A Study on the Role of Software Architecture in the Evolution and Quality of Software

Ehsan Kouroshfar*, Mehdi Mirakhorli†, Hamid Bagheri*, Lu Xiao‡, Sam Malek*, and Yuanfang Cai‡
*Computer Science Department, George Mason University, USA
†Software Engineering Department, Rochester Institute of Technology, USA
‡Computer Science Department, Drexel University, USA

***Abstract*—Conventional wisdom suggests that a software system's architecture has a significant impact on its evolution. Prior research has studied the evolution of software using the information of how its files have changed together in their revision history. No prior study, however, has investigated the impact of architecture on the evolution of software from its change history. This is mainly because most open-source software systems do not document their architectures. We have overcome this challenge using several architecture recovery techniques. We used the recovered models to examine if co-changes spanning multiple architecture modules are more likely to introduce bugs than co-changes that are within modules. The results show that the co-changes that cross architectural module boundaries are more correlated with defects than co-changes within modules, implying that, to improve accuracy, bug predictors should also take the software architecture of the system into consideration.**

***Index Terms*—Software Repositories, Software Architecture, Defects.**

I. Introduction

Software engineers have developed numerous abstractions to deal with the complexity of implementing and maintaining software systems. One of those abstractions is software architecture, which has shown to be particularly effective for reasoning about the system's structure, its constituent elements and the relationships among them. Software architecture enables the engineers to reason about the functionality and properties of a software system without getting involved in low-level source code and implementation details.

At the outset of any large-scale software construction project is an architectural design phase. The architecture produced at this stage is often in the form of Module View [10], representing the decomposition of the software system into its implementation units, called *architectural modules*, and the dependencies among them.[1] This architecture serves as a high-level blueprint for the system's implementation and maintenance activities.

Well-designed software architecture employs the principle of separation of concern to allocate different functionalities and responsibilities to different architectural elements comprising the system [18], [23]. Conventional wisdom suggests that it is easier to make changes to a software system that has a well-designed architecture. Conversely, bad architecture, manifested as architectural bad smells [18], can increase the complexity, possibly leading to poor software quality [23]. In particular, scattered functionality, a well-known architectural bad smell, increases the system's complexity by intermingling the functionality across multiple architectural modules. While certain level of concern scattering is unavoidable due to non-functional concerns (e.g., security), a good architecture tries to minimize it as much as possible.

Monitoring the complexity of making changes to an evolving software system and measuring its effect on software quality are essential for a mature software engineering practice. It has been shown that the more scattered the changes among a software system's implementation artifacts such as source files and classes, the higher is the complexity of making those changes, thereby the higher is the likelihood of introducing bugs [22]. In addition, *co-changes* (i.e., multiple changed files committed to a repository at the same time) have shown to be good indicators of logically coupled concerns [17], which are known to correlate with the number of defects [5], [13].

However, a topic that has not been studied in the prior research, and thus the focus of this paper, is *whether co-changes involving several architectural modules(cross-module co-changes) have a different impact on software quality than co-changes that are localized within a single module (intra-module co-changes)*. Two insights seem to suggest that not all co-changes have the same effect. First, an architectural module supposedly deals with a limited number of concerns, and thus co-changes localized within an architectural module is likely to deal with less concerns than those that crosscut the modules. Second, it is reasonable to assume in a large-scale software system, the developers are familiar with only a small subset of the modules, and thus the more crosscutting the co-changes, the more difficult it would be for the developer to fully understand the consequences of those changes on the system's behavior.

Given that a large body of prior research has leveraged co-change history for building predictors (e.g., predicting bugs in a future release of the software) [12], [22], [27], [37], [41], a study of this topic is highly relevant, as it has the potential to support the construction of more accurate predictors by leveraging architecture information. In addition, empirical evidence corroborating our insights would underline the importance of software architecture in the construction and maintenance of software. In fact, the approach would pave the way for building predictors of architectural bad smells based

[1]The notion of *architectural module* should not be confused with *module* traditionally used in the literature to refer to files or classes. Here, we use the notion of module to mean architecturally significant implementation artifacts, as opposed to its typical meaning in the programming languages. Architectural modules represent the construction units (subsystems), and therefore, also different from *software components* that represent the runtime units of computation in the *Component-Connector View* [10].

978-0-7695-5594-2/15 $31.00 © 2015 IEEE
DOI 10.1109/MSR.2015.30

on the co-change and bug history of the software system, i.e., pointing out the parts of the architecture that would need to be refactored.

The contribution of this paper is two-fold. Firstly, it presents an empirical method designed for investigating yet unexplored but important software engineering research questions to better understand the impact of architecturally dispersed co-changes on software quality. In particular, we contribute two new metrics to quantify the differences between cross-module and intra-module co-changes.

Since in reality many software systems do not have a complete and updated documentation of their software architecture, our method introduces the concept of *"Surrogate Architectural Views"*. Surrogate views are obtained through a set of diverse reverse engineering methods to approximate the system's architecture for this experimental study.

The second contribution of this paper is in fact the observed results of applying this empirical method on development data of real, large scale, open source software systems, showing that *cross-module co-changes* are more correlated with defects than that of *intra-module co-changes*.

The remainder of paper is organized as follows. Section II describes the prior research. Section III provides an overview of our research methodology. Sections IV, V, and VI respectively explain the surrogate architectural views that we use in our study, details of our hypotheses, and data collection and analysis method. Section VII provides the results of empirical study, while Section VIII discusses the implications of our findings. The paper concludes with a discussion of threats to validity in Section IX and our future work in Section X.

II. Related Work

Several studies have shown that metrics mined from change history can be effective in locating defect-prone code areas [21], [31], [32]. Previous research has also investigated the relationships between code dependency and software quality [6], [9]. Yet another group of studies have investigated the relationships between change coupling and software quality [12], [37]. Our study, on the other hand, is different and new as it investigates the effects of change coupling together with syntactic dependency from an architectural perspective.

There is prior empirical evidence that the part of code that changes frequently tends to have more defects than other code areas [31], [32]. Metrics that measure code dependency are also known to be useful indicators of defect-prone code areas [8]. While code dependency can represent some level of logical relationships between code elements, change coupling metrics are known to be useful in finding hidden dependency between code elements [17].

Gall et al. [17] proposed the idea of logical coupling that can be identified from change coupling. They found that there is a stronger logical dependency between the changed subsystems when those systems change together in a long subsequence of releases.

Wong et al. [42] proposed an approach to detect object-oriented modularity violation by comparing the expected change coupling and actual change coupling. They identified expected change coupling using structural coupling identified based on the Baldwin and Clark's design rule theory and identified actual change coupling from software revision history.

Breu and Zimmerman used co-changes to identify crosscutting concerns [5]. The idea is that a code change is likely to introduce a crosscutting concern if various locations are modified within a single code change. This study did not consider the architecture of the system and also did not correlate the co-changes with bugs in the system.

Eick et al. used increases in change coupling over time as an indicator of code decay [14], [15]. Since change coupling can be an evidence of concern scatteredness [5], studies on concern scatteredness are relevant to our studies. Eaddy et al. [13] showed that the degree of concern scattering and the number of defects are strongly correlated. The biggest difference between their study and ours is that they manually mapped concerns to program elements to find concern scatteredness, while we use co-changes as an indirect indication of concern scatteredness. In addition, their metrics are at class level while our approach works at the architectural level.

D'Ambros et al.'s study [12] is closer to ours in that they identified the relationships between change coupling and defects. However, they performed the study at the class level, while our focus is at the architectural level. They did not distinguish between the change coupling of classes from different architectural modules and same module.

