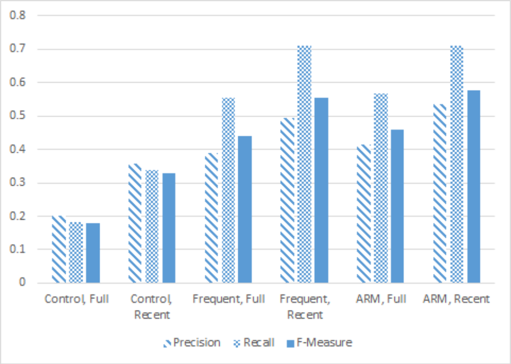


Figure 6: Overall effectiveness of prediction approach

4.3 Threshold Derivation

The specific support and confidence levels used in this analysis were determined by evaluating a wide range of support and confidence levels and then picking the values that yielded the highest F-measure in each case.

Table 2 shows the precision, recall and F-measure values for several different support and confidence levels. As shown in the table, the highest F-measure occurs at a support level of 0.1 and a confidence level of 0.25. As expected, the lower the support and confidence levels, the more tests were predicted, increasing recall. But at the same time these additional lower quality predicted failures decreased precision leading to an overall decrease in F-measure.

Similarly, Figure 7 shows the values for the frequent failures approach at various support levels. As shown in the figure, the highest F-measure occurs at a support level of 0.2. If the support level is dropped below 0.2, then additional tests are predicted to fail which do not fail. This leads to a low precision level which drops F-measure. As the precision increases due to the higher support level, the recall drops, leading to a lower F-measure. Due to this inverse relationship between precision and recall based on support level, we find that the highest F-measure occurs at a support level of 0.2, so that is what was used for the research.

Table 2: Threshold Derivation of Failure by Association

Support	Confidence	Precision	Recall	F-Measure
0.100	0.100	0.412	0.729	0.480
0.100	0.250	0.463	0.701	0.515
0.100	0.500	0.318	0.766	0.418
0.200	0.100	0.389	0.743	0.466
0.200	0.250	0.367	0.762	0.456
0.200	0.500	0.371	0.644	0.450
0.400	0.100	0.379	0.696	0.446
0.400	0.250	0.397	0.627	0.467

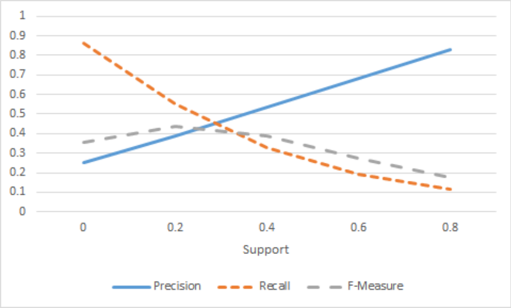


Figure 7: Threshold Derivation of Frequent Failures

The size of the sliding window was also determined empirically as shown in Figures 8 and 9. These figures show the precision, recall and F-measure for both ARM and frequent failures approaches using a variety of window sizes from 15 to 50 previous test runs, as well as using all historical test run data.

By examining these figures, we can see that the F-measure is best for both approaches at a window size of approximately 25 previous runs. We use F-measure as the variable we examine, as both precision and recall can be artificially increased by selecting all or virtually no tests, neither of which is helpful. Instead, we wish to improve both precision and recall at the same time, which can be measured by F-measure.

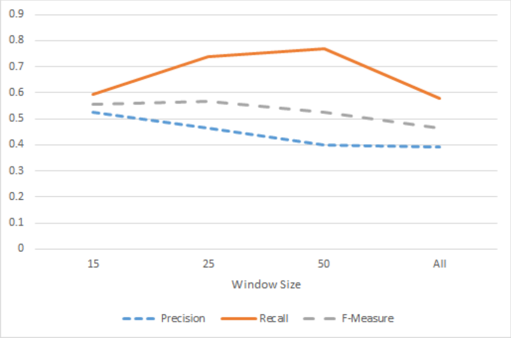


Figure 8: Window Derivation for ARM

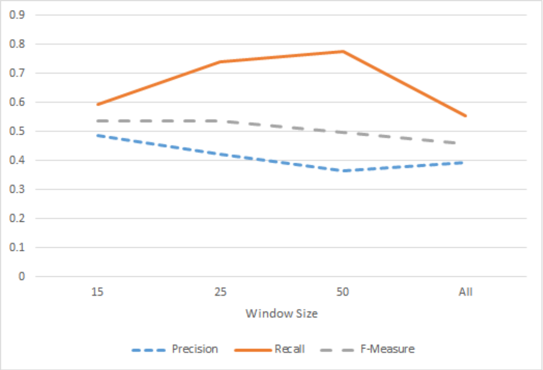


Figure 9: Window Derivation for Frequent Failures

4.4 Control Prediction Size

As previously discussed, the number of recommended failures for the control set had to be determined, and the average number of previous failures were used. Because the best support, confidence and window sizes were used for the heuristics, it could be argued that the optimum number of recommendations should also have been used for the control set. Figure 10 shows precision, recall and F-measure using both the average number of previous failures, as well as the results by predicting all previous failures. The different dataset represented in the figure are as follows:

