First International Symposium on Empirical Software Engineering and Measurement

Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study

Nachiappan Nagappan
Microsoft Research
Redmond,USA
nachin@microsoft.com

Thomas Ball
Microsoft Research
Redmond,USA
tball@microsoft.com

Abstract

Commercial software development is a complex task that requires a thorough understanding of the architecture of the software system. We analyze the Windows Server 2003 operating system in order to assess the relationship between its software dependencies, churn measures and post-release failures. Our analysis indicates the ability of software dependencies and churn measures to be efficient predictors of post-release failures. Further, we investigate the relationship between the software dependencies and churn measures and their ability to assess failure-proneness probabilities at statistically significant levels.

1. Introduction

The IEEE standard [12] for software engineering terminology defines "architecture" as the organizational structure of a system or component. Related is IEEE's definition of architectural design [12] as the results of the process of defining a collection of hardware and software components and their interfaces to establish the framework (or architecture) for the development of a computer system.

In any large-scale software development effort, a software architecture enables teams to work independently on different components in the architecture. Large software systems often are decomposed hierarchically in a tree, where the leaves of the tree provide lower level services that components higher in the tree depend on. One usually finds that work groups are organized in a hierarchy that reflects the hierarchical organization of the software.

Overlaying this hierarchy is a graph of dependencies between the components. In theory, the dependencies between components would follow the edges of the tree. In practice, this rarely is the case. Dependencies may exist between peers in the software hierarchy or may cross many levels in the tree. The number of dependencies between parts of the hierarchy reflects the degree of 'coupling' between components, which can greatly affect the amount of work needed by development and test teams to keep the different components in synch.

We use software dependencies together with code churn to build models for predicting the post-release failures of system binaries. A software dependency is a relationship between two pieces of code, such as a data dependency (component A uses a variable defined by component B) or call dependency (component A calls a function defined by component B). Code churn is a measure of the amount of code change taking place within a software unit over time.

Suppose that component A has many dependencies on component B. If the code of component B changes (churns) a lot between versions, we may expect that component A will need to undergo a certain amount of churn in order to keep in synch with component B. That is, churn often will propagate across dependencies. Together, a high degree of dependence plus churn can cause errors that will propagate through a system, reducing its reliability.

In this paper we investigate the use of software dependencies and churn measures to explain post-release failures for a period of six months for the Windows Server 2003 operating system. Early estimates regarding the post-release failures can help software organizations to guide corrective actions to the quality of the software early and economically.

Software architectures [25] and software code churn [11] have been studied extensively in software engineering. We *quantify* how the dependencies that exist in the implementation of the software system and software churn measures correlate with post-release failures of a commercial software system. The size and wide-spread operational use of the system adds strength to our analysis.

In a prior study [19] we investigated the use of a set of relative code churn measures in isolation as predictors of software defect density. The relative churn [19] measures are normalized values of the various measures obtained during the evolution of the system. In an evolving system it is highly beneficial to use a relative approach to quantify the change in a system. We also showed that these relative measures can be devised to cross check each other so that the measures do not provide conflicting information.

Our current work differs significantly from our previous work as we integrate in architectural dependencies to investigate the propagation of churn across the system. This paper is one of the largest efforts to quantify architectural dependency entities. Our study serves to bridge the gap between the actual code and the architectural layout of the system. Further this paper uses only three simple churn measures in absolute terms compared to the multiple churn measures used in the relative approach in our prior study in order to show the effect of dependencies on code churn and failures.

Figure 1 illustrates the temporal aspect of the metrics collection. At the release date for Windows Server 2003 the dependency information is collected. Also at the release point, the code churn measures are collected using Windows 2000 as the baseline. This churn helps us quantify the evolution of the system in terms of change from the previous baseline version of Windows 2000 (The metrics are discussed in detail in Section 3).

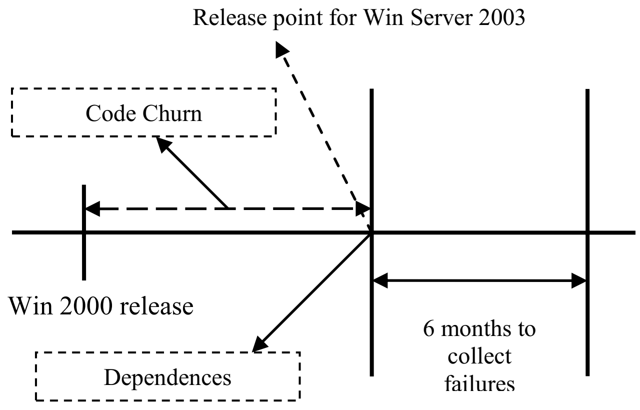

Release point for Win Server 2003
Code Churn
Win 2000 release
Dependences
6 months to collect failures

Figure 1: Data collection explanation

The overall size of the code base was 28.3 M LOC (Million Lines of code). The 28.3 M LOC comprises of 2075 compiled binaries that form the major part of the Windows Server 2003 system. In our analysis, software dependencies are between binaries (for example, in Windows, DLL files), and can be mapped up in the hierarchy of the system. Dependencies have two associated measures, the total number of dependencies (frequency) and the unique dependencies between the binaries (count). We leverage both these measures by computing the ratio of the dependence frequency to the dependence count to obtain a relative dependency ratio.