Martin and Martin [28] introduced the Common Closure Principle (CCP) as a design principle about package cohesion. This principle implies that a change that affects a component could affect all the classes in that component, but no other components. Although the authors introduce CCP as a guideline for good decomposition of architecture, they do not investigate the impact of it on software defects.

Shihab et al. [37] showed that the number of co-changed files is a good indicator of defects that appear in unexpected locations (surprise defects). Hassan [22] predicted defects using the entropy (or complexity) of code changes. Unlike these works, we examine the nature of logical coupling from an architectural point of view.

Mockus and Weiss [31] found that in a large switching software system, the number of subsystems modified by a change can be a predictor of whether the change contains a fault, but no definition of subsystem was provided, making it difficult to generalize their observation to other projects.

Yu et al. [44] has studied the correlation between multiple packages dependencies and their co-evolution. However this study does not take into account the impact of packages' co-evolution on software quality and defects.

Nagappan and Ball [32] used code dependency metrics and change history metrics to predict failure-proneness. However, they did not examine change-coupling effects. Additionally, they used Windows Server 2003 as a project under study, the source code and architecture of which is not publicly available.

In summary, while the majority of existing studies have focused on examining the effects of change-coupling on defects at the file level, we examine this at the architectural level. Moreover, our research is the first to show that reverse engineered approximation of the system's architecture is effective for conducting this type of studies. The first author has

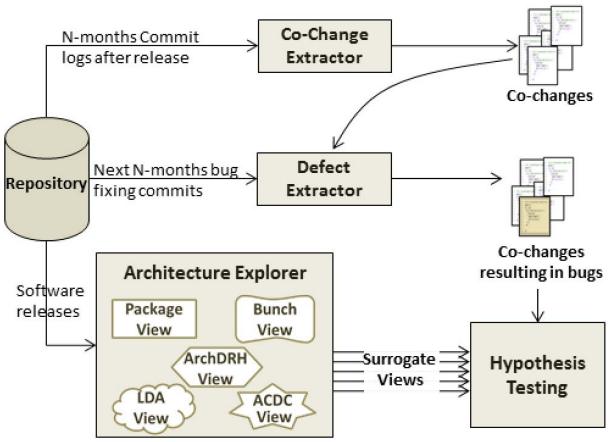

Fig. 1: Overview of the experimental method

previously published a two-page short paper on the importance of this study [24].

III. Methodology Overview

The method designed and implemented to run our empirical study involves four components represented by rectangles in Figure 1. The first component is *Co-change Extractor*, which searches source code repositories and retrieves the groups of files which have been changed together. It identifies the co-changes by going through the developer commits to the SVN repository and extracting the groups of files in the same commit transaction that have been modified together. The current implementation of the *Co-change Extractor* component utilizes *SVNKit*, a Java toolkit providing APIs to access and work with subversion repositories. This component has a modular design, and can be easily extended to support other source code repositories as well.

The second component is *Defect Extractor*, which parses the commit logs of projects and identifies the software changes which introduced the defects/bugs in the system. *Defect Extractor* and *Co-Change Extractor* components are synchronized with each other, to implement an *n-months data collection* approach, where the co-changes are extracted from the first n-months after a certain release and the introduced bugs are retrieved from the next n-months after the co-changes are retrieved. While *Co-change Extractor* component obtains the information of co-changes from the source code repository, the *Defect Extractor* component retrieves the information from the next n-months, and finds which of the original co-changed files has introduced defects in the next n-months time slice.

To examine the effects of co-change dispersion among the system's architectural modules on defects, we incorporated a third component, *Architecture Explorer*, which reconstructs the module view of architecture for our experiments. At the state of practice, there is no reverse engineering technique that can produce the "ground truth" architecture [19]. *Architecture Explorer* component thus utilizes different reverse engineering approaches and obtains several *Surrogate Views* that approximate the system's architecture. The surrogate views are then used in the last experimental module, *Hypothesis Testing*, where the effects of software co-change dispersion are examined from an architectural perspective. Particularly, we are interested in exploring three research questions:

> ***RQ1:*** *Are co-changes dispersed across multiple architectural modules more likely to have defects than co-changes localized within an architectural module?*

The positive answer to this question will enable the practitioners (software architects and developers) to use co-change dispersion metrics to assess quality of software, cope with architectural degradation, and also focus on important co-changes or architecturally significant ones first.

> ***RQ2:*** *Do different surrogates for module view exhibit different results in terms of the relationship between co-change dispersion and defects? If so, which surrogate module view provides better estimate of software defects?*

If the co-change dispersions measured from the various surrogate module views are different in their ability to reveal software defects, practitioners would need to use the views that best reveal defects to further inspect the root causes of the problems; otherwise, it would make more sense to use the view that is easier to obtain.

> ***RQ3:*** *Does a metric that differentiates cross-module co-changes have higher correlation with defects than a co-change metric that does not take into account the architecture?*

If that is the case, then using a metric that distinguishes between the different types of co-change could produce more accurate bug prediction models. The co-change differences, from an architectural perspective, is a factor that has been largely ignored in the prior research. To answer these questions, the *Hypothesis Testing* component utilizes new metrics we have defined in this paper plus a set of established statistical techniques.

IV. Obtaining Surrogates for Architectural Module View

In this section, we describe the architectural representations that we used in our study.

Comprehending the architecture and architecturally significant issues of any complex systems requires looking at the architecture from different perspectives [2], [25] .These perspectives are known as *architectural views*, each dealing with a separate concern. According to Clements et al. [10] three view types are commonly used to represent the architecture of a system: *Module View*, *Component-and-Connector View*, and *Allocation View*. Module View shows units of implementation, Component-and-Connector View represents a set of elements that have runtime behavior and interactions, and Allocation View shows the relationship between software and non-software resources of development (e.g., team of developers) and execution environment (e.g., hardware elements).

Since this study is concerned with the construction and evolution of software, and not its runtime or deployment

characteristics, Module View is the relevant view to focus on. Module View determines how a system's source code is decomposed into units and it provides a blueprint for construction of the software.

In reality, many projects lack trustworthy architecture documentation, therefore, we used different techniques to reverse engineer five surrogate models that approximate such architecture: *Package View*, *Bunch View* [30], *ArchDRH View* [7], *LDA View* [3] and *ACDC View* [39]. Although there might have been various techniques to reconstruct the architecture, we chose those that have the highest degree of automation, and therefore applicable to the context of our empirical study.

A. Package View

An intuitive approximation of the system's architecture in the Module View is Package View, where packages represent the system's architectural modules. It is reasonable to assume the package structure to be a good approximation of the decomposition of the system into architecturally significant elements, as packages are created by the developers of the system. In fact, package structuring has been used as a decomposition reference in prior research as well [4]. Therefore, one can say that package structuring of a Java project is representative of Module View architecture in which each architectural module consists of several Java classes (as a package) and the relation between them is *is-part-of*. There could be different decomposition layers when we are looking at the package structuring. It can be seen as a tree considering each class as a leaf that is part of a package, which itself may be part of a bigger package, with a top package as the root. Package view is considered at two levels. In the high-level package view, we consider each of the top-level directories as one of the architectural modules. For example, in *OpenJPA*, we consider each subfolder of the project (i.e., *org.apache.openjpa.jdbc*, *org.apache.openjpa.kernel* and *org.apache.openjpa.persistence*) as an architectural module. In the low-level view, architectural modules are represented by enclosing directories of each file.

