An Empirical Study of Architectural Change in Open-Source Software Systems

Duc Minh Le[§*] Pooyan Behnamghader[§*] Joshua Garcia[‡] Daniel Link[§] Arman Shahbazian[§] Nenad Medvidovic[§]

[§]Computer Science Department
University of Southern California
Los Angeles, CA, USA
{ducmle,pbehnamg,dlink,armansha,neno}@usc.edu

[‡]Department of Computer Science
George Mason University
Fairfax, VA, USA
jgarci40@gmu.edu

***Abstract*—From its very inception, the study of software architecture has recognized architectural decay as a regularly occurring phenomenon in long-lived systems. Architectural decay is caused by repeated changes to a system during its lifespan. Despite decay's prevalence, there is a relative dearth of empirical data regarding the nature of architectural changes that may lead to decay, and of developers' understanding of those changes. In this paper, we take a step toward addressing that scarcity by conducting an empirical study of changes found in software architectures spanning several hundred versions of 14 open-source systems. Our study reveals several new findings regarding the frequency of architectural changes in software systems, the common points of departure in a system's architecture during maintenance and evolution, the difference between system-level and component-level architectural change, and the suitability of a system's implementation-level structure as a proxy for its architecture.**

***Index Terms*—software architecture, architectural change, software evolution, open-source systems, architecture recovery.**

I. Introduction

Software maintenance tends to dominate the cost and effort across activities in a system's lifecycle. Changes to a software system require understanding and, in many cases, updating its architecture. Over time, a system's maintenance is increasingly affected by architectural decay, which is caused by careless or unintended addition, removal, and modification of architectural design decisions [34]. Decay results in systems whose implemented architectures differ significantly, sometimes fundamentally, from their designed architectures.

The observation that architectural decay occurs regularly in long-lived systems has been part of software engineering folklore from the very beginnings of the study of software architecture [33]. It is widely accepted that, during the lifetime of a software system, *the system's architecture changes constantly*, leading to instances of decay. Consequently, to identify and track architectural decay across the evolution history of a software system, architectural change must be reliably determined and understood. In particular, engineers must be able to pinpoint important architectural changes at different levels of abstraction and from multiple architectural views, which can, in turn, point to factors that cause decay.

[*] Duc Le and Pooyan Behnamghader contributed equally to this work.

To study architectural change, the architecture at a given point in time during a system's evolution must be extracted. To that end, a number of software architecture *recovery* techniques have been designed [22], [13], [15], [36], [26], with the shared objective of analyzing a system's implementation in order to extract its architecture.

At the same time, there is a relative scarcity of empirical data about the nature of architectural change. One major reason behind this scarcity has been a limited understanding of the efficacy of existing architecture recovery techniques: How do we know that we can draw reliable conclusions about the architecture recovered from the code? Our recent work has studied this question. To better understand the accuracy of the existing architecture-recovery techniques and the conditions under which a given technique excels or falters, we performed an extensive comparative analysis of state-of-the-art recovery techniques [15]. To evaluate their accuracy, we developed [17] and applied [16] a process for producing "ground-truth" software architectures, which were used to assess the output of the automated recovery techniques.

With this improved understanding of the existing recovery techniques, we are well-positioned to study architectural change. To that end, we present a novel approach, *Architecture Recovery, Change, And Decay Evaluator (*ARCADE*)*. *ARCADE* is a software workbench that employs (1) a suite of architecture-recovery techniques and (2) a set of metrics for measuring different aspects of architectural change. *ARCADE* constructs an expansive view showcasing the actual (as opposed to idealized) evolution of a software system's architecture. While analogous analyses have been attempted at the level of system implementation [24], [19], [20], [11], [14], [30], *ARCADE* represents the first solution of which we are aware that enables investigating such issues at the level of architecture.

We have employed *ARCADE* in an empirical study in which we analyzed several hundred versions of 14 open-source Apache systems. Specifically, we applied three of the ten architecture recovery techniques that *ARCADE* currently implements. Two of these techniques—*Algorithm for Comprehension-Driven Clustering* (*ACDC*) [35] and *Architecture Recovery using Concerns* (*ARC*) [18]—recover conceptual views of a system's architecture; the third—*PKG*—recovers a system's package-level organization which represents the implementation

view of the architecture [23]. *ACDC* and *ARC* were chosen because they demonstrated better accuracy and scalability compared to other recovery techniques in our previous empirical evaluation [15]. *PKG* provides an objective (if partial from an architectural perspective) baseline for assessing our results. Additionally, the three techniques approach recovery from different, complementary angles: *ACDC* leverages a system's module dependencies; *ARC* relies on information retrieval to derive a more semantic view of a system's architecture; and *PKG* strictly reflects the system's implementation organization. The empirical study reported in this paper has resulted in the following findings regarding architectural changes in software systems:

1) A semantics-based architectural view (yielded by *ARC*) highlights notably different aspects of a system's evolution than the corresponding structure-based views (yielded by *ACDC* and *PKG*). We found several cases in which the semantics-based view revealed important architectural changes that remained concealed in the two structure-based views. At the same time, existing architecture recovery techniques have heavily relied on structural information [13], [15], [21], [22], [26]. This suggests that more research on semantics-based recovery is needed in order to properly aid software system maintenance.
2) Architectural changes occur *within* software components during a system's evolution, even when the system's overall architectural structure remains relatively stable. Intra-component architectural changes are especially important to track in cases of relatively small system evolution increments. Relying on the architecture's structural stability in those cases may conceal non-trivial issues that will become apparent much later, when subsequent architectural changes make them more difficult to address.
3) While useful as an accurate representation of how a system's code base is organized (i.e., of the system's "implementation architecture" view [23]), the package structure is a limited indicator of the system's underlying architecture. *PKG* yielded especially misleading results when implementation changes that were confined to specific, already existing packages actually had far-reaching architectural implications. Such implications were more readily uncovered by *ACDC* and *ARC*, and were independently confirmed by the authors through code and architecture inspections.
4) Finally, dramatic architectural change tends to occur, both, (1) between the end of one major version and the start of the next one, and (2) across one or more minor versions of a software system. In other words, minor versions may result in major architectural changes. Furthermore, we discovered that, in some cases, significant architectural changes happen between pre-releases of a minor version. In other words, major changes to a system's architecture occur very late in the run-up to a new release, when common sense suggests that the architecture should be stable. This suggests that a system's versioning scheme is not strongly related to the extent of architectural change. In turn, this may be an added factor complicating the maintenance of a system's architecture and contributing to architectural decay.