Software fault-proneness is defined as the probability of the presence of faults in the software [7]. Failure-proneness is the probability that a particular software element (binary, component or area) will fail in operation. We explain both failure-proneness and failures using our dependency and churn measures. The research hypotheses we investigate in our study are shown in Table 1.

Table 1: Research hypotheses

	Hypothesis
H_1	The software dependence ratios and churn measures are positively correlated;
H_2	The software dependence ratios and churn measures can be used as indicators of post-release failures in binaries
H_3	The software dependence ratios and churn measures can be used as indicators of the failure-proneness of the binaries

The rest of our paper is organized as follows. Section 2 describes our metrics in detail. Section 3 presents the results of our analysis and the experimental limitations. Section 4 discusses related work and Section 5 the conclusions and future work.

2. Software data and metrics

In this section we explain the software dependence and churn data that are collected. The Windows Server 2003 operating system is decomposed into a hierarchy of sub systems as shown in Figure 2. At the highest level we have an "Area" (shown by the dashed box). For example, an area would be "Internet Explorer" shipped with Windows Server 2003. Within Internet Explorer we could have several sub-systems. For example, the HTML rendering engine could be a component and the JavaScript interpreter could be another component (shown by rectangular solid boxes). Within each component we have several binaries denoted in Figure 2 by the black ovals. The binaries are the lowest level to which failures can be accurately mapped to. The 2075 binaries in our study are located within 453 'Components' which are present in 53 'Areas'.

2.1 Software dependencies

A software dependence is a relationship between two pieces of code, such as a data dependence, call dependence, etc. Microsoft has a completely automated tool called MaX [26] that tracks dependence information at the function level, including caller-callee dependencies, imports, exports, RPC, COM, Registry access, etc. MaX builds a system-wide

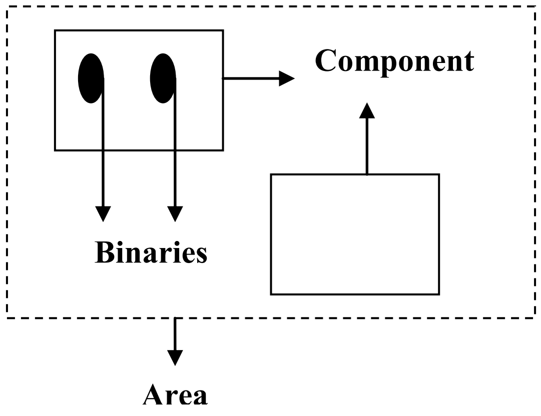


Figure 2: Sub-system description in Windows

dependency graph for Windows. The system-wide dependency graph can be viewed as the low-level architecture of Windows Server 2003. MaX works with systems consisting of both native x86 and .NET managed binaries and uses the dependency information of each binary to build the dependence graph. MaX [26] has been used to study what binaries (or procedures) will be affected by a change, so that corrective action can be taken accordingly.

We now explain the metrics associated with call dependencies (similar metrics are collected for data dependencies). A relation (A,B) between binaries A and B signifies that A makes a call on B. Given two entities A and B, A may call B from many different call sites. The ***count*** of a dependence (A,B) is either 0 or 1, based on whether A contains a call to B (1) or not (0). The ***frequency*** of a dependence (A,B) is the (total) number of calls from A to B. All information is analyzed and mapped at a binary level.

Based on this information we collect a set of eight dependency measures described below for each binary.

- Same Component Count
- Same Component Frequency
- Different Component Count
- Different Component Frequency
- Same Area Count
- Same Area Frequency
- Different Area Count
- Different area Frequency

Consider the dependence frequencies/counts for the binary (D) in Internet Explorer area shown in Figure 3. The Table in Figure 3 shows an example computation of the various dependence frequencies and counts and the measures that lead to those values. The binary (D) has three outgoing dependencies. Two of these are within the component (HTML rendering engine), directed from binary D to binary C. So the same component dependence frequency is two and the same component dependence count is one (i.e. D→C). There exists one dependence between the binary D and binary A in different component. The different component frequency and the different component count is thus one (D→A). There is a no dependence from binary D (in the Internet Explorer area) to the Control Panel area. This is the cross area dependency. The different area count and frequency are hence zero. Also within the Internet Explorer area, binary A has a same area dependency count of two and frequency of three. The dependency data is mapped for each binary accounting for its relationship to other binaries based on their locations in different areas/components. This allows us to measure how many dependencies a binary has in the same/different areas and components in order to quantify its architectural layout. Thus each binary in our analysis has its respective same/different component/area dependency counts and frequencies. The overall distribution of the dependencies in Windows due to space limitations in the paper are discussed in a technical report [18] .