B. Bunch View

Bunch [30] is a reverse engineering tool that produces clusters based on the dependencies among the classes. Prior research has shown that it is among the best available tools for reverse engineering the system's architecture [43]. Bunch relies on source code analysis tools to transform the source code to a directed graph which is a representation of the source code artifacts and their relations. Dependencies between classes are binary relations that are supported by programming languages like procedure invocation, variable access, and inheritance. The clustering output of Bunch is a representation of Module View architecture, where the elements of this architecture correspond to the clustered classes. These clusters represent *depends-on* and *is-a* relationships in the system.

C. ArchDRH View

Cai et al. [7] proposed an architecture recovery algorithm, called the Architectural Design Rule Hierarchy (ArchDRH). The key features of ArchDRH algorithm are as follows. First, it finds design rules and gives them a special position in the architecture, rather than aggregating them into subordinating modules as other clustering methods would do. Second, based on the observation that a software system usually has one or more main programs that depend on many other elements, acting as controllers or dispatchers, ArchDRH also separates these controllers and gives them a special position in the architecture. After that, ArchDRH separates the rest of the system into modules based on the principle of maximizing parallelism between modules. Concretely, given the dependency graph formed by the rest of the system, it calculates its connected subgraphs. For a subgraph that is still large, the algorithm further separates design rules and controllers within the subgraph, and processes the rest of it recursively till a stop condition is met, e.g, all the subgraphs are strongly connected. This way, the algorithm outputs a hierarchical structure, which is called a *design rule hierarchy*.

D. LDA View

Yet another way to reconstruct the modular decomposition of architecture is to utilize *Information Retrieval* and *Data Mining* techniques, such as *Latent Dirichlet Allocation (LDA)*, which is a known approach to automatically discover the underlying structure of the data. In the context of software engineering, this method has been used to discover the modular decomposition of a system [3], a conceptual implementation architecture [29] or, capturing coupling among classes in OO software systems [20].

LDA analyzes the underlying latent topics, words and terms used to implement each class/source files and discovers the most relevant topics describing the system. Therefore, based on similarity of each source file and the discovered topics, it decides which source files should be part of the same module.

Unlike the previous reconstruction approaches which utilize the structural dependencies between classes to find a potential modularization view, LDA uses the textual similarities between the contents of these classes and clusters them into different modules. The number of reconstructed modules is equal to the number of discovered underlying topics.

E. ACDC View

Algorithm for Comprehension-Driven Clustering (ACDC) [39] clusters program entities based on the principle of easing comprehension. Accordingly, it clusters program entities based on a list of system patterns, such as source file pattern, naming pattern, body-header pattern, leaf collection pattern, and support library pattern. After first constructing the skeleton of the system using these patterns, it then aggregates the leftover elements using orphan adoption methods [40]. The idea is to assign these elements into existing modules using various criteria, such as name similarity or dependency density between the element and a module. This algorithm also provides meaningful names for the resulting clusters and limits the cardinality of each cluster to ensure the resulting clusters are easy to understand.

V. Measuring Effects of Co-change Dispersion

The goal of our study is to examine the effects of co-change dispersion from an architectural perspective. To that end, we

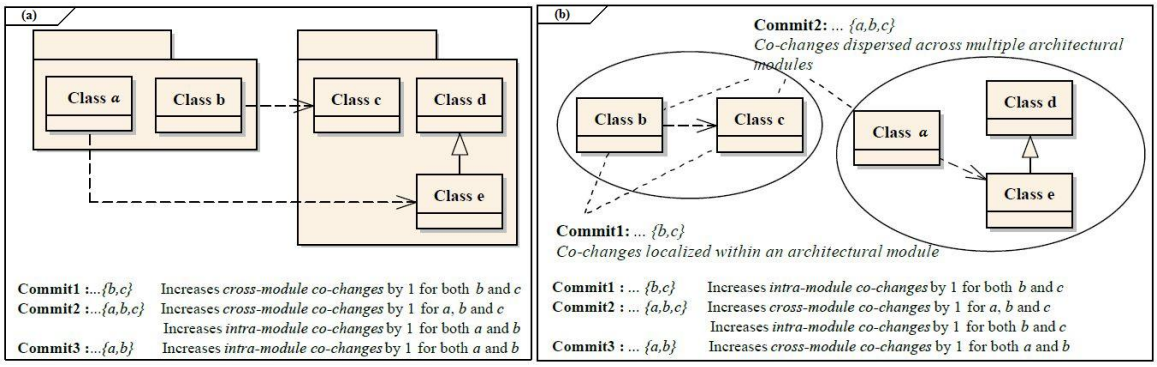

Fig. 2: Architectural Module View Surrogates of a system: (a) Package View, and (b) Bunch View.

formulated the three research questions described in Section III. In order to answer those questions, we define two metrics discussed in the following section.

A. Metric Definition

To answer *RQ1*, we compare the number of co-changes made within an architectural module and across multiple architectural modules for a given file. For this purpose, this section defines metrics to quantify the number of co-changes with respect to the system's architectural modules.

Let $S = < F, P_m, C >$ be a project, consisting of a set of files *F*, structured in a set of modules P_m under the architectural model *m*, and a set of commits C. Each file is assigned to a module and none of the modules overlap. More formally, the set $P_m \subseteq \mathcal{P}(F)$ is a partition of F under the architectural model *m*. The relationship between a file and a module in the *m* architectural model is captured by a function $p_m : F \rightarrow P_m$, and a set of co-changed committed files is identified by a function $h : C \rightarrow \mathcal{P}(F)$.

We can now define our two metrics for *intra-module co-changes (IMC)* and *cross-module co-changes (CMC)* using set cardinality expression.

Definition 1 (CMC)**.** *Number of co-changes for a file, f_i, where the co-changes are made across more than one architectural module:*

$$CMC(f_i) = \mathbf{card}(\{c : C |\ f_i \in h(c)\ \wedge\ \exists\ f_j\ \in\ h(c)\ .\ p_m(f_i)\ \neq\ p_m(f_j)\})$$

Definition 2 (IMC)**.** *Number of co-changes for a file, f_i, where there is at least another co-changed file in the same architectural module:*

$$IMC(f_i) = \mathbf{card}(\{c : C |\ f_i\ \in\ h(c)\ \wedge\ \exists\ f_j\ \in\ h(c)\ .\ p_m(f_i)\ =\ p_m(f_j)\})$$

More intuitively, Figure 2 illustrates the differences between these two metrics using two surrogate architectures for a small hypothetical example. Figure 2(a) depicts the Package View, which includes two packages and classes inside them, denoting the *is-part-of* relation. Based on this architectural view surrogate, package 1 and package 2 are the architectural modules of the system. Package 1 includes two classes of a and b; Package 2 includes three classes of c, d and e.

In Figure 2(b) an alternative surrogate view is shown, which is obtained by clustering the classes using Bunch (recall Section IV-B) . Here, there are two *depends-on* relationships (i.e., *a-e* and *b-c*) and one *is-a* relationship (i.e., *e-d*). Based on these dependency relations, Bunch generates two clusters: Cluster 1 includes classes *b* and *c*; Cluster 2 includes classes *a*, *e* and *d*. These clusters here are considered as the architectural modules of the system.

As illustrated at the bottom of Figure 2(a) and Figure 2(b), suppose that *a*, *b* and *c* are the co-changing files from three commits to the repository: {*a, b*}, {*a, b, c*}, {*b, c*}. All the files in the same set have been changed in a single commit. We are able to calculate our metrics from these commits. For example, in Figure 2(b), from the commit {*b, c*}, the values of *IMC* for both *b* and *c* increase by 1, because both of the files are in the same architectural module. From the commit {*a, b, c*}, the values of *CMC* for *a*, *b* and *c* increase by 1, because *a* is in a different architectural module. The *IMC* values for both *b* and *c* also increase by 1, since they are in the same architectural module.