The remainder of the paper is organized as follows. Section II summarizes the two related research threads that have been brought together to enable the work described in this paper. Section III presents the details of the *ARCADE* workbench. Section IV describes the setup for our empirical study, Section V its key results, and Section VI the threats to its validity. A discussion of related work (Section VII) and conclusions (Section VIII) round out the paper.

II. Foundation

Our work discussed in this paper was directly enabled by two research threads: (1) architecture change metrics and (2) software architecture recovery. Before we discuss the details of the *ARCADE* workbench in Section III, we will summarize this foundational work. Some of the outcomes reported here were described in prior publications, while others are novel; we will clearly delineate the two in the remainder of this section.

A. Architectural Change Metrics

We consider architectural change at two different levels: system-level and component-level. At the system-level, architectural change refers to the addition, removal, and modification of components; at the component-level, architectural change reflects the placement of a system's implementation-level entities inside the architectural components (i.e., clusters). Studying architectural change at these two levels of abstraction allows us to determine when a system-level architectural view evolves significantly differently than a component-level view. Identifying such discrepancies may reveal points in a software system's evolution where architectural maintenance issues occur, as well as the scope of those issues.

Due to the lack of metrics for quantifying architectural change, we created two new metrics for our study: *a2a*, a system-level metric, and *cvg*, a component-level metric. These two metrics have been recently used in a study of the impact of the granularity of module dependencies on the quality of architecture recovery [25]. We will also describe a metric, *c2c* [15], because it enables the computation of *cvg*.

Architecture-to-architecture (*a2a*) is a metric we developed for assessing system-level change. *a2a* was inspired by the widely used MoJoFM metric [37]. MoJoFM proved to be ill suited for our study because it assumes that the component sets in the architectures undergoing comparison will be identical; this is unrealistic for systems whose versions are known to have evolved, sometimes substantially. *a2a* is a distance measure between two architectures:

$$a2a(A_i,A_j) = (1 - \frac{mto(A_i,A_j)}{aco(A_i)+aco(A_j)}) \times 100\%$$

$$mto(A_i,A_j) = remC(A_i,A_j) + addC(A_i,A_j) + remE(A_i,A_j) + addE(A_i,A_j) + movE(A_i,A_j)$$

$$aco(A_i) = addC(A_\emptyset,A_i) + addE(A_\emptyset,A_i) + movE(A_\emptyset,A_i)$$

where $mto(A_i,A_j)$ is the minimum number of operations needed to transform architecture A_i into A_j; and $aco(A_i)$ is the number of operations needed to construct architecture A_i from a "null" architecture $A_\emptyset$.

Functions *mto* and *aco* are used to calculate the total numbers of the five operations used to transform one architecture into another [28], [32]: additions (*addE*), removals (*remE*), and moves (*movE*) of implementation-level entities from one cluster (i.e., component) to another; as well as additions (*addC*) and removals (*remC*) of clusters themselves. Note that each addition and removal of an implementation-level entity requires two operations: an entity is first added to the architecture and only then moved to the appropriate cluster; conversely, an entity is first moved out of its current cluster and only then removed from the architecture.

In order to ensure that the minimal number of move operations $movE(A_i,A_j)$ is applied in calculating *a2a*, we have implemented an algorithm that finds the largest subset of elements that stay in the same cluster when transforming A_i into A_j. Our algorithm builds a weighted bipartite graph that comprises the clusters of A_i and A_j. The weight of the edge from $c_1 \in A_i$ to $c_2 \in A_j$ is $|entities(c_1) \cap entities(c_2)|$. We use the Hungarian algorithm [29] to find the maximum weighted matching in this bipartite graph. This matching between clusters of the two architectures maximizes the number of matched elements, and consequently minimizes $movE(A_i,A_j)$.

Cluster-to-cluster (*c2c*) is a metric we developed and applied in our recent work [15] to assess component-level change. This metric measures the degree of overlap between the implementation-level entities contained within two clusters:

$$c2c(c_i,c_j) = \frac{\left|entities(c_i) \cap entities(c_j)\right|}{max(|entities(c_i)|,\left|entities(c_j)\right|)} \times 100\%$$

where $entities(c)$ is the set of entities in cluster c; and c_i is a cluster from version i of system S. The denominator is used to normalize the entity overlap in the numerator by the number of entities in the larger of the two clusters. This ensures that *c2c* provides the most conservative value of similarity between two clusters.

Cluster coverage (*cvg*) is a new change metric we developed to indicate the extent to which two architectures' clusters overlap according to *c2c*:

$$cvg(A_1,A_2) = \frac{|simC(A_1,A_2)|}{|allC(A_1)|} \times 100\%$$

$$simC(A_1,A_2) = \{c_i \mid (c_i \in A_1, \exists c_j \in A_2)(c2c(c_i,c_j) > th_{cvg})\}$$

$simC(A_1,A_2)$ returns the subset of clusters from A_1 that have at least one "similar" cluster in A_2. More specifically, $simC(A_1,A_2)$ returns A_1's clusters for which the *c2c* value is above a threshold th_{cvg} for one or more clusters from A_2. $allC(A_1)$ returns the set of all clusters in A_1.

cvg allows an engineer to determine the extent to which certain components existed in an earlier version of a system or were added in a later version. For example, consider a system whose version $v2$ was created after $v1$, and for which $cvg(A_1,A_2) = 70\%$, and $cvg(A_2,A_1) = 40\%$. This means that 70% of the components in version $v1$ still exist in version $v2$, while $100\% - cvg(A_2,A_1) = 60\%$ of the components in version $v2$ have been newly added.

B. Architecture Recovery Tool Suite

We recently conducted a comparative evaluation of software architecture recovery techniques [15]. The objective was to evaluate the existing techniques' accuracy and scalability on a set of systems for which we and other researchers had previously obtained "ground-truth" architectures [16]. To that end, we implemented a tool suite offering a large set of architecture recovery choices to an engineer.

Our study shows that a number of the state-of-the-art recovery techniques suffer from accuracy and/or scalability problems. At the same time, two techniques consistently outperformed the rest across the subject systems. We select these techniques for our analysis. These two techniques—*ACDC* [35] and *ARC* [18]—take different approaches to architecture recovery: *ACDC* leverages a system's *structural characteristics* to cluster implementation-level modules into architectural components, while *ARC* focuses on the *concerns* implemented by a system. The former is obtained via static dependency analysis, while the latter leverages information retrieval and machine learning. *ACDC* [35] groups entities into clusters based on patterns, most of which involve the dependencies among the entities. For example, *ACDC*'s main pattern attempts to group entities so that only a single dependency exists between any two clusters. *ARC* [18] groups entities that handle similar system concerns into a single cluster. For instance, *ARC* may group together the entities that handle user interface behaviors. We complement these two clustering-based architectural views with *PKG*, a tool we implemented to extract a system's *package structure*. Package structure is considered a reliable view of a system's "implementation architecture" [23]; while not indicative of the actual architecture underlying the system [34], the package structure provides a useful baseline (a "sanity check") for our study.