2.2 Software churn

Code churn is a measure of the amount of code change taking place within a software unit over time. It is easily extracted from a system's change history, as recorded automatically by a version control system. We use a file comparison utility (such as diff) to automatically estimate how many lines were added, deleted and changed by a programmer to create a new version of a file from an old version. This measure is used to compute the overall change in terms of the lines of code (added, deleted, and modified). The software evolution measures that are collected are:

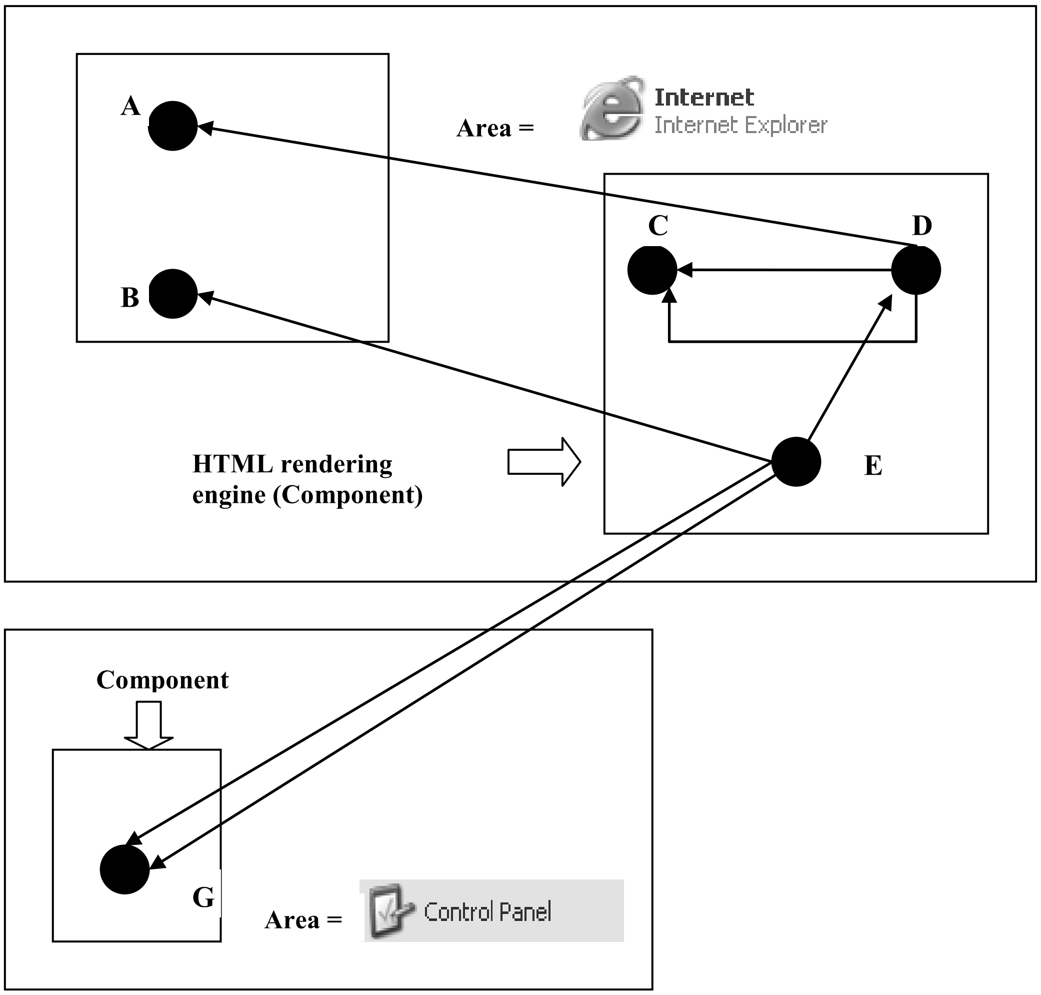
- Delta LOC: The overall change in the lines added, deleted or modified between two versions of a binary.
- Churn Files: The number of files within the binary that churned.
- Churn count: The number of changes made to the files comprising a binary between the two versions.

Code churn is used in our study as it measures the evolution of the system from a baseline version. As the system evolves from Windows 2000, our baseline version to Windows Server 2003 we measure the delta LOC, churned files and the churned count. Considering the churn measures from an architectural perspective, if for example the core Internet Explorer binary churns very frequently with a large delta LOC, then all binaries that have dependencies with this binary might churn (or change) appropriately to be in synch with the core Internet Explorer binary. If they are not in synch then it could lead to build failures which would be very expensive to fix late in the development process. This also explains our motivation to integrate churn and architectural dependency metrics to explain post-release failures. As with the dependencies each binary has its associated delta LOC, churn files and churn count mapped to it.

2.3 Metrics description

We utilize a set of seven metrics, four related to the dependency metrics and the remaining three based on the software churn measures. These metrics are described in Table 2.

In the computation in Table 2 we make two mathematical transformations. In the dependence ratios, all the denominator values are count+1 in order to eliminate division by zero. Second, we take a log transformation of the delta LOC in the software churn measures to scale the variables into a standard comparison scale [16].