B. Underlying Characteristics of the Data

Before adopting any statistical method to answer our research questions, we examined the nature and statistical characteristics of our mined data. This section presents the characteristics of our dataset and provides rationale for choosing the appropriate statistical methods.

The data used in our experiments are non-negative integers, representing the numbers of bugs, therefore can be considered as *count* or *frequency* data. By using Q-Q normal plot, we realized that the data does not follow a normal distribution. Also by using scatter plot of *CMC* and *IMC* with defects, we observed that these two metrics do not have a linear relationship with defects. The collected data, moreover, contains numerous zeroes (i.e., files that do not change or do not have defects). The same phenomena have also been observed by other researchers [34], [36], [45]. In fact, it has been shown that the distribution of fault data over modules in real systems are highly unbalanced [45].

C. Analysis Method

Considering the characteristics of data, we ruled out the option of using Linear Regression as it assumes the defects to be normally distributed [45], which is certainly not the case here. Unlike linear regression, *negative binomial regression (NBR)* makes no assumptions on either the linearity of the

TABLE I: Studied Projects and Release Information

Project	Description	Releases	SLOC	Architecture
HBase	Distributed Scalable Data Store	0.1.0 ,0.1.3 ,0.18.0 ,0.19.0, 0.19.3, 0.20.2, 0.89.20100621, 0.89.20100924, 0.90.2, 0.90.4, 0.92.0, 0.94.0	39K-246K	Recovered
Hive	Data Warehouse System for Hadoop	0.3.0, 0.4.1, 0.5.0, 0.6.0, 0.7.0, 0.7.1, 0.8.1, 0.9.0	66K-226K	Recovered
OpenJPA	Java Persistence Framework	1.0.1, 1.0.3, 1.1.0, 1.2.0, 1.2.1, 1.2.2, 2.0.0, 2.0.0-M3, 2.0.1, 2.1.0, 2.1.1, 2.2.0	153K-407K	Recovered
Camel	Integration Framework based on Enterprise Integration Patterns	1.6.0 ,2.0.M ,2.2.0 ,2.4.0, 2.5.0, 2.6.0, 2.7.1, 2.8.0, 2.8.3, 2.9.1	99K-390K	Recovered
Cassandra	Distributed Database Management System	0.3.0 ,0.4.1 ,0.5.1 ,0.6.2, 0.6.5, 0.7.0, 0.7.5, 0.7.8	50K-90K	Recovered
Hadoop	Distributed Computing Framework	0.19	224k	Ground-truth
System J	An Industrial Product	-	300K	Recovered

relationship between the variables, or the normality of the variables distributions. Therefore NBR is an appropriate technique to relate the number of defects in a source file to our two co-change metrics.

This model is thus applicable to count data and even more importantly addresses circumstances such as over-dispersion [11], as used in previous studies [34], [35].

We want to model the relationships between the number of defects (Y) in the source files and our two metrics. Suppose that y_i represents the number of defects for file i and x_i is a vector of our two metrics (*CMC* and *IMC*) for that file. NBR specifies that y_i, given x_i, has a Poisson distribution with mean $\lambda_i = \gamma_i e^{\beta' x_i}$, where β is a column vector of regression coefficients (β' is the transpose β) and γ_i is a random variable drawn from a gamma distribution [34].

Specifically, the output of this regression model is the vector $\beta = [\beta_1, \beta_2]$, where β_1 is the coefficient of *CMC* and β_2 is the coefficient of *IMC*. By using NBR, the expected number of defects varies as a function of *CMC* and *IMC* in multiplicative factor (unlike the case of linear regression, where the outcome is an additive function of the explanatory variables).

Since the co-changes data have a long tail (some files have a lot of co-changes compared to others), we use the log_2 transformation of our metrics to reduce the influence of extreme values [11]. Furthermore, as NBR models the natural log of the dependent variable (number of faults), the coefficients can be interpreted as follows: for one unit of change in the independent variable (e.g., *CMC*), the log of the dependent variable (number of faults) is expected to change by the value of the regression coefficient (β_1). To make the idea concrete, suppose that the coefficient of *CMC* is 0.8. This means that a unit change in log_2 of *CMC* is associated with an increase of 0.8 in natural logarithm of the expected number of defects. In essence, this would result in multiplicative factor of $e^{0.8} = 2.22$ in defects.

We hypothesize that co-changes cross different architectural modules are more correlated with defects than co-changes within the same architectural module. If our hypothesis is true, β_1 should be greater than β_2.

VI. Executing the Analysis

This section describes in detail how we conducted an experimental study to investigate our research questions about impacts of architecture and modularity on bugs.

Projects Studied. Our experimental subjects are seven projects from diverse domains, listed in Table I. For the Hadoop project, we had access to its ground-truth architecture, obtained through a manual recovery process by other researchers and verified by the key developers of Hadoop [19]. The last project that we studied is an industrial software project, called *System J*, which is codename for a system that has also been the subject of a prior empirical study [36]. It is a two-year old development project, comprised of about 300 KSLOC of Java in 900 files, and structured in 165 Java packages. The system aggregates a certain type of data from many sources and uses it to support both market and operational decision-making at a time granularity of minutes to hours. It has a service-oriented architecture and a transactional database, both implemented with third-party platform technologies.

Data Collection. There are several techniques for linking a bug database and a version archive of a project for finding fix-inducing changes. For example, searching the commit logs for specific tokens like *bugs, fixes* and *defects* followed by a number [38]. Unfortunately, developers do not always report which commits are bug fixes. Prior work suggests that such links can be a biased sample of entire population of fixed bugs [1]. But in the software repositories (Apache Foundation) and the projects studied in this paper, the commits that are bug fixes are distinguishable as they specify project name and bug number as a value pair in their commit logs in SVN.

We intentionally chose equal periods of time for collecting both co-changes and bug fixes in order to have a meaningful comparison of the results. We performed the study using both 3 and 6 months time intervals, which produced consistent results. Due to space constraint, the results reported in this paper are based on a 3 months time interval. In the first 3 months, we obtain the information of co-changes from the source code repository. Subsequently, in the next three months, we find the files that have been changed to fix bugs. This is repeated for the entire duration of the revision history.

At first it may seem that we could have simply used the time between consecutive releases, but we observed that in many cases, the periods of time between releases are not consistent. For example, the intervals between 4 consecutive releases of HBase project (0.90.3-0.90.6) are 66, 153, and 85 days. If we were to follow the release dates, one data point would be based on collecting the co-changes in 66 days and bug fixes in 153 days, while the next data point would be based on collecting the co-changes in 153 days and the bug fixes in 85 days. Rather, to have unbiased results required for conducting this study, we take the approach of using equal time for collecting both co-changes and bug fixes, which is consistent with previous studies in the literature [33].

Interested reader may access our research artifacts at: http://www.sdalab.com/projects/ccdispersion.

TABLE II: Regression results for architectural views of (a) Bunch, (b) ArchDRH, (c) ACDC, (d) High-level package, (e) Low-level package, and (f) LDA.