III. ARCADE

To study architectural change and decay, we have developed *ARCADE*, a software workbench that (1) performs architecture recovery from a system's implementation, uses the recovered information to compute (2) architectural change metrics and (3) decay metrics, and (4) performs different statistical analyses of the obtained data. As discussed previously, this paper presents our study of architectural change. To that end, we will focus on the first two aspects of *ARCADE*.

ARCADE's foundational element is architecture recovery, depicted as the *Recovery Techniques* component in the pipeline represented in Figure 1. The architectures produced by *Recovery Techniques* are directly used for studying change. *ARCADE* currently provides access to ten recovery techniques; nine techniques use algorithms for clustering implementation-level elements into architectural components, while one technique reports the implementation view of a system's architecture (i.e.,

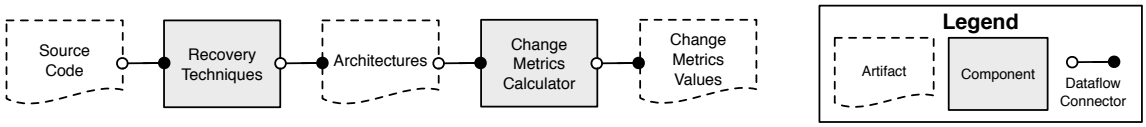

Fig. 1: ARCADE's components leveraged in this study and the artifacts it uses and produces.

the system's directory and package structure). *ARCADE* thereby allows an engineer (1) to extract multiple architectural views and (2) to ensure maximum accuracy of extracted architectures by highlighting their different aspects.

Since our previous evaluation [15] showed that two of the techniques—*ACDC* and *ARC*—exhibit significantly better accuracy and scalability than the remaining clustering-based techniques, and that they produce complementary architectural views (recall Section II-B), we focus on them in our study. *ACDC*'s view is oriented toward components that are based on structural patterns (e.g., a component consisting of entities that together form a particular subgraph). On the other hand, *ARC*'s view produces components that are semantically coherent due to sharing similar system-level concerns (e.g., a component whose main concern is handling of distributed jobs). We complement the architectures recovered by *ACDC* and *ARC* with each system's package-structure view extracted by *PKG*.

In implementing the required architecture recovery features in *ARCADE*, we had to overcome two related problems. First, in order to represent the topic models needed for *ARCADE*'s concern-based architecture recovery (*ARC*), we have used the MALLET machine learning toolkit [27]. The topic-model extraction algorithms implemented by MALLET are non-deterministic. On the other hand, in order to meaningfully compare two concern-based architectures as required for our study, we needed a shared topic model for their recovery. Therefore, for each subject system, we created a topic model by using all available versions of the system as the input to MALLET. The number of topics was determined based on our experience with *ARC* from a previous empirical evaluation [15]. We then used this topic model to retrieve the architectures for all of that system's versions.

In addition, we also computed architectural changes between a large number of pairs of different systems' versions by using topic models created from only the involved two versions. The architectural change results yielded by the two approaches—a single topic model for all system versions vs. different topic models for each pair of versions—are highly similar, with the variation of 1-2%. This supports our hypothesis that topic models created from a large number of versions would not produce significant noise when recovering the architecture of a particular version.

Second, to recover architecture using *ACDC*, we obtained an implementation of the technique from *ACDC*'s authors [35] and used its default settings. Although *ACDC* relies on a deterministic clustering algorithm, it turned out that its implementation is not deterministic, which created inaccuracies in our empirical analysis. We traced the source of *ACDC*'s non-determinism to the Orphan Adoption (OA) algorithm used in its implementation. OA is an incremental clustering algorithm that *ACDC* employs to assign a system's implementation entities to architectural components. The order of entities provided as input affects the result of OA, and subsequently the architecture recovered by *ACDC*. In the original implementation of *ACDC*, this order is not the same in every execution of the algorithm, causing the non-deterministic output. We resolved this problem by first sorting the input to OA based on the full package name of each class file.

For each architecture, *ARCADE* computes the three change metrics discussed in Section II-A. To that end, the *Change Metrics Calculator* component analyzes the architectural information yielded by *Recovery Techniques*. The computed metrics comprise the final artifact produced by *ARCADE* (*Change Metrics Values* in Figure 1) that is relevant to this paper.[1] This artifact is then used to interpret the degree of architectural change in the manner discussed in Section IV. For the three change metrics, *ARCADE* employs our own implementations.

ARCADE computes the *Change Metrics Values* by comparing the architecture of a software system version with the architectures of its ancestors or descendants. To conduct this comparison, *ARCADE* needs to know the evolution path of the software system. The *evolution path* is a sequence of version pairs. A *version pair* is an ordered pair (s,t) of versions from a given system, where t is the target version that evolved directly from the source version s. Each value for the three metrics from Section II-A is computed using a version pair. We obtained the correct evolution paths for our subject systems by using git-log [3] and svn-graph-branches [7]. *ARCADE* is implemented in Java and Python; it is available for download from [2].

IV. Empirical Study Setup

Our study targets four research questions regarding architectural change. The absence of empirical data on architectural change in real systems has resulted in that phenomenon being relatively poorly understood. As a result, the extent of architectural change, types of architectural change, and the points in a system's lifecycle when major architectural change occurs are generally unclear.

RQ1: To what extent do architectures change at the system level? This research question focuses on the structural stability of a system's architecture. During development and evolution, a system's implementation entities are usually reallocated (added, removed, moved) among its architectural components. This question will shed light on when, how, and to what extent this reallocation happens.

RQ2: To what extent do architectures change at the component level? This research question focuses on the structural

[1]The current version of *ARCADE* [2] also analyzes and quantifies different symptoms of architectural decay for a given system. However, these features are currently under evaluation and are outside the scope of this paper.

stability of a system's individual components. Implementation-level entities that realize an architectural component will change over time as the system evolves. Beyond a certain change threshold, it may be difficult to argue that an evolved component is still "the same" as the original component. This question will, therefore, study the component evolution patterns and thresholds.

RQ3: Do architectural changes at the system and component levels occur concurrently? This research question aims to reveal the extent to which changes to overall architectural structure are also accompanied by changes to individual system components, and when and why the two fall out of step.