- 25, Avg is using a window size of 25 and the average number of failures from previous runs
- 25, All is using a window size of 25 and predicting all previous failures
- Infinite, Avg is using all historical results and predicting the average number of failures from all previous runs
- Infinite, All is using a all historical results and predicting all tests that have ever failed

Examining the figure, it is notable that the number of recommendations does not impact the precision of the prediction, as any test that is recommended has an equal chance of matching a failure. The only variation is in recall, where more predictions will obviously yield a higher recall, and higher F-measure. Therefore, it follows that the best control size would be to recommend all previous failures. But this is effectively the same as predicting the most frequent previous failures with a support level of 0.0. As discussed above, the frequent technique yields the highest F-measure at a support of 0.2. So even with the most optimal recommendation size for the control, it is still outperformed by the frequent technique.

As the goal of this research is to reduce the number of tests which need to be run, recommending all tests by default is also not beneficial. For those reasons, a control prediction size equaling the average number of previous failures was used.

5. DISCUSSION AND IMPLICATIONS

Our results indicate that the effectiveness of test case failure prediction can be improved through the use of historical test results, together with an application of a concept of test result age. As shown in Figures 5 and 6, the use of frequent failures and failure by association produced better precision and recall values (between 0.38 and 0.56) compared to the control technique (less than 0.20) This essentially doubles the effectiveness of prediction. Similarly, applying a concept of test age to use only the 25 most recent test results showed increases in the effectiveness of not only the frequent failures and failure by association approaches, but also in increasing the effectiveness of the control approach.

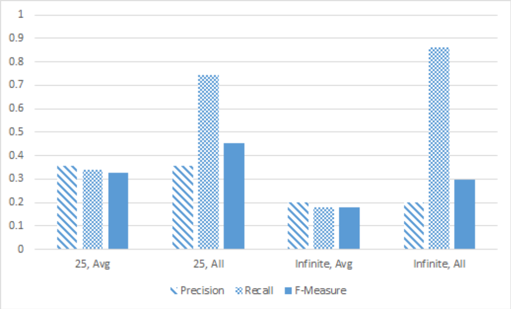


Figure 10: Control Result Size by Suggestion Volume

While the overall results show the effectiveness of the proposed approaches, there are additional observations and implications, and we discuss these in the subsections.

5.1 Trading Precision for Recall

An important discussion related to this analysis is the trade-off between precision and recall, the two measures that were used to examine the effectiveness of the approaches. Because recall is only a measure of the ratio of the failures that were correctly predicted, recall can easily be increased all the way to 1.0 by just predicting more and more failures until all potential failures are predicted. Similarly, precision can be increased artificially by not predicting any failures in cases where there is any doubt. In industrial practice though, neither of these extremes is beneficial. When too few tests are run, the implication is obviously that some failures will not be caught, and the quality of the product will suffer.

The more difficult trade-off to justify is decreasing the number of tests run when those tests may have found bugs. This would be the case if recall were artificially increased by running all applicable tests. It must be remembered that development time and hardware budgets are finite resources in an industrial product release. This means that additional processing and analysis time spent on extra test runs which do *not* increase quality are taking those resources away from other applications. Those other applications may be performance analysis, tours testing, ad-hoc testing, or a variety of other engineering practices. The challenge then is in balancing the precision and recall to catch as many bugs as possible while not stealing resources that could better be utilized elsewhere.

In the case of our analysis, the primary driver of the result set size is the support and confidence thresholds described in Section 2. As shown in Table 2, a variety of values were used ranging from extremely small to extremely large values. As would be expected, lower support and confidence thresholds predicted more failures and therefore increased recall. Similarly, higher confidence thresholds increased precision by only predicting tests more likely to fail.

While we picked the levels that showed the best F-measure in this research, it is important to note that virtually *all* levels of support and confidence yielded better precision and recall than the control. The same is true for the frequent failures approach as shown in Figure 7 for all values of support below 0.5. The fact that F-measure was higher with ARM and frequent failure approaches than with the control approach at a range of support and confidence thresholds means the support and confidence thresholds can be further tuned for a given development organization to increase or decrease the number of tests predicted based on available resources and what other quality projects could otherwise be funded.