Binary Name	Same Component Count	Same Component Frequency	Different Component Count	Different Component Frequency	Same Area Count	Same Area Frequency	Different Area Count	Different Area Frequency
D	1 (D→C)	2 (D→C) (D→C)	1 (D→A)	1 (D→A)	2 (D→C) (D→A)	3 (D→C), (D→C), (D→A)	0	0
E	1 (E→D)	1 (E→D)	2 (E→B) (E→G)	3 (E→B), (E→D), (E→G)	2 (E→B) (E→D)	2 (E→B) (E→D)	1 (E→G)	2 (E→G) (E→G)

Figure 3: Software dependency measurement description

The rationale for the dependence ratio is best described with an example. Consider two binaries, say X and Y, where binary X has a same component count of 2 and frequency of 7. Binary Y similarly has a same component count of 1 and frequency of 7. Computing a ratio lets us leverage the fact that though binaries X and Y have a similar number of dependencies they may differ in their failure-proneness ability as binary X is related to two different binaries compared to binary Y. This relative approach is similar to our prior work on code churn measures[19] where a relative approach yielded better predictors for estimating system defect density than absolute measures.

Table 2: Metric descriptions

Metric name	Description
Software Dependence Metrics	
Same component ratio	$\frac{\text{Same Component Frequency}}{\text{Same Component Count}}$
Different component ratio	$\frac{\text{Different Component Frequency}}{\text{Different Component Count}}$
Same area ratio	$\frac{\text{Same Area Frequency}}{\text{Same Area Count}}$
Different area ratio	$\frac{\text{Different Area Frequency}}{\text{Different Area Count}}$
Software Churn Measures	
Churn count	See Section 3.2
Churn files	See Section 3.2
Delta LOC	See Section 3.2

3. Experimental analysis

This section presents our experimental analysis of the relationship between software dependencies, churn and post-release failures. Section 4.1 investigates the empirical relationship between the software dependency and churn measures with the post-release failures using a Spearman rank correlation. Section 4.2 demonstrates the ability of the dependence ratios and software churn measures to explain the number of post-release failures, via multiple regression analysis. Section 4.3 shows how the metrics can be used to estimate the failure-proneness probabilities using logistic regression techniques. Section 4.4 discusses the threats to the validity of our study.

3.1 Correlation results

In order to identify the relationship between the software dependence ratios, churn measures and post-release failures we run a Spearman rank correlation (ρ) between the measures and the post-release failures. Spearman rank correlation is a commonly-used robust correlation technique [9] because it can be applied even when the association between elements is non-linear. The correlation results magnitude and sign can be used to identify the strength and characteristics of the relationship between the software dependence ratios, churn measures and post-release failures. The results of such a correlation are presented below in Table 3. From the correlation matrix it is interesting to observe the relationship between the same component ratio, delta LOC, churn times and churn files measures. These correlations show a positive statistically significant value confirming that when there are more dependencies for a binary and there are changes then there is a "propagation" of this change due to the dependencies, as indicated by the increase in the number of files churned and the number of times churned. This also confirms our initial argument about the propagation of churn across dependencies. We observe a similar relationship for the same area ratios also.

We see that correlations with failures except the different component/area ratio of dependencies are statistically significant at 99% confidence[1]. The lack of statistical significance of the different component/area can be attributed to the fact that compared to the number of dependencies among components; there is a proportionally smaller number of dependencies among areas.

The churn measures and other dependency correlations are all positive and statistically significant; indicating that with an increase in the ratios there is an increase in churn measures. *These results indicate that with an increase in the software dependency ratios the churn measures also increase (H1) except for the lack of statistical significance for the different component/area ratio which is due to the relatively small number of different area dependencies. This is purely a correlation and is thus only a preliminary result. We plan to do a causal root-cause analysis for confirming this hypothesis in future work.*

3.2 Post-release failure analysis

For explaining the post-release failures we use multiple linear regression (MLR), where the post-release failures form the dependent variable and the seven metrics described in Table 2 form the independent variables. The goal of this analysis is to identify if the software dependence ratios and churn measures can be used to model (explain at statistically significant levels) and estimate the post-release failures. In multiple linear regression, we measure the R^2 value and the F-test significance. R^2 is a measure of variance in the dependent variable that is accounted for by the model built using the predictors [3]. R^2 is a measure of the fit for the given data set. Additionally we present the adjusted R^2 measure also. Adjusted R^2 explains for any bias in the R^2 measure by taking into account the degrees of freedom of the independent variables and the sample population, i.e. in other words R^2 values keep increasing if we add more variables. The adjusted R^2 eliminates that bias regarding the number of variables. The adjusted R^2 tends to remain constant as the R^2 measure for large population samples. The computation of the adjusted R^2 is shown

[1] SPPS® for computation that does not give an accuracy of greater than 3 decimal places. So p=0.000 should be interpreted as p<0.0005

Table 3: Correlation results between the dependency ratios, churn measures and failures