RQ4: Does significant architectural change occur between minor system versions within a single major version? As a commonly adopted rule of thumb, developers decide to introduce a new major version for their system when the new APIs become incompatible with the previous versions (e.g., as in the case of the Apache Portable Runtime (APR) project [1]). In turn, this should imply a substantial change to the system's architecture. This research question will target our hypothesis that a system's architecture may experience significant change even though the system remains in the same major version.

In order to answer these research questions, we applied *ARCADE* to a total of 572 versions of 14 Apache open-source systems. The largest versions of these systems range between 150KSLOC and 800KSLOC. All of these systems are implemented in Java and managed in the Apache Jira repository. Table I summarizes each system we analyzed, its application domain, number of versions analyzed, timespan between the earliest and latest analyzed version, and cumulative size of all selected versions.

We applied *ARCADE*'s workflow depicted in Figure 1 to the different versions of each system. For each version, *ARCADE* produced (1) three recovered *Architectures*, by *ACDC*, *ARC* and *PKG*, and the values of *Change Metrics*. All artifacts produced in our study are available at [2].

In our analysis of the subject systems, we leveraged their shared hierarchical versioning scheme: `major.minor.patch-pre-release`. A *Major* version entails extensive changes to a system's functionality and typically results in API modifications that are not backward-compatible.

TABLE I: Subject systems analyzed in our study

System	Domain	No. of Ver.	Time span	MSLOC
ActiveMQ	Message Broker	20	8/04-1/07	3.40
Cassandra	Distributed DBMS	127	9/09-9/13	22.0
Chukwa	Data Monitor	7	5/09-2/14	2.20
Hadoop	Data Process	63	4/06-8/13	30.0
Ivy	Dependency Manager	20	12/07-2/14	0.40
JackRabbit	Content Repository	97	8/04-2/14	34.2
Jena	Semantic Web	7	6/12-9/13	3.50
JSPWiki	Wiki Engine	54	10/07-3/14	1.20
Log4j	Logging	41	01/01-6/14	2.40
Lucene	Search Engines	21	12/10-1/14	4.90
Mina	Network Framework	40	11/06-11/12	2.30
PDFBox	PDF Library	17	2/08-3/14	2.70
Struts2	Web Apps Framework	36	10/06-2/14	6.70
Xerces	XML Library	22	3/03-11/09	2.30
Total		**572**	**01/01-6/14**	**118.3**

A *Minor* version involves fewer and smaller changes than a major version and typically ensures backward-compatibility of APIs. A *Patch* version, also referred to as a *point version*, results from bug fixes or improvements to a system that involve limited change to the functionality. A *Pre*-release version, which can be classified as alpha, beta, or release candidate (RC), usually contains new features and is provided to users before the official version (major or minor) to get feedback.

This shared versioning scheme enabled us to make certain comparisons despite the differences among the systems and their numbers of versions. However, different systems follow different release evolution paths (recall Section III). Determining the accurate evolution paths for each system turned into an unexpected, non-trivial challenge. For example, in one system, version *1.2.0* may represent a direct evolution of version *1.1.7*; in another system, *1.2.0* may represent a completely new development branch. In order to determine the correct version sequences in our subject systems, we relied on git-log [3] and svn-graph-branches [7]. We then manually analyzed, and if appropriate updated, the results of those tools to ensure the accuracy of the suggested evolution paths.

In this process, we identified three frequently-occurring patterns that affected our selection of version pairs and evolution paths. In a number of cases, a minor version directly evolved from a previous minor version, rather than from a numerically more proximate patch version. Similarly, a new major version frequently evolved from a minor version, rather than from a numerically more proximate patch version; however, changes in patch versions would be merged at a later time. Lastly, the evolution paths for patch and pre-release versions typically followed the numeric ordering of their version numbers.

The evolution paths we selected in our study contain the four types of versions (*Major*, *Minor*, *Patch*, and *Pre*). In the case of major versions, we decided to consider two separate evolution paths because that allowed us to uncover different aspects of a system's evolution:

1) The evolution path involving all changes from the start of one major version to the start of the subsequent major version (e.g., the version pair $(1.0.0, 2.0.0)$). This evolution path represents the totality of changes a system undergoes within a single major version (hence we refer to it as *Major* below).
2) The evolution path involving a single version pair that comprises the last minor (or patch) version within a major version and the next major version (e.g., the version pair $(1.9.0, 2.0.0)$, where there are no other system versions between the two). This evolution path represents the degree of change to the system at the time the developers decide to make the "jump" to the next major version. We refer to this evolution path as *MinMaj*.

As an example of selected version pairs and evolution paths, consider the following set of versions obtained from the same system: *1.0.0*, *1.1.0*, *1.1.1*, *1.2.0*, *1.2.1*, *1.2.2*, *2.0.0-beta1*, *2.0.0-beta2*, and *2.0.0*. For the *Major* evolution path, only the

TABLE II: Average *a2a* values between versions; the bottom-most row is the average-of-averages. Value unit is percentage. Lower numbers mean more change. Empty table cells indicate versions that do not exist for a given system.

	ACDC					*ARC*					*PKG*				
System	Major	MinMaj	Minor	Patch	Pre	Major	MinMaj	Minor	Patch	Pre	Major	MinMaj	Minor	Patch	Pre
ActiveMQ	62	69	95	100	99	59	63	91	99	96	62	71	94	100	98
Cassandra	42	80	77	99	99	37	79	72	98	97	36	79	74	99	99
Chukwa	-	-	78	-	95	-	-	73	-	92	-	-	79	-	94
Hadoop	17	73	86	98	-	13	70	87	97	-	14	81	91	100	-
Ivy	50	67	91	98	99	29	56	84	91	97	35	57	89	98	99
JackRabbit	38	76	84	91	98	29	77	87	99	95	30	82	92	100	99
Jena	-	-	88	99	-	-	-	84	93	-	-	-	94	99	-
JSPWiki	18	30	86	98	99	7	21	76	98	99	8	13	87	99	100
Log4j	9	13	64	97	85	3	5	66	94	86	1	2	61	98	91
Lucene	12	8	96	98	94	8	7	97	100	93	1	1	97	99	90
Mina	28	30	92	99	88	14	14	93	99	84	13	13	98	100	86
PDFBox	-	-	97	97	-	-	-	91	99	-	-	-	97	100	-
Struts2	-	-	90	99	-	-	-	91	99	-	-	-	93	99	-
Xerces	21	54	92	83	-	16	52	88	83	-	15	63	91	90	-
AVG	**30**	**50**	**87**	**97**	**95**	**21**	**44**	**84**	**96**	**93**	**21**	**46**	**88**	**98**	**95**

pair $(1.0.0, 2.0.0)$ is in the path, as expected. On the other hand, for the *MinMaj* evolution path, $(1.2.0, 2.0.0)$ is in the path for this system, rather than $(1.2.2, 2.0.0)$. The *Minor* evolution path contains $(1.0.0, 1.1.0)$, as expected, but instead of $(1.1.1, 1.2.0)$ it contains $(1.1.0, 1.2.0)$. The *Patch* evolution path consists of the pairs $(1.1.0, 1.1.1)$, $(1.2.0, 1.2.1)$ and $(1.2.1, 1.2.2)$. Finally, the *Pre*-release path includes $(2.0.0\text{-}beta1, 2.0.0\text{-}beta2)$ and $(2.0.0\text{-}beta2, 2.0.0)$.