Project	Metrics	Bunch View		ArchDRH View		ACDC View		High-level Package		Low-level Package		LDA View	
		Est	Pr(> |z|)	Est	Pr(> |z|)	Est	Pr(> |z|)	Est	Pr(> |z|)	Est	Pr(> |z|)	Est	Pr(> |z|)
HBase	(Intercept)	-3.59	<2e-16	-3.57	<2e-16	-3.64	<2e-16	-3.71	<2e-16	-3.74	<2e-16	-3.74	<2e-16
	log(IMC)	-0.03	0.562	0.15	0.00169	0.02	0.588	0.13	0.0161	0.15	0.00772	0.14	0.0151
	log(CMC)	0.57	<2e-16	0.40	5.43e-10	0.52	<2e-16	0.40	3.7e-12	0.38	2.39e-09	0.36	9.86e-09
	log(LOC)	0.36	<2e-16	0.37	<2e-16	0.37	<2e-16	0.39	<2e-16	0.39	<2e-16	0.39	<2e-16
Hive	(Intercept)	-4.06	<2e-16	-4.34	<2e-16	-4.06	<2e-16	-3.65	<2e-16	-3.68	<2e-16	-4.21	<2e-16
	log(IMC)	0.15	0.102	0.13	0.171	0.26	0.00315	0.16	0.04	0.05	0.454	0.22	0.0252
	log(CMC)	0.53	1.29e-07	0.67	1.15e-09	0.48	7.76e-06	0.50	1.19e-10	0.72	<2e-16	0.56	4.09e-07
	log(LOC)	0.32	<2e-16	0.33	2.66e-15	0.31	<2e-16	0.28	<2e-16	0.24	<2e-16	0.31	<2e-16
OpenJPA	(Intercept)	-4.47	<2e-16	-4.70	<2e-16	-4.41	<2e-16	-4.37	<2e-16	-4.41	<2e-16	-4.42	<2e-16
	log(IMC)	0.01	0.922	0.19	0.00814	0.05	0.484	0.29	8.88e-05	0.02	0.709	0.05	0.438
	log(CMC)	0.59	6.02e-10	0.44	5.96e-06	0.55	1.02e-08	0.23	0.00526	0.62	4.68e-10	0.54	2.73e-09
	log(LOC)	0.37	<2e-16	0.40	<2e-16	0.37	<2e-16	0.39	<2e-16	0.36	<2e-16	0.37	<2e-16
Camel	(Intercept)	-4.14	<2e-16	-4.18	<2e-16	-4.17	<2e-16	-4.25	<2e-16	-4.30	<2e-16	-4.16	<2e-16
	log(IMC)	0.11	0.00155	0.09	0.00971	0.14	8.70e-05	0.18	9.52e-07	0.14	8.29e-05	0.07	0.0358
	log(CMC)	0.21	4.22e-09	0.25	1.96e-11	0.17	1.92e-05	0.17	4.13e-07	0.22	1.03e-08	0.25	3.62e-10
	log(LOC)	0.52	<2e-16	0.52	<2e-16	0.53	<2e-16	0.53	<2e-16	0.53	<2e-16	0.52	<2e-16
Cassandra	(Intercept)	-3.60	<2e-16	-3.48	<2e-16	-3.54	<2e-16	-3.20	<2e-16	-3.27	<2e-16	-3.28	<2e-16
	log(IMC)	0.20	0.0166	-0.05	0.336799	-0.06	0.418	-0.05	0.55872	-0.01	0.962640	-0.01	0.84988
	log(CMC)	0.41	1.36e-05	0.63	<2e-16	0.69	1.13e-11	0.69	7.75e-12	0.64	1.77e-10	0.64	8.72e-13
	log(LOC)	0.21	1.89e-05	0.18	0.000192	0.19	5.46e-05	0.14	0.00163	0.15	0.000942	0.15	0.00115
System J	(Intercept)	-3.58	<2e-16	-2.89	<2e-16	-2.82	<2e-16	-2.93	<2e-16	-2.98	<2e-16	-	-
	log(IMC)	-0.05	0.68	-0.02	0.79	0.09	0.31	-0.07	0.46	-0.03	0.53	-	-
	log(CMC)	0.82	<2e-16	0.75	<2e-16	0.72	<2e-16	0.78	<2e-16	0.81	<2e-16	-	-

VII. Results of the Study

This section describes the result of empirically analyzing the data in the manner described in the previous section.

A. Results for Research Question 1

To address our first research question—whether co-changes dispersed across architectural modules are more likely to have defects than intra-module co-changes—we use NBR to model the count data against the two metrics we defined (recall Section V-A). We include the file size (LOC) to control for the relationship between the size of files and the number of defects, as it could be argued that the larger files are more likely to have bugs and be a party in cross-module co-changes, thus creating a confounding effect [16], [45].

Table II summarizes the results for the five surrogate models of Bunch, ArchDRH, ACDC, Package and LDA view. Since we did not have access to the source code of the commercial project, we could not generate the data for the LDA view. Each row shows the regression coefficient for a variable along with the p-value. For example, the regression result for Bunch view in Hive project (Table II) indicates that the coefficient of *IMC* is 0.15 and its significance level is at 89% (the p-value is 0.102), while the coefficient of *CMC* is 0.53 and its significant level is more than 99% (the p-value is 6.02e-10).

We can see from these regression models that for the studied projects, the coefficient of *CMC* is highly significant and is greater than the coefficient of *IMC* in all surrogate views except the high-level package view—which will be discussed in the next subsection. In addition, we can observe that in several instances, the attribution of *IMC* in the model is not even significant. These data lend to support the proposition that *cross-module co-changes* have a bigger impact on the number of bugs than *intra-module co-changes*. We also observe no difference between the open-source projects and the commercial project.

Table III shows the results of the regression analysis for the ground truth architecture of Hadoop. The results of analyzing the ground truth architecture in this project are in line with those obtained using surrogate models for the other projects (i.e., *CMC* is highly significant and larger than *IMC*), thereby, giving us confidence in the validity of our conclusions.

TABLE III: Regression results for Hadoop and using the ground-truth architecture.

	Estimate	Pr(> |z|)
(Intercept)	-4.51	1.9e-06
log(IMC)	0.94	0.00034
log(CMC)	1.80	7.1e-09
log(LOC)	0.10	0.42524

Furthermore, we compared the Spearman correlation of *CMC* and *IMC* with defects (see table IV). As we can see, in all of the projects, CMC has higher correlation with defects. We used the Spearman rank correlation method, since it makes no assumption about the distribution of data, and thus is more appropriate for data that is not normally distributed.

Conclusion 1: Co-changes crosscutting the system's architectural modules have more impact on defects than co-changes localized within the same architectural modules.

B. Results for Research Question 2

Up to this point, we described the effect of co-change dispersion across architectural modules using the five surrogate models. In all of the projects that we investigated, co-changes crosscutting multiple architectural modules had a stronger impact on bugs than co-changes localized within the same architectural module. But which one of the views is a better predictor of defects and should be used to analyze the effect of co-changes? The answer to this question is relevant, as it helps the practitioners understand which view should be employed for collecting the data in practice.

To that end, we calculated the Spearman correlation between *CMC* and defects in all projects using the five views. Table IV summarizes correlation coefficients between defects and the *CMC* metric calculated for five different surrogate views of each project. The data shows consistently similar correlation between *CMC* and defects in all surrogate architectural views except in the high-level package view, where the correlation is relatively lower than other surrogate views. We also observed

TABLE IV: Correlation coefficients between defects and the metrics for cross-module co-changes (CMC), intra-module co-changes (IMC), and number of co-changed files (NCF). (Correlations significant at the 0.01 level are highlighted)

	HBase			Hive			OpenJPA			Camel			Cassandra			System J		
	CMC	IMC	NCF	CMC	IMC	NCF	CMC	IMC	NCF	CMC	IMC	NCF	CMC	IMC	NCF	CMC	IMC	NCF
Bunch	0.39	0.24	0.22	0.33	0.28	0.31	0.30	0.13	0.15	0.20	0.13	0.07	0.33	0.31	0.24	0.49	0.05	0.49
ACDC	0.39	0.21	0.22	0.37	0.29	0.31	0.31	0.13	0.15	0.20	0.11	0.07	0.34	0.27	0.24	0.52	0.33	0.49
ArchDRH	0.38	0.27	0.22	0.33	0.23	0.31	0.26	0.18	0.15	0.21	0.1	0.07	0.33	0.06	0.24	0.52	0.18	0.49
Package High	0.36	0.29	0.22	0.31	0.28	0.31	0.21	0.21	0.15	0.19	0.16	0.07	0.32	0.26	0.24	0.51	0.21	0.49
Package Low	0.36	0.26	0.22	0.38	0.22	0.31	0.32	0.15	0.15	0.22	0.12	0.07	0.32	0.27	0.24	0.52	0.19	0.49
LDA	0.36	0.24	0.22	0.36	0.29	0.31	0.31	0.15	0.15	0.20	0.15	0.07	0.33	0.25	0.24	-	-	-

TABLE V: Regression results for Bunch View including Num-cochanged-files.