		Same comp ratio	Diff comp ratio	Same area ratio	Diff area ratio	Churn count	Churn files	Delta LOC	Failures
Same comp ratio	ρ	1.000	.187	.954	.021	**.496**	**.538**	**.472**	**.401**
	(p)	.	(.000)	(.000)	(.334)	(.000)	(.000)	(.000)	(.000)
Diff comp ratio	ρ		1.000	.104	.032	.161	.176	.122	.181
	(p)		.	(.000)	(.150)	(.000)	(.000)	(.000)	(.000)
Same area ratio	ρ			1.000	.018	**.478**	**.519**	**.453**	**.382**
	(p)			.	(.406)	(.000)	(.000)	(.000)	(.000)
Diff area ratio	ρ				1.000	.036	.040	.047	.014
	(p)				.	(.101)	(.070)	(.033)	(.522)
Churn count	ρ					1.000	.946	.902	**.642**
	(p)					.	(.000)	(.000)	(.000)
Churn files	ρ						1.000	.917	**.652**
	(p)						.	(.000)	(.000)
Delta LOC	ρ							1.000	**.592**
	(p)							.	(.000)
Failures	ρ								1.000
	(p)								.

in Equation 1.

$$\text{Adjusted } R^2 = \frac{R^2 - (V^i - 1)}{(n - V^i) * (1 - R^2)} \quad \ldots (1)$$

where n is the number of samples used to build the regression model and V^i is the number of independent variables used to build the regression model. The F-test is a test of statistical significance used to test the hypothesis that all regression coefficients are zero. The F-test and its associated p values govern the acceptability of the built regression model in terms of statistical significance.

One difficulty associated with MLR is multicollinearity among the metrics that can lead to an inflated variance in the estimation of the failures. One approach that has been used to overcome this difficulty is Principal Component Analysis (PCA) [13]. With PCA, a smaller number of uncorrelated linear combinations of metrics that account for as much sample variance as possible are selected for use in regression (linear or logistic). From Table 3 we can see the statistically significant inter-correlations among the metrics. We ran a PCA on the seven metrics discussed in Table 2; the resulting principal components that account for a variance greater than 95% are shown below in Table 4. These resulting five principal components can be used to build regression models with minimal loss of information compared to the original metrics while handling the problem of multicollinearity. A logistic regression equation also can be built to model data using the principal components as the independent variable [7]. The individual and cumulative variances for each principal component that altogether account for 95% of the total sample variance is explained in Table 4.

Table 4: Principal component variances

	Initial Eigenvalues		
Principal Components	Total	% of Variance	Cumulative %
1	2.739	39.128	39.128
2	1.69	24.14	63.268
3	1.395	19.935	83.203
4	0.607	8.675	91.877
5	0.324	4.623	96.501

These five principal components are used to build our multiple regression equation. The overall fit using these principal components as the independent variable and the failures as the dependent variable is performed using all the 2075 binaries. The overall R^2 value of the regression fit is 0.629, (adjusted R^2 value =0.628) F=686.188, $p<0.0005$. The general form of the fit regression is shown in Equation 2.

$$\text{Post-release failures} = c + a_1PC1 + a_2PC2 + a_3PC3 + a_4PC4 + a_5PC5 \quad \ldots (2)$$

where c is the regression constant and $a_1, \ldots a_5$ are the regression coefficients. PC1…PC5 represent the principal components obtained from using the dependency and churn measures.

The regression model characteristics are shown Table 5. The coefficients of the multiple regression equation are removed to protect proprietary information. The individual metrics statistical significance is evaluated by the use of a t-test that indicates the significance of the principal components towards explaining the failures in terms of the multiple regression equation. The t-test and R^2 values show the efficacy of our regression model built for 2075 binaries using the five principal components as the independent variables.

Table 5: Complete model summary

	t	significance
(Constant)	56.441	$p < 0.0005$
Principal component 1	46.788	$p < 0.0005$
Principal component 2	-6.075	$p < 0.0005$
Principal component 3	-28.272	$p < 0.0005$
Principal component 4	15.336	$p < 0.0005$
Principal component 5	-13.055	$p < 0.0005$

In order to assess the ability of the regression models to predict the post-release failures we use the technique of data splitting [17]. That is, we randomly pick two-thirds (1384) of the binaries to build our prediction model and the remaining one-third (691) to verify the efficacy of the built model. Table 6 shows the results obtained on performing such random splits. In order to ensure the repeatability of our results we performed the random splitting five times. From Table 6 we see consistent R^2 and adjusted R^2 values that indicate the efficacy of the regression models built using the random splitting technique. The correlation results (both Spearman and Pearson) between the actual failures and estimated failures are at similar levels of strength and are statistically significant. This indicates the sensitivity of the predictions to estimate post-release failures. That is, an increase/decrease in the estimated values is accompanied by a corresponding increase/decrease in the actual values of post-release failures.

Table 6: Random data splits summary

R^2	Adj. R^2	F-test	Pearson *	Spearman *
0.641	0.639	480.776, $p<0.0005$	0.778	0.676
0.628	0.627	435.561, $p<0.0005$	0.793	0.662
0.634	0.633	468.756, $p<0.0005$	0.785	0.624
0.654	0.653	509.410, $p<0.0005$	0.748	0.620
0.614	0.612	427.999, $p<0.0005$	0.809	0.664

* All correlations are statistically significant at 99% confidence

Thus using principal component analysis of the metrics we can estimate the post-release failures at statistically significant levels. As the dependency information and churn measures are available early in the development process, we can use the *software dependency ratios and churn measures as early indicators of post-release failures (H2).* These estimates can be used to identify the binaries that will have a higher number of failures than acceptable standards and how testing can be directed more effectively at these binaries.