In addition to excluding minor and patch versions, as in the above example, in a limited number of cases we also excluded a major version along with all of its associated minor, patch, and pre-release versions. That occurred when a major version was actually an entirely different development branch from the system's other major versions. For instance, Struts 1 and Struts 2 [6] have been developed independently and comparing their architectures would yield no useful information from the perspective of architectural change. In this case, we selected Struts 2 for our study since it provided a richer set of minor, patch, and pre-release versions.

V. Results

To shed light on the four research questions about architectural change, we leveraged *ARCADE* to compute the *a2a* and *cvg* metrics (recall Section II-A). For each version pair within each evolution path of a system (recall Section III), we computed these metrics using the three architectural views produced by *ACDC*, *ARC*, and *PKG*. Table II shows the average *a2a* values and Table III the average *cvg* values for each system. Empty table cells indicate comparisons of versions that are invalid or cannot be determined. For example, if a software system has only one major version, architectural change values for $Major$ and $MinMaj$ cannot be computed. We discuss our findings for each of research question below.

A. RQ1: Architectural Change at the System-Level

To study RQ1, we leveraged *a2a*, which allows us to compute architectural change at the system-configuration level. Table II shows average *a2a* values for the five different types of evolution paths we selected across the three architectural views.

We observed a consistent trend for system-level architectural change among the three views. The *a2a* similarity values for the *Major* and *MinMaj* evolution paths are lower than for the remaining three types. This means that most significant architectural changes tend to involve major system versions. Although the differences among the values in the *Minor*, *Patch* and *Pre* columns are small, we can see a prevalent overall trend: $a2a_{Patch} > a2a_{Pre} > a2a_{Minor} > a2a_{MinMaj} > a2a_{Major}$. This observation is expected: as discussed earlier, patch versions usually come with bug-fixes, minor versions usually come with new features, and pre-release versions are wait-for-feedback versions of a minor or major version which usually require more changes than patch versions.

Differences between the $a2a_{MinMaj}$ and $a2a_{Major}$ values for a given software system reflect different aspects of change that has occurred both within and across that system's major versions. For example, in the case of Hadoop, $a2a_{MinMaj}$ is 73% while $a2a_{Major}$ is 17%. Hadoop had more than twenty minor versions between versions 0.1.0 and 0.20.x, before releasing version 1.0.0 [4]. We consider 0.1.0 to be Hadoop's first major release since it is, in fact, Hadoop's very first release. As a result, the architectural gap between version 0.1.0 and 1.0.0 is expected to be very large, yielding a low $a2a_{Major}$ value. On the other hand, changes between the last minor version and the subsequent major version that is derived from it (i.e., for the version pair $(0.20.0, 1.0.0)$) are comparatively small, resulting in a relatively high $a2a_{MinMaj}$ value.

This need not always be the case, however. There are situations where the architectural changes between consecutive minor versions within a single major version "cancel out" one another (e.g., when developers decide to revert their changes). In such cases, the $a2a_{MinMaj}$ value may be much closer to, or even lower than, $a2a_{Major}$. We will revisit this issue in Section V-D, when discussing RQ4.

Similarly, we have found situations where the architectural changes that occur between pre-releases surpass those that occur between minor versions. For example, in Log4j, $a2a_{Pre}$*(1.3-alpha6, 1.3-alpha7)* is lower (49% for *ACDC*, 45% for *ARC*, 48% for *PKG*) than Log4j's corresponding $a2a_{Minor}$ values (see Table II).

Obtaining the consistent trends across the recovered architectural views that are shown in Table II at times required that

TABLE III: Average *cvg* values between versions; the bottom-most row is the average-of-averages. Value unit is percentage. Lower numbers mean more change. Empty table cells indicate versions that do not exist for a given system.

	ACDC										*ARC*										*PKG*									
	Major		MinMaj		Minor		Patch		Pre		Major		MinMaj		Minor		Patch		Pre		Major		MinMaj		Minor		Patch		Pre	
System	(s,t)	(t,s)	(s,t)	(t,s)	(s,t)	(t,s)	(s,t)	(t,s)	(s,t)	(t,s)	(s,t)	(t,s)	(s,t)	(t,s)	(s,t)	(t,s)	(s,t)	(t,s)	(s,t)	(t,s)	(s,t)	(t,s)	(s,t)	(t,s)	(s,t)	(t,s)	(s,t)	(t,s)	(s,t)	(t,s)
ActiveMQ	28	19	61	48	95	92	100	100	99	99	36	26	55	52	89	84	99	99	94	91	33	27	60	67	96	93	100	100	100	97
Cassandra	5	4	59	53	52	46	98	99	98	98	35	17	73	63	63	53	96	95	93	93	29	18	77	69	69	60	98	100	99	99
Chukwa	-	-	-	-	63	54	-	-	86	86	-	-	-	-	54	44	-	-	72	72	-	-	-	-	76	67	-	-	93	93
Hadoop	0	0	54	46	83	74	95	98	-	-	20	2	72	51	81	75	95	95	-	-	0	0	54	46	95	85	100	99	-	-
Ivy	6	4	46	45	67	57	100	96	100	96	5	3	48	41	48	39	80	78	92	91	14	11	48	46	81	65	100	97	100	97
JackRabbit	16	7	53	57	87	81	98	97	96	96	41	13	64	63	83	76	98	98	91	90	28	12	65	73	93	86	99	98	98	98
Jena	-	-	-	-	81	74	96	96	-	-	-	-	-	-	79	75	89	89	-	-	-	-	-	-	97	93	99	99	-	-
JSPWiki	0	0	0	0	38	35	85	84	98	98	0	0	38	7	43	32	88	85	98	98	0	0	25	5	63	51	97	96	100	100
Log4j	0	0	0	0	29	21	94	93	85	82	6	1	4	1	56	40	86	83	79	76	0	0	0	0	69	54	99	97	92	88
Lucene	0	0	0	0	87	84	98	98	99	99	0	0	0	0	95	94	99	99	84	89	0	0	0	0	88	85	99	99	70	89
Mina	4	2	4	2	78	78	99	99	87	80	10	6	10	6	83	83	98	98	80	76	8	4	8	4	96	96	97	96	91	83
PDFBox	-	-	-	-	94	92	95	94	-	-	-	-	-	-	86	83	96	95	-	-	-	-	-	-	97	97	98	97	-	-
Struts2	-	-	-	-	79	83	96	96	-	-	-	-	-	-	77	77	94	94	-	-	-	-	-	-	91	95	98	98	-	-
Xerces	0	0	20	16	83	81	86	83	-	-	11	3	24	19	83	78	84	80	-	-	7	3	20	10	85	83	90	88	-	-
AVG	**6**	**4**	**30**	**27**	**72**	**68**	**95**	**95**	**94**	**93**	**16**	**7**	**39**	**30**	**72**	**66**	**92**	**91**	**92**	**91**	**13**	**8**	**36**	**32**	**85**	**79**	**98**	**97**	**94**	**94**