5.2 Test Case Age

Another interesting aspect of this research is the shape of the curves depicted in Figures 8 and 9. Note in both of these cases that the F-measure increases with a smaller window size compared to using all results, but then decreases again as the window size becomes smaller yet.

We speculate that these curved variations in the F-measure based on test result age are resulted from the fact that different areas of the product have been developed at different times throughout the development cycle. Suppose we have a system would go through a number of code changes. During the phase of active development and stabilization, a given test failure in one test run is likely to be followed by a similar failure in a subsequent run if the area impacting that test is still unstable. Similarly, fixing bugs found by tests in an area will cause code churn and make other failures in that area more likely until it is done stabilizing.

At the same time, other areas of the product are not being worked on as actively and therefore do not exhibit the same instability. Over time as one area is stabilized and the development team moves on to another area, the stability moves accordingly. This is why we believe that more recent test results are better predictors of failure than older ones. Once that window becomes too small, the number of previous results being examined decreases markedly. At some point, not enough data is available to make good predictions any more.

Further, we believe the size of the window will likely differ from product to product. Various factors, such as the length of development cycles, the stabilization time frames, and even the development methodologies used by the team may impact the choice of the window size.

5.3 Cost of Implementation

Implicit in this research but not yet discussed is the fact that performing analysis such as failure by association or frequent failures to predict test failures is an engineering activity which in itself has a cost. As discussed earlier in this section, spending cost on one quality activity in product development necessarily takes away time and resources that could have been spent on other, potentially more important activities. The cost of performing the research in this paper was not explicitly captured, so it will not be discussed quantitatively. A qualitative discussion is still helpful however.

The two primary techniques used for predicting based on previous failures were the failure by association and frequent failures approaches. Between these two approaches, failure by association demonstrated approximately a four percent benefit over that of frequent failures as shown in Figure 6. This does not necessarily mean that failure by association is a better approach. One challenging task is how to efficiently mine the applicable association rules for every test run. This is especially significant considering the fact that most frequent itemset mining algorithms (e.g., Apriori [1], ECLAT [14], FP-growth [6]) enumerate the entire extremely large frequent itemsets search tree.

This means that as the number of items in the itemsets increases, the cost of running these algorithms increases dramatically. In our case, it was only after about a large amount of pruning and performance tuning that the data and approach of failure by association were efficient enough to complete in a less than a few days. One such performance tuning activity was searching only within test runs which contained the same failures as occurred in the smoke tests.

Agrawal and Srikant [1] discuss many other association rule mining algorithms, many of which are more efficient than Apriori. But compared to performing analysis of most frequent failures, all of these algorithms are significantly more complex to write, as well as significantly more costly to execute on a computer. The most frequent failures approach took only around a day to write and less than an hour to execute.

Based on this experience and the fact that failure by association only yielded an improvement of around four percent in effectiveness, we recommend applying a most frequent failures approach in industrial application. The additional resource savings by using the frequent failures approach can be applied to other quality activities during the software release cycle.

5.4 Threats to Validity

In this section, we discuss the threats to validity of our study. The study was performed on the Microsoft *Dynamics AX* 2012 R2 release. As this is a single release of a single product, the results seen in this study will not necessarily translate to all other products or releases. This particular release involved the significant use of software branches, meaning that development of different portions of development were occurring separately. The branching was not controlled for in the results, and therefore may have had an impact on the results of the study since branch integrations are times are artificially large code change in the branch from which the tests are run.

This study focuses on the relationships between different tests and their propensity to fail. Since the amount of coupling or other inter-test relationships is purely based on the actions of the engineers building those tests, other products built by other companies and other groups of developers may exhibit stronger or weaker relationships than those observed in this study.

Similarly, the release cycle for this product was approximately one year. As discussed earlier, the changes in development focus are believed to contribute significantly to the validity of relationships between test failures. Thus, other products with different release cadences may experience different results.

As no test or product development environment is ideal, the set of tests which could be and were run varied in each test run. This study ignores all tests that did not exist prior to a given test run, or which stopped existing after that test run due to standard ongoing development during the release. Since a test being added or removed from the test suite is not an indication of previous or future failures, it was ignored. Because of this, not all test results were considered in the study. Exclusion of results for this reason may have had an impact on the results of the study.

6. RELATED WORK

Performing cost effective regression testing is an important issue for most companies, in particular, for those who build large-scale products with large number of test cases. Advanced regression testing techniques can improve the cost-effectiveness of regression testing by identifying important tests that are more likely to detect defects. Identifying such test cases can decrease the time to finding and fixing bugs, as well as minimize the resources required to run test suites. A wide variety of regression testing techniques have been studied, and an overview of many of these techniques is discussed by Yoo and Harman [13] and Engstrom et al. [5].