3.3 Failure-proneness analysis

In order to assess failure-proneness (i.e. simply stated the probability of a binary to fail in the field) we use a binary logistic regression approach. The overall goal of this analysis is to use the software dependence ratios and churn measures as early indicators of failure-proneness modeled using logistic regression techniques. The higher the failure-proneness of a binary, the higher is the likelihood of it failing in the field. For the purpose of using a binary logistic regression we need to define a binary-cutoff point for failure-proneness. For this purpose, we define binaries with no failures as not failure-prone and vice versa. The general form of a binary logistic regression is shown in Equation 3:

$$\Pi\text{ (prob)} = \frac{e^{(c + a1PC1 + a2PC2 + a3PC3 + a4PC4 + a5PC5)}}{1 + e^{(c + a1PC1 + a2PC2 + a3PC3 + a4PC4 + a5PC5)}} \quad \dots (3)$$

where the cut-off value for probability of failure π is 0.5 to classify binaries as failure-prone or not.

We perform the binary logistic regression using the principal components generated from the dependency ratios and the churn measures (independent variables) and failure-proneness (dependent variable). We also perform a repeated random splitting approach where we use two-thirds of the binaries to build the logistic regression equation and the remaining one-thirds to evaluate the built logistic regression model. Table 7 shows the results of this model building and the Nagelkerke's R^2 and the Cox and Snell R^2 [3] to present the overall consistency. The Nagelkerkes R^2 and the Cox and Snell R^2 are consistent across the five random splits (different from the random splits in Section 4.2) indicating the efficacy of the built logistic regression models in terms of the consistency of the datasets.

Table 7: Logistic regression model characteristics

Random split	Nagelkerkes R^2	Cox and Snell R^2
1	0.247	0.333
2	0.241	0.325
3	0.232	0.312
4	0.229	0.308
5	0.244	0.328

Similarly in order to evaluate the efficacy of the built logistic regression we use statistical correlations and compute the precision and recall ratios. A positive and strong correlation (we present both Spearman and Pearson correlations) between the predicted failure-proneness probability and the actual number of failures indicates the sensitivity of the failure-proneness probability (i.e. the higher the failure-proneness probability the higher the actual number of failures that occurred in the field). We use the precision and recall rates defined in Equation 4 and 5 to govern the accuracy of our models to identify failure-prone binaries. The results of the sensitivity of the predictions in terms of the correlations for the five random splits and precisions and recall values are shown in Table 8.

$$Precision = \frac{\text{Correctly predicted failure-prone binaries}}{\text{All predicted failure-prone binaries}} \quad (4)$$

$$Recall = \frac{\text{Correctly predicted failure-prone binaries}}{\text{All Observed failure-prone binaries}} \quad (5)$$

The correlation results in Table 8 are positive and statistically significant indicating that with increase in the predicted failure-proneness of the binaries there is an increase in the number of actual post-release failures. The precision and recall values indicates the accuracy of the predictions comparable with respect to earlier studies [23].

Table 8: Logistic regression random splits results

Pearson *	Spearman*	Precision	Recall
0.577	0.613	71.4 %	77.4 %
0.595	0.632	73.4 %	75.3 %
0.619	0.660	75.3 %	76.1 %
0.614	0.657	74.3 %	78.3 %
0.606	0.627	76.4 %	72.0 %

* All correlations are statistically significant at 99% confidence

Based on these results we can assess the ability of software dependency ratios and churn measures to be used as early indicators of the failure-proneness of the binaries (H3). For each binary we can compute this probability of failure using the logistic regression equation based on its evolution (churn) and architecture. Such failure-proneness information can be used to make early decisions on the quality of the binaries by focusing more test effort.

3.4 Threats to validity

In this section we discuss the main threats to validity of our study. The data analysis is performed as a post-mortem operation, i.e. all the data points are from the same software system. It is possible that these results might be valid only for the current system under study. We intend to address this issue in our future work plans by incorporating this methodology into the next generation Windows operating system development. These results were applicable to the Windows system that has strong prior history information that could be leveraged. This project had a timeline of analysis of 4.5 years. In the absence of prior historical information in this context it is not possible to make early estimates regarding the post-release failures/ failure proneness. We plan to investigate the topic of building prediction models using comparable projects in the absence of historical information. For a replication standpoint the MaX tool used for extracting the dependencies is internal to Microsoft. But there exists public tools like for Java, Dependency Finder or JDepend that can be used to extract dependencies to replicate these studies on open source systems. As with all empirical studies, these analyses should be repeated in different environments and in different contexts before generalizing the results. Results that were found in this study on the Windows operating system might differ with other software systems based on environment, context, size etc. We plan to join with researchers working on open source systems (similar to studies performed on Eclipse) to build an empirical body of knowledge of these results.