we manually adjust the inputs into two of the three architecture recovery techniques. Namely, in several instances we observed that *ARC* (the semantics-based view) provided a significantly better insight into architectural change than *ACDC* and *PKG* (the structure-based views). Inspection of our subject systems' source code uncovered that, in some systems (e.g., Log4j, Lucene), developers decided to change the root package name when releasing a new major version. Since *ACDC* and *PKG* rely directly on the package structure of the system, such an architecturally inconsequential change caused them to return exceptionally low *a2a* values. On the other hand, *ARC* performs clustering based on topic models of systems, and changing package names had no effect on its accuracy.

Although *PKG* performed significantly better at the system level than at the component level (see Section V-B), our analysis of the *a2a* metric's results provided the first indication that *PKG* may not always accurately reflect architectural change. Namely, the $a2a_{Patch}$ values for the architectures suggested by *PKG* are uniformly very high (98-100%). On the one hand, this is explained by the observation that changes in patch versions are usually confined to one or at most a few existing packages. On the other hand, this suggests a simple scenario under which *PKG* falters: if developers put all of the, arbitrarily many, new features of a system's new minor or major version into a small subset of the system's packages, *PKG* will still indicate only small, if any, architectural changes.

B. RQ2: Architectural Change at the Component-Level

To understand architectural change at the level of individual components, we relied on *ARCADE*'s *cvg* metric. In the results reported here, we set the threshold th_{cvg} (recall Section II-A) to 67%. We experimented with several th_{cvg} values, and 67% gave us the most intuitive result. This setting allows *ARCADE* to treat two clusters as different versions of the same cluster, while allowing a reasonable fraction of the new cluster's constituent code elements to change. Table III depicts average *cvg* values for architectures recovered by *ACDC*, *ARC*, and *PKG*. As in the case of *a2a*, these values are computed for *Major*, *MinMaj*, *Minor*, *Patch*, and *Pre*-release version pairs. Average *cvg* values are computed for each version pair (s,t), which obtains the percentage of extant components, and its inverse (t,s), which allows us to determine the extent to which new components were added to a version.

The *cvg* values for a version pair and its corresponding inverse pair shared the same general trend, across all three recovery techniques, that we observed with *a2a* values: $cvg_{Patch} > cvg_{Pre} > cvg_{Minor} > cvg_{MinMaj} > cvg_{Major}$. However, individual version pairs and their inverses were notably dissimilar in some cases. For example, across the four major versions of ActiveMQ, *ACDC* yielded $cvg_{MinMaj}(s,t) = 61\%$ and $cvg_{MinMaj}(t,s) = 48\%$. This means that a newly introduced major version retained 61% of the immediately preceding minor version's components. In turn, this comprised only 48% of the new major version's components due to the system's increase in size; the remaining 52% were newly introduced components. In other words, ActiveMQ grew by an average of 27% ($cvg_{MinMaj}(s,t)/cvg_{MinMaj}(t,s)$) in the number of components during the introduction of a new major version. Overall, the differences between the average *cvg* values for version pairs and their inverses across all subject systems (the AVG row in Table III) ranged between 0% (*Patch* versions in *ACDC*) and 9% (*MinMaj* versions in *ARC*).

All three recovery techniques show extensive component-level change at the *Major* and *MinMaj* levels. Conversely, all three show significant stability at the *Minor*, *Patch*, and *Pre*-release levels. However, the results yielded by analyzing *ARC*'s recovered architectures are notably different from, both, *ACDC* and *PKG*. First, both *PKG* and especially *ACDC* tended to under-report the degree of component-level similarity of architectures between major version pairs. In several cases, the two techniques yielded no similarity (the 0% values in Table III) even though a manual inspection of the corresponding versions suggested that some component-level similarity was, in fact, preserved. While *ARC* also yielded very low values for the same cases, in most of those cases it did, accurately, maintain some component-level similarity. The reason for this is that both *ACDC* and *PKG* rely on the system's structural dependencies and are significantly affected by changes that span

most or all of the system's implementation packages. On the other hand, *ARC*'s reliance on the information contained in the system's implementation elements, rather than on the relative organization of those elements, made it less susceptible to misinterpreting the large system changes that typically happen at the *Major* and *MinMaj* levels.

An analogous argument explains why *ARC* yields lower component similarity values for the *Minor*, *Patch*, and *Pre*-release levels: *ACDC* and especially *PKG* fail to recognize large architectural changes to system components if those changes are confined within a package or a small number of packages.

C. RQ3: System-Level vs. Component-Level Change

While the discussion of RQ1 and RQ2 indicated that architectural change followed the same general trends in our subject systems at the overall-structure and individual-component levels, the extent of that change differed. We can see significant differences between the *a2a* and *cvg* metrics. Furthermore, these differences steadily grow from *Patch* and *Pre*-release versions, where the two metrics return virtually identical values, to *Major* versions, where the *cvg* values are notably lower (e.g., see the AVG rows in Tables II and III). For example, all three architecture recovery techniques yielded 0% *cvg* values for JSPWiki's *Major* versions; none of them did so in the case of *a2a*.

Another revealing example is Lucene. Lucene may be thought of as a catalog of multiple information retrieval systems that have historically been added to and removed from it. For example, the Solr project was initially developed by CNET Networks, and later released as an open-source project and merged with the Lucene code base [5]. Due to this nature of Lucene, it has tended to undergo a lot of significant changes before the release of a new major version. Although some parts of the system structure would be maintained (indicated by $a2a_{Major}$ and $a2a_{MinMaj}$), Lucene's components changed significantly (both cvg_{Major} and cvg_{MinMaj} are 0% across all three recovery techniques).