	HBase		Hive		OpenJPA		Camel		Cassandra		System J	
	Est.	Pr(> |z|)	Est.	Pr(> |z|)	Est.	Pr(> |z|)	Est.	Pr(> |z|)	Est.	Pr(> |z|)	Est.	Pr(> |z|)
(Intercept)	-3.20	<2e-16	-4.49	< 2e-16	-4.35	< 2e-16	-3.82	< 2e-16	-2.59	2.24e-13	-2.77	0.00044
log2(IMC)	0.04	0.495515	0.07	0.44580	0.04	0.582	0.20	5.41e-07	0.27	0.0013	-0.06	0.59429
log2(CMC)	0.74	<2e-16	0.38	0.00149	0.67	2.15e-09	0.37	1.33e-15	0.77	1.33e-09	1.00	1.00E-07
log2(NCF)	-0.17	0.000189	0.16	0.01743	-0.07	0.227	-0.15	6.67e-08	-0.40	1.04e-05	-0.27	0.26421
log2(LOC)	0.34	<2e-16	0.33	< 2e-16	0.37	< 2e-16	0.51	< 2e-16	0.22	5.71e-06	-	-

that (cf. Table II) in two cases—i.e., OpenJPA and Camel high-level package views—*IMC* is even greater than *CMC*. Further analysis showed that high-level package view is not a proper representation for the architectural modules of a system due to its coarse granularity. For instance, nearly 65% of files in the Camel (version 2.9.1) are located in one of its top level packages (called "components").

The data suggests that developers can use any of the available surrogate views except the high-level packages to monitor the changes being made in the system. In fact, it means that even using the low-level package structure and not any complex reverse engineering methods can be helpful in monitoring the health of a system from its change history (e.g., identify co-changes that may indicate architectural bad smells, as further discussed in Section VIII).

Conclusion 2: No surrogate view is conclusively better than others, as they all—except the high-level package view—produce similar results in terms of the relationship between co-change dispersion and defects.

C. Results for Research Question 3

To address the third research question—whether a co-change metric considering architectural modules has higher correlation with defects than one that does not—we compare our *CMC* architecture-relevant metric with the *num-co-changed-files* (NCF) metric of Shihab et al. [37] to see which one is more correlated with defects. Their metric does not take into account the notion of architectural modules.

Shihab et al. [37] in their extensive study of bug prediction extracted 15 different metrics from three categories of (1) *traditional metrics* (e.g., file-size), (2) *co-change metrics* (e.g., num-co-changed-files) and (3) *time factors* (e.g., latest-change-before-release) to predict defects. Since some of these metrics are highly correlated, they performed a multicollinearity test to remove the metrics that have overlapping degree of impact. After removing the overlapping metrics, five were left that covered all of the three categories. One of these five metrics was *num-co-changed-files*, which indicates the total number of files a file has co-changed with. Note that *NCF* measures the *magnitude* of change, as opposed to whether the co-changed files were from different architectural modules or not.

We compared *NCF* with *CMC* to see which one is more correlated with defects. Table IV shows the results of Spearman correlation with defects. As we can see, in all of the projects, *CMC* has higher correlation with bugs.

To further evaluate the effect of the *NCF* metric, we regressed *NCF* and LOC against defects, and corroborated the earlier study that *NCF* has a significant positive impact on defects. However, as shown in Table V, when we added NCF in a regression model including our metrics (i.e., *CMC* and *IMC*), we see that the effect of *NCF* is often not statistically significant, and it does not have a positive impact on defects. This is while *CMC* remains positively correlated with defects, and its effect is consistently significant across the projects.

This result is interesting, as it indicates that the type of change (i.e., cross-module versus intra-module) is more important than the magnitude of change. It also suggests that using a metric that distinguishes cross-module co-changes has the potential to improve bug prediction accuracy. The co-change differences, in particular from an architectural perspective, is a factor that has been largely ignored in the prior research.

Conclusion 3: A co-change metric that considers architectural modules have higher correlation with defects than one that does not distinguish cross-module co-changes.

VIII. Discussion

Conventional wisdom suggests that a software system's architecture plays an important role in its evolution and maintenance, in particular the ease with which changes can be made to that system. In this study, we have tried to collect empirical evidence as to the role of software architecture in the evolution of a software system. We summarize the findings and the implications of our study here.

A. Role of Architecture in Maintenance

In Section VII-A, we showed co-changes that crosscut multiple architectural modules are more correlated with defects than co-changes that are localized in the same module. This

could be attributed to the fact that an architectural module supposedly deals with a limited number of concerns, and thus co-changes localized within an architectural module is likely to deal with less complicated issues than those that crosscut the modules. In addition, it is reasonable to assume in a large scale software system, the developers are familiar with only a small subset of the modules, and thus the more architecturally disperse the co-changes, the more difficult it would be for the developer to fully understand the consequences of those changes on the system's behavior, and therefore more likely to make changes that induce bugs.

There are also cases where dispersed co-changes, which have introduced bugs, have happened in source files without any apparent architectural dependencies. Further exploration, however, revealed the existence of tacit (indirect) dependencies among these files. Our metrics have shown to be effective in bringing awareness of these hidden coupling and complexities in the system's software architecture.

As an example, our *cross-module co-changes* metric helped us to discover one such case in Hbase project. This system utilizes ZooKeeper, an external middleware for providing high performance coordination in distributed environments. Furthermore, Hbase implements a *master-slave* architecture, where functionalities are clearly divided into separate roles of Master and Slave servers. But when looking at the recovered architecture of the system, and manually investigated the change logs of this system: we recognized that *HRegionServer.java* and *HRegion.java* files, located in the *slave* module, and *HBase.java* and *HMaster.java*, located in the *master* module exhibit very high *cross-module co-changes*, even though there is no direct dependency between them.

Our analysis showed that despite the fact that these files do not have any direct method calls, they are communicating with one another through ZooKeeper. Therefore, architectural decisions impacting one module would often impact the other module, thereby bringing about the observed co-change effect. Such architectural decisions included regular *synchronization* between master and slaves, *leader election* mechanism to handle the failure of a master, and data *locking and unlocking* used to manage access to shared resources. The existence of this tacit dependency has turned this part of the system into a critical spot, exposing inherent complexities that have resulted in various bugs. By looking at the bug reports for the involved modules, we observed bugs such as deadlock due to problematic implementation or modification of locking decision, performance issues due to excess synchronization between master and slaves, and various other bugs as the developers tweaked the code snippets related to the leader election mechanism originally used to handle fail over of master servers. The recurring bugs in this part of the system could be attributed to the lack of visible architectural dependency in the code, which our metrics could detect.

This result is useful, as it corroborates the conventional wisdom that the software architectural decisions (e.g., how a software system is decomposed into its elements) have a significant impact on the system's evolution. In addition, it underlines the impact of software architecture on open-source projects, a community that has not been generally at the forefront of adopting software modeling and architecting practices. We hope this study serves as an impetus for the open-source community to document and maintain the architecture of such systems alongside the code.