4. Related Work

Over the years research related to software architectures has ranged from analysis of mismatch of components in the architectural composition [6, 10] to architectural description languages [15]. Perry and Wolf [21] formulated a model of software architecture that emphasizes the architectural elements of data, processing, and connection, their relationships and properties. Shaw et al. [24] define *architectural style* to mean a set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem, together with local or global constraints on the way the composition is done.

Prior studies [1] analyzed and investigated error propagation through software architectures, focusing on the level of components and connectors (rather than dependencies between code). Von Mayrhauser [27] et al. investigated the relationship of the decay of software architectures with faults using a bottom-up approach of constructing a fault-architecture model. The fault-architecture model was constructed incorporating the degree of fault-coupling between components and how often these components are involved in a defect fix [27]. Their results indicated for each release what the most fault-prone relationships were and showed that the same relationships between components are repeatedly fault-prone, indicating an underlying architectural problem. Similarly studies have explored the use of software architectures for testing [22]. Our work is closely related to this effort as estimates of post-release failures can identify files that require more testing effort. Related to the prior work in this area we do not discuss any new ways for formalizing architectural connections or the use of any new architectural description languages. We simply leverage the software dependency metrics combined with the software churn measures to explain post-release failures.

Studies have also been performed on the distribution of faults during development and their relationship with metrics like size, complexity metrics [8]. From a design metrics perspective there have been studies involving the CK metrics [5]. These metrics can be a useful early internal indicator of externally-visible product quality [2]. The CK metric suite consist of six metrics (designed primarily as object oriented design measures): weighted methods per class (WMC), coupling between objects (CBO), depth of inheritance (DIT), number of children (NOC), response for a class (RFC) and lack of cohesion among methods (LCOM). The CK metrics have also been investigated in the context of fault-proneness. Basili et al. [2] studied the fault-proneness in software programs using eight student projects. They observed that the WMC, CBO, DIT, NOC and RFC were correlated with defects while the LCOM was not correlated with defects. Further, Briand et al. [4] performed an industrial case study and observed the CBO, RFC, and LCOM to be associated with the fault-proneness of a class.

Software evolution via code churn has been studied [11, 19, 20] to understand its relationship on software quality. Prior studies have involved analysis of code churn measures in isolation and using a relative code churn approach [19] to predict defects at statistically significant levels and as part of a larger suite of metrics [14] to understand defect density. Graves et al. [11] predict fault incidences in software systems using software change history. Ohlsson et al. [20] identify 25 percent of the most fault-prone components successfully by analyzing legacy software through successive releases using predominantly size and change measures. Zimmermann et al. [28] mined eight large scale open source systems (Eclipse, Postgres, KOFFICE, gcc, Gimp, JBOSS, JEdit and Python) version histories to guide programmers along related changes. They predict where future changes take place in systems and upon evaluation using these open source projects the top three recommendations made by them contained a correction location for future change with an accuracy of 70%. Schröter et al. [23] also showed that the import dependencies can predict defects in large systems like Eclipse.

We have discussed prior work on software architectures and the relationship between software metrics, churn and quality. Prior studies show that architectural mismatch, poor architectural design (or layout), and excessive dependence on a particular component are detrimental to the quality of software systems. Studies have also investigated how the architecture of the system can be used in the testing process. To the best of our knowledge, this paper is the first large scale empirical study in this area that uses architectural metrics and software churn to explain failures. The quantification of the propagation of failures in the system based on the churn (or evolution) and architecture of the system helps in early identification of binaries that require further testing.

5. Conclusions and future work

Software developers can benefit from an early estimate regarding the quality of their product as field quality information is often available late in the software lifecycle to affordably guide corrective actions. Identifying early indicators of field quality (in terms of failures/failure-proneness) is crucial in this regard. In this paper we have studied the software dependency and churn measures from the standpoint of the post-release failures for Windows Server 2003. From our analysis in section 4.1 we observe a preliminary correlation that with an increase in software dependency ratios there is an increase in the code churn measures. Based on statistical models built from a random two thirds of the data points to construct a prediction model and the remaining one third to evaluate the built model we get statistically significant results between the actual and estimated values for both, predicting post-release failures and failure-proneness. Table 6 and Table 8 summarize the strength of the statistical results in terms of the correlation between the actual and estimated values for estimating failures and failure-proneness and the precision and recall values across all the ten random splits (five for estimating the failures and five for estimating failure-proneness). The strength of the correlations indicates the sensitivity of the predictions which are all consistent across different random splits. These results indicate that we can predict the post-release failures and failure-proneness of the binaries at statistically significant levels. These predictions can help identify binaries that are more likely to fail in the field which can be used to focus efforts more efficiently to prioritize testing, perform code inspections etc.

Our current focus has been to leverage the architectural dependency and churn measures. We plan on using data from the testing process, pre-release faults, inspection data, faults found using static analysis tools etc. in future investigations. We also plan to do similar studies on other Microsoft products and with external researchers on non-Microsoft systems to compare our results to understand the underlying differences in architecture for different systems.

References

[1] W. Abdelmoez, Nassar, D.M., Shereshevsky, M., Gradetsky, N., Gunnalan, R., Ammar, H.H., Bo Yu, Mili, A, "Error propagation in software architectures", Proceedings of International Symposium on Software Metrics, pp. 384 - 393, 2004.