We note that the growth of divergence between the *a2a* and *cvg* values from *Patch* and *Pre*-release versions at one end of the spectrum to *Major* versions at the other end is more pronounced in the case of *ACDC* and *PKG* than in the case of *ARC*. This is another indicator that the two structure-based recovery techniques are indeed much better equipped to track system-level than component-level architectural changes. Additionally, the consistently higher *a2a* values that both techniques yield as compared to *ARC* suggests that they tend to overestimate the actual architectural similarity of versions at the system level.

To illustrate the overestimation of architectural similarity by the structural views—*ACDC* and *PKG*—as compared to *ARC*, Figures 2–4 depict architectural changes among minor versions of Ivy: Figure 2 depicts the *a2a* values; Figure 3 depicts the *cvg* values for each version pair (s,t); and Figure 4 shows its inverse, i.e., the *cvg* values for version pairs (t,s). Note that, for clarity, we do not depict the *MinMaj* evolution paths in Figures 2–4. Finally, the lines connecting the discrete points in the three figures are intended only to aid visualizing of trends.

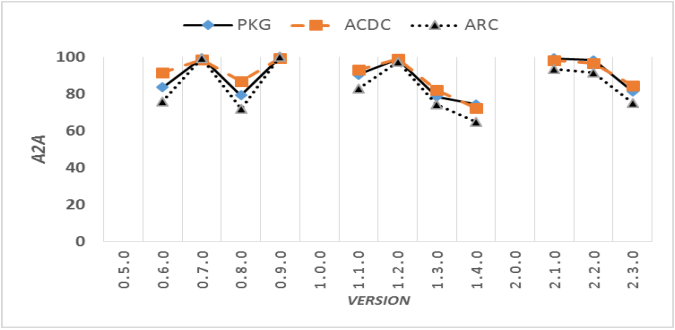

Fig. 2: *a2a* values between minor versions of Ivy

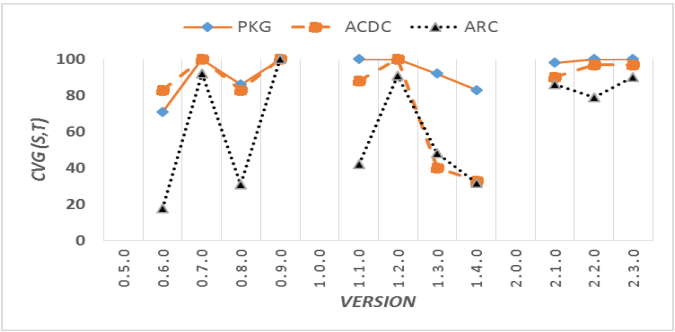

Fig. 3: $cvg(s,t)$ values between minor versions of Ivy

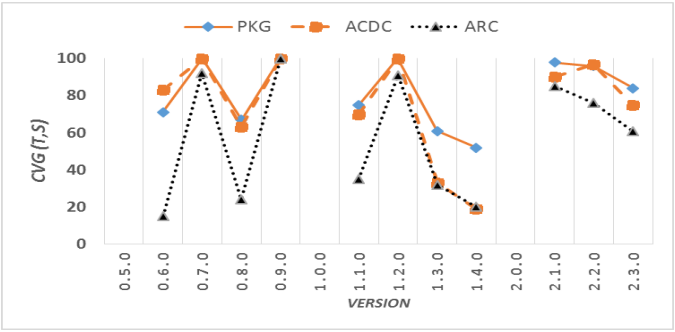

Fig. 4: $cvg(t,s)$ values between minor versions of Ivy

Figure 2 shows that the trends for *a2a* values involving Ivy's minor versions are similar among the three architectural views, with the *ARC* values generally slightly lower than the *ACDC* and *PKG* values. However, Figures 3 and 4 show that ARC reveals significant component-level changes for the same set of minor versions of Ivy.

To verify these and other similar results, we examined the changes that occurred in the involved versions. We found two key reasons for the lower *ARC* values, particularly at the component level: (1) class additions and (2) renaming of classes and variables. Classes were added, e.g., in Ivy's versions 0.6.0 and 0.8.0, indicating that the semantics of the affected components changed. However, these classes were mostly added to existing packages or components, resulting in a much smaller change to the architecture's structure. This type of semantic change at the component level is precisely the kind of change that the *cvg* values for *ARC* are intended to highlight. Furthermore, many classes and variables underwent refactoring across system versions (e.g., from *URLDownloader*

to *URLHandler* in Ivy). These are semantic rather than structural changes, and are more readily uncovered by *ARC* than in either of the structural views.

D. RQ4: Architectural Change in Consecutive Minor Versions

Our finding that major architectural change tends to involve major system versions was not surprising (although several of its facets, discussed above, were unexpected). In particular, we have found that a "jump" to a new major version (*MinMaj*) results in significant change, sometimes comparable to the cumulative sequence of changes experienced by a system across an entire major version. This can be seen in the *MinMaj* results in Tables II and III. These results also indicate that, on the average, a transition to a new major version involves more pronounced architectural changes than transitions between minor versions within the same major version.

TABLE IV: Minimum *a2a* values between minor versions

System	*ACDC*	*ARC*	*PKG*
ActiveMQ	86	78	84
Cassandra	60	55	49
Chukwa	72	73	76
Hadoop	57	51	72
Ivy	85	65	79
JackRabbit	76	69	74
Jena	83	77	86
JSPWiki	47	55	58
Log4j	62	62	59
Lucene	96	89	90
Mina	93	92	98
PDFBox	87	87	87
Struts2	79	80	83
Xerces	41	37	48
AVG	**73**	**69**	**75**

An interesting question we set out to explore in this study was whether this is always the case. In other words, can a system's architecture experience changes between two consecutive minor versions that are comparable to the changes between a minor version and the subsequent major version? To this end, we conducted an analysis to determine the minimum similarity among all consecutive minor version pairs within a major version. Table IV shows the *a2a* results of that analysis on architectures produced by *ACDC*, *ARC* and *PKG*.

Several values in the table indicate that considerable architecture change can indeed occur between two minor versions (e.g., 47% for *ACDC* in JSPWiki; 37% for *ARC* in Xerces; 49% for *PKG* in Cassandra). In some systems (e.g., Cassandra), the minimum *a2a* values between consecutive minor versions (60% for *ACDC*; 55% for *ARC*; 49% for *PKG*) are lower than the corresponding *MinMaj* values (80% for *ACDC*; 79% for *ARC*; 79% for *PKG*, as shown in Table II). The analogous analysis involving minimum *cvg* values shows similar results, but is elided due to space constraints. The main reason for this is that developers tended to add a large number of new features to a new minor version of a system, especially at the beginning of the system's life cycle. For example, Xerces more than doubled in size from version 1.0 to version 1.2, which is its next downloadable minor version.