B. Building Better Bug Predictors

Co-changes have been used extensively in the past for building defect predictors [37]. Our study shows that not all co-changes have the same effect on the system's quality. Moreover, in Section VII-C, we showed that our co-change metric (*cross-module co-changes*) has a higher correlation with defects than a co-change metric that has been used previously in bug prediction models (*num-co-changed-files*). This implies that by distinguishing between the types of co-changes, it is possible to develop more accurate defect prediction models.

C. Architectural Bad Smell Predictors

We experimented with different surrogate representations of the system's architecture in our study. Our study shows that the correlation of *cross-module co-changes* and defects is statistically significant at 99% confidence interval in all data points using all of the views (recall Table II). We believe these experiments could inform future research in the discovery of *architectural bad smells*, i.e., architectural choices that have detrimental effects on system lifecycle properties. One approach to identify the architectural bad smells is to leverage the metrics introduced in this paper. For instance, by collecting the number of crosscutting co-changes per architectural module over several releases of a software system, one is able to identify the architectural modules that contribute the most to crosscutting co-changes and thus likely to harbor bad smells.

D. Empirical Research

Surprisingly few empirical studies have explored the impact of software architecture on its evolution. We believe this is mainly because many open-source software projects commonly used in the empirical software engineering research do not explicitly document and maintain the architecture of the system as it evolves. Thus, an implicit contribution of our work is the research methodology, whereby in the absence of actual models, multiple surrogate models were used as approximation of the system's software architecture. Although these surrogate models inevitably pose a threat to the validity of our results (as discussed in more detail in the next section), they also present a unique opportunity for the research community to investigate and learn from the vast information available in the open-source software repositories.

The potential of applying this methodology to study other relevant questions in light of the system's software architecture are promising. For instance, it is said that *multi-component defects* (i.e., defects requiring changes to multiple components of a software system) tend to expedite architectural degeneration of a system [27]. Similarly, it is said that architectural defects could account for as much as 20 percent of all defects, but compared to other types of defects they could consume twice as much time to fix [26]. However, adequate empirical research on open-source projects has not actually verified these behaviors. We believe the research methodology followed in

TABLE VI: Regression Results Using Random Clusters.

	HBase		Hive		OpenJPA		Camel		Cassandra	
	Est.	Pr(> |z|)	Est.	Pr(> |z|)	Est.	Pr(> |z|)	Est.	Pr(> |z|)	Est.	Pr(> |z|)
(Intercept)	-3.70	<2e-16	-4.04	< 2e-16	-4.43	< 2e-16	-4.19	< 2e-16	-3.47	< 2e-16
log2(IMC)	0.20	0.00078	0.46	5.75e-07	0.37	1.63e-05	0.22	1.73e-08	-0.07	0.466172
log2(CMC)	0.35	1.35e-08	0.31	1.72e-05	0.13	0.0595	0.11	0.00068	0.69	3.02e-10
log2(LOC)	0.38	<2e-16	0.33	< 2e-16	0.40	< 2e-16	0.52	< 2e-16	0.18	0.000179

our work (i.e., using the reverse engineered views of the system's architecture) could pave the way for empirically investigating such hypothesized phenomena.

IX. Threats to Validity

We now describe the main threats to validity of our findings.

A. Construct Validity

Construct validity issues arise when there are errors in measurement. First threat to validity is in the way we link bugs with the classes in the system. The pattern matching technique that we use to find bug references in commit logs does not guarantee to find all the links. Furthermore, since we are using bug fixes, not reported bugs, we do not consider faults that are reported, but not yet fixed. There may be modules with several reported defects that have not been fixed in the period of analysis, although the chances of that happening are low.

There is also a threat to validity of the results regarding the 3 months interval for data collection. However, as mentioned in Section VI, when we repeated the experiments using the 6 months interval, we obtained consistent results as those reported in the paper. In fact, using equal periods for collecting co-changes and bug fixes is an approach that we have borrowed from prior research [33].

There is also a threat to validity regarding the reverse engineering methods that we used. For example, Bunch uses several heuristics in a hill climbing approach to find the clusters and therefore the clustering results may be slightly different in consecutive runs on the same project. That said, Mitchell et al. [30] have showed that the result of Bunch is mostly stable over individual runs. Moreover, we did some sensitivity analysis and observed that these differences would not have a considerable effect on the results of our study. The other reverse engineering techniques have been previously used by other researchers; we also manually examined their accuracy and usefulness of their output before incorporating them in the paper.

One could argue our findings are not due to the recovered architectures and basically any clustering of files would have produced the same results. To assess this threat, we repeated the experiments by replacing the surrogate architectural modules with randomly constructed clusters. The results (summarized in Table VI) are not consistent across the projects. In HBase and Cassandra, CMC is greater than IMC, while in the other three projects we observe the reverse of that.

B. External Validity

External threats deal with the generalization of the findings. First, we intentionally chose projects that have bug-fix information in the commit logs, but this information may not be available for other projects. Second threat is related to the projects that we used in this empirical study, since all of them are developed in Java. An interesting future work could be to replicate this study on software projects implemented in other object oriented languages, like C++. Third threat is the way we defined Package View, which is based on package structuring of Java language and is only applicable to Java projects, although one may be able to use similar concepts in other programming languages, e.g., namespace in C#.

X. Conclusion

Software architecture plays an important role in the construction and maintenance of software. In practice, however, we see it being discounted, in particular by the open source community that has traditionally placed a premium on the code rather than the underlying architectural principles holding a system together. This is not to say such systems are devoid of architecture, but that the architecture is not explicitly represented and maintained during the system's evolution.

This paper reports on an empirical study that aims to provide concrete evidence of the impact of architecture on the quality of software during its evolution. Although several studies have used the co-change history to build bug prediction models, no prior study investigated the impact of co-changes involving several architectural modules versus co-changes localized within a single module. In the absence of explicit architectural models in open-source projects, to conduct this study we used surrogate models that approximate the architecture of a system. Our findings show that co-changes that crosscut multiple architectural modules are more correlated with defects than co-changes that are localized in the same module. We also arrived at the same conclusion when we performed the study using a commercial project, as well as an open-source project with a documented architecture. Our study corroborates the importance of considering software architecture as one of the key factors affecting the quality of a changing software system.

We are formulating a research agenda that aims to correlate the revision/defect history of software with its architecture to shed light on the root cause of problems. The insight in our research is that predicting where bugs are likely to occur, which has received much attention in the past decade, is not as useful to the developers as helping them understand why they occur. To that end, we believe architecture provides an appropriate level of granularity for understanding the root cause of a large class of defects that are due to bad architectural choices.

XI. Acknowledgments

This work was supported in part by awards CCF-1252644, CCF-0916891, CCF-1065189, CCF-1116980 and CCF-0810924 from the US National Science Foundation.

REFERENCES

[1] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links: bugs and bug-fix commits," in *18th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. FSE '10. Santa Fe, New Mexico: ACM, Nov. 2010, pp. 97–106. [Online]. Available: http://doi.acm.org/10.1145/1882291.1882308

[2] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[3] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Improving software modularization via automated analysis of latent topics and dependencies," *Accepted to appear in ACM Transactions on Software Engineering and Methodology (TOSEM)*.

[4] F. Beck and S. Diehl, "Evaluating the impact of software evolution on software clustering," in *17th Working Conference on Reverse Engineering*, Beverly, Massachusetts, Oct. 2010, pp. 99–108.