[2] V. Basili, Briand, L., Melo, W., "A Validation of Object Oriented Design Metrics as Quality Indicators", *IEEE Transactions on Software Engineering*, 22(10), pp. 751 - 761, 1996.

[3] N. Brace, Kemp, R., Snelgar, R., *SPSS for pshychologists*: Palgrave Macmillan, 2003.

[4] L. C. Briand, Wuest, J., Ikonomovski, S., Lounis, H., "Investigating quality factors in object-oriented designs: an industrial case study", Proceedings of International Conference on Software Engg., pp. 345-354, 1999.

[5] S. R. Chidamber, Kemerer, C.F., "A Metrics Suite for Object Oriented Design", *IEEE Transactions on Software Engineering*, 20(6), pp. 476-493, 1994.

[6] D. Compare, Inverardi, P., Wolf, A., "Uncovering Architectural Mismatch in Component Behavior", *Science of Computer Programming*, 33(2), pp. 101-131, 1999.

[7] G. Denaro, Pezze, M, "An empirical evaluation of fault-proneness models", Proceedings of International Conference on Software Engineering, pp. 241-251, 2002.

[8] N. Fenton, Ohlsson, N., "Quantitative analysis of faults and failures in a complex software system", *IEEE Transactions in Software Engineering*, 26(8), pp. 797 - 814, 2000.

[9] N. E. Fenton, Pfleeger, S.L., *Software Metrics*. Boston, MA: International Thompson Publishing, 1997.

[10] D. Garlan, Allen, R., Ockerbloom, J., "Architectural mismatch or why it's hard to build systems out of existing parts ", Proceedings of International Conference on Software Engineering, pp. 179-185, 1995.

[11] T. L. Graves, Karr, A.F., Marron, J.S., Siy, H., "Predicting Fault Incidence Using Software Change History", *IEEE Transactions on Software Engineering*, 26(7), pp. 653-661, 2000.

[12] IEEE, "IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology," 1990.

[13] E. J. Jackson, *A Users Guide to Principal Components*. Hoboken, NJ: John Wiley & Sons Inc., 2003.

[14] T. M. Khoshgoftaar, Allen, E.B., Goel, N., Nandi, A., McMullan, J., "Detection of Software Modules with high Debug Code Churn in a very large Legacy System", Proceedings of International Symposium on Software Reliability Engineering, pp. 364-371, 1996.

[15] N. Medvidovic, Taylor, R.N., "A Classification and Comparison Framework for Software Architecture Description Languages", *IEEE Transactions in Software Engineering*, 26(1), pp. 70-93, 2000.

[16] A. Mockus, Zhang, P., Li, P., "Drivers for customer perceived software quality", Proceedings of International Conference on Software Engineering (ICSE 05), St. Louis, MO, pp. 225-233, 2005.

[17] J. Munson and T. Khoshgoftaar, "The Detection of Fault-Prone Programs", *IEEE Transactions on Software Engineering*, 18(5), pp. 423-433, 1992.

[18] N. Nagappan, Ball, T., "Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study," Microsoft Research MSR-TR-2006-03, 2006.

[19] N. Nagappan, Ball, T., "Use of Relative Code Churn Measures to Predict System Defect Density", Proceedings of International Conference on Software Engineering, pp. 284-292, 2005.

[20] M. C. Ohlsson, von Mayrhauser, A., McGuire, B., Wohlin, C., "Code Decay Analysis of Legacy Software through Successive Releases", Proceedings of IEEE Aerospace Conference, pp. 69-81, 1999.

[21] D. E. Perry, Wolf, A.E., "Foundations for the Study of Software Architecture", *ACM SIGSOFT Software Engineering Notes*, 17(4), pp. 40-52, 1992.

[22] D. Richardson, Wolf, A., "Software Testing at the Architectural Level", Proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops, pp. 68-71, 1996.

[23] A. Schröter, Zimmermann, T., Zeller, A., "Predicting Component Failures at Design Time" Proceedings of International Symposium on Empirical Software Engineering, pp. 18-27, 2006.

[24] M. Shaw, Clemants, P., "Toward boxology: preliminary classification of architectural styles", Proceedings of the Second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops, pp. 50 - 54, 1996.

[25] M. Shaw, Garlan, D., *Software Architectures: Perspectives on an Emerging Discipline*: Prentice-Hall, 1996.

[26] A. Srivastava, Thiagarajan. J., Schertz, C., "Efficient Integration Testing using Dependency Analysis," Microsoft Research-Technical Report, MSR-TR-2005-94, 2005.

[27] A. von Mayrhauser, Wang, J., Ohlsson, M.C., Wohlin, C., "Deriving a fault architecture from defect history", Proceedings of International Symposium on Software Reliability Engineering, pp. 295 - 303, 1999.

[28] T. Zimmermann, Weißgerber, P., Diehl, S., Zeller, A., "Mining Version Histories to Guide Software Changes ", *IEEE Transactions in Software Engineering*, 31(6), pp. 429-445, 2005.