A study relevant to our work is done by Nagappan and Ball [11]. The proposed approach has been used in many companies including Microsoft and is often referred to as *churn based testing*. It is the act of correlating the code changes made in a given build with the tests that should be run on that build. For instance, cross references in the product may indicate which tests call a method that was involved in the change, and then these tests would be run to validate the change. A problem with this approach comes when the cost of running the tests is prohibitively high, such as the case with this research. The number of code changes made between each regression test suite run was large enough that the vast majority of tests would be predicted in each test run. Therefore, our research focused on further reducing the set of predicted failures beyond what traditional code churn based testing would allow.

Another related area of study receiving attention is in the area of applying network analysis to software defects. These techniques range from network analysis of dependency graphs [15] to applying dependency graphs in determining testing strategies [8]. While our research focuses on the dependencies between tests, we are looking primarily at historical test results as opposed to static analysis of code and module dependencies such as this research.

Beyond the regression testing area, in the software engineering field, data mining has recently been used to analyze the voluminous amounts of data generated from version control systems or fault reporting systems [9, 12, 16]. Nagappan et al. [12] present an approach to mining data to predict error-prone components. To investigate their approach, they retrieved various software metrics and failure information from the version control system for five software projects developed at Microsoft. Zimmermann et al. [16] present an approach that applies data mining to provide information related to code changes to programmers, such as suggestions or predictions of likely changes. Livshits and Zimmermann [9] present an automatic way to discover common error patterns that reside in software revision histories by combining data mining with dynamic analysis techniques. All of these approaches described have shown that data mining could be useful in finding patterns and relationships that can help various software engineering tasks from massive software repositories.

7. CONCLUSIONS AND FUTURE WORK

As shown in this research, there are hidden relationships between test failures which may be mined to more accurately predict failures in future runs. It was also shown that the majority of these relationships are simple frequency relationships as opposed to more complex interactions between groups of test failures. This likely suggests that the tests themselves exhibit low coupling, which is advantageous. The tests are intended to not overlap each other in functionality, and this research suggests that they largely do not.

The other interesting result of this research is the drastic increase in prediction abilities when using a relatively small window of recent historical data, as opposed to full historical data. This suggests that not only are more recent failures better predictors than less recent failures, it also suggests that using less recent failures can be detrimental in prediction abilities. This is likely a result of different phases of development focusing on different areas of the product. While an area is being actively developed, tests that exercise that area are more likely to fail more often. When that development ceases, little variation in test failures is expected.

A potential future expansion of this research is to examine pairing frequent failure and windowing approaches with churn-based test prediction. Our results suggest that recent frequent failures are likely to reoccur, so it would be interesting to determine if additional tests beyond those proposed through churn-based techniques would be valuable to run. It would also be interesting to apply this same approach to additional industrial products to determine if similar relationships and distributions exist in other products.

8. ACKNOWLEDGMENTS

This work was supported in part by NSF CAREER Award CCF-1149389 to North Dakota State University, grants from the National Center for Research Resources (P20 RR016471) and the National Institute of General Medical Sciences (P20 GM103442) from the National Institutes of Health.

9. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, Sept. 1994.

[2] R. V. Binder. *Testing Object-Oriented Systems*. Addison Wesley, Upper Saddle River, NJ, 1999.

[3] 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*, 26(5), Sept. 2010.

[4] 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.

[5] E. Engstrom, P. Runeson, and M. Skoglund. A systematic review on regression test selection techniques. *Information and Software Technology*, 52(1):14 – 30, 2010.

[6] J. Han, J. Pei, Y. Yin, and R. Mao. Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach. In *Data Mining and Knowledge Discovery*, pages 53–87, Jan. 2004.

[7] B. Kaner and pettichord. *Lessons Learned in Software Testing*. Wiley Computer Publishing, 2002.

[8] B. Korel. The program dependence graph in static program testing. In *Information Processing Letters*, volume 24, pages 103–108, 1987.

[9] B. Livshits and T. Zimmermann. DataMine: Finding common error patterns by mining software revision histories. In *International Symposium on Foundations of Software Engineering*, pages 296–305, Sept. 2005.

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

[11] N. Nagappan and T. Ball. Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. In *International Symposium on Empirical Software Engineering and Measurement*, pages 364–373, 2007.

[12] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the International Conference on Software Engineering*, May 2006.

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

[14] M. Zaki and K. Gouda. Fast vertical mining using diffsets. In *Proceedings of the ningth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 326–335, 2003.

[15] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th International Conference on Software Engineering*, pages 531–540, 2008.

[16] T. Zimmermann, P. Weibgerber, S. Diehl, and A. Zeller. Mining versions histories to guide software changes. In *Proceedings of the International Conference on Software Engineering*, pages 563–572, May 2004.