[5] S. Breu and T. Zimmermann, "Mining aspects from version history," in *21st IEEE/ACM International Conference on Automated Software Engineering*, Tokyo, Japan, Sep. 2006, pp. 221–230.

[6] L. C. Briand, J. Wst, J. W. Daly, and D. Victor Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," *Journal of Systems and Software*, vol. 51, no. 3, pp. 245–273, May 2000. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121299001028

[7] Y. Cai, H. Wang, S. Wong, and L. Wang, "Leveraging design rules to improve software architecture recovery," in *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures*, ser. QoSA '13. Vancouver, Canada: ACM, Jun. 2013, pp. 133–142. [Online]. Available: http://doi.acm.org/10.1145/2465478.2465480

[8] M. Cataldo, A. Mockus, J. Roberts, and J. Herbsleb, "Software dependencies, work dependencies, and their impact on failures," *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 864–878, 2009.

[9] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994. [Online]. Available: http://dx.doi.org/10.1109/32.295895

[10] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond*. Pearson Education, Oct. 2010.

[11] J. Cohen and J. Cohen, *Applied multiple regression/correlation analysis for the behavioral sciences*. Mahwah, N.J.: L. Erlbaum Associates, 2003.

[12] M. D'Ambros, M. Lanza, and R. Robbes, "On the relationship between change coupling and software defects," in *16th Working Conference on Reverse Engineering*, Lille, France, Oct. 2009, pp. 135–144.

[13] M. Eaddy, T. Zimmermann, K. Sherwood, V. Garg, G. Murphy, N. Nagappan, and A. Aho, "Do crosscutting concerns cause defects?" *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 497–515, 2008.

[14] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 1–12, Jan. 2001.

[15] S. Eick, T. Graves, A. Karr, A. Mockus, and P. Schuster, "Visualizing software changes," *IEEE Transactions on Software Engineering*, vol. 28, no. 4, pp. 396–412, 2002.

[16] K. El Emam, S. Benlarbi, N. Goel, and S. Rai, "The confounding effect of class size on the validity of object-oriented metrics," *IEEE Transactions on Software Engineering*, vol. 27, no. 7, pp. 630–650, Jul. 2001.

[17] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *International Conference on Software Maintenance*, Bethesda, Maryland, Nov. 1998, pp. 190–198.

[18] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *13th European Conference on Software Maintenance and Reengineering*, Kaiserslautern, Germany, Mar. 2009, pp. 255–258.

[19] J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic, "Obtaining ground-truth software architectures," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE '13. San Francisco, CA, USA: IEEE Press, May 2013, pp. 901–910. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486911

[20] M. Gethers and D. Poshyvanyk, "Using relational topic models to capture coupling among classes in object-oriented software systems," in *IEEE International Conference on Software Maintenance (ICSM)*, Timisoara, Romania, Sep. 2010, pp. 1–10.

[21] T. Graves, A. Karr, J. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653–661, 2000.

[22] A. E. Hassan, "Predicting faults using the complexity of code changes," in *31st International Conference on Software Engineering*, ser. ICSE '09. Vancouver, Canada: IEEE Computer Society, May 2009, pp. 78–88. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070510

[23] L. Hochstein and M. Lindvall, "Combating architectural degeneration: a survey," *Inf. Softw. Technol.*, vol. 47, no. 10, pp. 643–656, Jul. 2005. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2004.11.005

[24] E. Kouroshfar, "Studying the effect of co-change dispersion on software quality," in *ACM Student Research Competition, 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, May 2013, pp. 1450–1452.

[25] P. Kruchten, "Architecture blueprints–the '4+1' view model of software architecture," in *Tutorial Proceedings on Ada's Role in Global Markets: solutions for a changing complex world*, ser. TRI-Ada '95. Anaheim, CA, USA: ACM, Nov. 1995, pp. 540–555. [Online]. Available: http://doi.acm.org/10.1145/216591.216611

[26] M. Leszak, D. Perry, and D. Stoll, "A case study in root cause defect analysis," in *International Conference on Software Engineering*, Limerick, Ireland, Jun. 2000, pp. 428–437.

[27] Z. Li, M. Gittens, S. Murtaza, N. Madhavji, A. Miranskyy, D. Godwin, and E. Cialini, "Analysis of pervasive multiple-component defects in a large software system," in *IEEE International Conference on Software Maintenance*, Edmonton, Alberta, Sep. 2009, pp. 265–273.

[28] R. C. Martin and M. Martin, *Agile principles, patterns, and practices in C#*. Upper Saddle River, NJ: Prentice Hall, 2007.

[29] M. Mirakhorli, J. Carvalho, J. Cleland-Huang, and P. Mader, "A domain-centric approach for recommending architectural tactics to satisfy quality concerns," in *Third International Workshop on the Twin Peaks of Requirements and Architecture*, Rio de Janeiro, Brazil, Jul. 2013, pp. 1–8.

[30] B. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193–208, 2006.

[31] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/bltj.2229/abstract

[32] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *27th International Conference on Software Engineering*, St. Louis, Missouri, May 2005, pp. 284–292.

[33] ——, "Using software dependencies and churn metrics to predict field failures: An empirical case study," in *First International Symposium on Empirical Software Engineering and Measurement*, Madrid, Spain, Sep. 2007, pp. 364–373.

[34] T. Ostrand, E. Weyuker, and R. Bell, "Predicting the location and number of faults in large software systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340–355, 2005.

[35] D. Posnett, R. D'Souza, P. Devanbu, and V. Filkov, "Dual ecological measures of focus in software development," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. San Francisco, CA, USA: IEEE Press, May 2013, pp. 452–461. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486848

[36] R. Schwanke, L. Xiao, and Y. Cai, "Measuring architecture quality by structure plus history analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. San Francisco, CA, USA: IEEE Press, May 2013, pp. 891–900. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486910

[37] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan, "High-impact defects: a study of breakage and surprise defects," in *19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ser. ESEC/FSE '11. Szeged, Hungary: ACM, Sep. 2011, pp. 300–310. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025155

[38] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, May 2005. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083147

[39] V. Tzerpos and R. C. Holt, "ACDC: An algorithm for comprehension-driven clustering," in *Proceedings of the Seventh Working Conference on Reverse Engineering*, ser. WCRE '00. Washington, DC, USA: IEEE Computer Society, Nov. 2000, p. 258. [Online]. Available: http://dl.acm.org/citation.cfm?id=832307.837118

[40] V. Tzerpos and R. Holt, "The orphan adoption problem in architecture maintenance," in *Working Conference on Reverse Engineering*, Amsterdam, The Netherlands, Oct. 1997.

[41] R. J. Walker, S. Rawal, and J. Sillito, "Do crosscutting concerns cause modularity problems?" in *20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. Cary, North Carolina: ACM, Nov. 2012, pp. 49:1–49:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393654

[42] S. Wong, Y. Cai, M. Kim, and M. Dalton, "Detecting software modularity violations," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. Honolulu, Hawaii: ACM, May 2011, pp. 411–420. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985850

[43] J. Wu, A. Hassan, and R. Holt, "Comparison of clustering algorithms in the context of software evolution," in *21st IEEE International Conference on Software Maintenance*, Budapest, Hungary, Sep. 2005, pp. 525–535.

[44] L. Yu, A. Mishra, and S. Ramaswamy, "Component co-evolution and component dependency: speculations and verifications," *IET Software*, vol. 4, no. 4, pp. 252–267, Aug. 2010.

[45] Y. Zhou, B. Xu, H. Leung, and L. Chen, "An in-depth study of the potentially confounding effect of class size in fault prediction," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 1, pp. 10:1–10:51, Feb. 2014. [Online]. Available: http://doi.acm.org/10.1145/2556777