Although significant architectural changes do happen between consecutive minor versions, they are not prevalent and tend to "cancel out" one another. This is illustrated by the case of Lucene, whose pair $(a2a_{Major}, a2a_{MinMaj})$ is (12%, 8%) for *ACDC* and (8%, 7%) for *ARC* (recall Table II). Looking into the code history of Lucene, we found that Lucene 3.6.0 contains packages that support backward-compatibility with versions 3.1.x and 3.3.x. Those packages were subsequently removed from version 4.0.0. This made $a2a(3.0.0, 4.0.0)$ higher than $a2a(3.6.0, 4.0.0)$ for the *ARC* and *ACDC* views of Lucene.

This was one of several findings that indicated that software engineers may be missing a crisply defined, shared intuition as to how and to what extent a software architecture changes as a system evolves. Such findings reveal that software developers may not always understand the architectural impact of their changes, and also that they do not consider that impact to be a relevant factor in their versioning scheme.

VI. Threats to Validity

We identify several potential threats to the validity of our results with their corresponding mitigating factors.

The key threats to **external validity** involve our subject systems. Although we used a *limited number of systems*, we selected them so that they vary along multiple dimensions, including application domain, number of versions, size, and time frame. The *different numbers of versions* analyzed per system pose another potential threat to validity. This is unavoidable, however, since some systems simply undergo more evolution than others. In order to mitigate this threat, we compared versions against each other based on type (major, minor, patch and pre-release). Another threat stems from the fact that we only analyzed *Apache systems* that use *Jira* as their issue repository and are implemented in *Java*. The diversity of the chosen systems helps to reduce this threat, as does the wide adoption of Apache software, Java, and Jira.

The **construct validity** of our study is mainly threatened by the accuracy of the *recovered architectural views* and of our *detection of architectural change*. To mitigate the first threat, we selected the two architecture recovery techniques, *ACDC* and *ARC*, that have demonstrated the greatest accuracy in our extensive comparative analysis of available techniques [15]. Furthermore, we complemented these techniques with *PKG*, which implements an objective measure of a system's "implementation architecture" [23]. These three techniques are developed independently of one another and use very different strategies for recovering an architecture. This, coupled with the fact that their results exhibit similar trends, helps to strengthen the confidence in our conclusions. To properly characterize architectural change between two versions, we created a new system-level change metric (i.e., a2a) and a new component-level change metric based on previously validated metric[15] (i.e., *cvg*). We have evaluated a2a by applying it on a large number of scenarios and manually inspecting its results, and by comparing it to the widely-used MoJoFM metric, especially in those cases when MoJoFM yielded counterintuitive results.

VII. Related Work

Software evolution has been studied extensively at the code level, dating back several decades (e.g., Lehman's laws [24]).

We will highlight a number of examples that have influenced our work. Godfrey and Tu [19] discovered that Linux's already large size did not prevent it from continuing to grow quickly. Eick et al. [14] found a reduction in modularity over the 15-year evolution of software for a telephone switching system. Murgia et al. [30] studied the evolution of Eclipse and Netbeans and found that 8%-20% of code-level entities contain about 80% of all bugs. While interesting, informative, and influential in our work, these studies do not examine the evolution of a software system's architecture.

A few studies [12], [31] have attempted to investigate architectural evolution. These studies are smaller in scope than our work in this paper. Additionally, unlike *ARCADE*'s use of structural and semantic architectural views, only one of these studies considers more than one architectural perspective—however, in that study, as well as several others, the chosen perspectives are arguably not architectural at all. Each study also differs from our work in other important ways.

D'Ambros et al. [12] present an approach for studying software evolution that focuses on the storage and visualization of evolution information at the code and architectural levels. Their study utilizes a different set of architectural metrics than ours, specifically targeted at their visualizations.

Nakamura et al. [31] present an architectural change metric based on structural distance, and apply it to 13 versions of four software systems. However, they define their metric on class dependency graphs, therefore measuring change at the level of a system's OO implementation rather than its architecture.

A group of studies by Bowers et al. [8], [9], [10] has treated implementation packages as architectural components in assessing the usefulness of metrics for balancing the number of components in a system and for measuring coupling between components. We considered both of these metrics for inclusion in *ARCADE*. We decided against including the balancing metric because our previous studies [15], [16] have indicated that *ACDC* and *ARC* obtain appropriate numbers of components in practice. We are currently studying the coupling metric and assessing its effectiveness in measuring recovered architectures.

Inspired by these studies, we have implemented and included *PKG* in our study as well. However, our recent work has shown that software architects consider the package structure to be useful but, by itself, an inaccurate architectural proxy [16]. We thus consider the results of Bowers et al.'s studies to be more indicative of implementation change than of architectural change. This is consistent with the widely referenced 4+1 architectural-view model [23], in which packages belong to a system's implementation view.

VIII. Conclusion and Future Work

This paper presented the largest empirical study to date of architectural change in long-lived software systems. The study's scope is reflected in the number of subject systems (14), the total number of examined system versions (572), the total amount of analyzed code (118.3 MSLOC), the number of applied architecture-recovery techniques (3) resulting in distinct architectural views produced for each system, the number of analyzed architectural models (1716, yielded by the three recovered views per system version), and the number of architectural change metrics (3) applied to each of the 1716 architectural models. This scope was enabled by *ARCADE*, a novel automated workbench for software architecture recovery and analysis.

Our study corroborated a number of widely held views about the times, frequency, scope, and nature of architectural change. However, the study also resulted in several unexpected findings. The foremost is that a system's versioning scheme is not an accurate indicator of architectural change: major architectural changes may happen between minor system versions. Even more revealing was the observation that a system's architecture may be relatively unstable in the run-up to a release. We believe that enabling engineers to spot such instability would go a long way toward stemming the types of developer habits that result in unstable, buggy system releases. Finally, our results further corroborated the observation made in recent interactions with practicing software architects [16] that the gross organization of a system's implementation is, by itself, not an adequate representation of the system's architecture. This is especially magnified in cases where the overall implementation architecture remained very stable while, in fact, the system experienced significant growth. For this reason, analyzing a system's recovered conceptual architecture, both at the level of overall structure and at the level of individual components, is a much more appropriate way of assessing and understanding architectural change.

Another broad conclusion of our study points to the significance of the semantics-based architectural perspective. We encountered multiple instances where a concern-based architectural view revealed important changes that remained concealed in the corresponding structure-based views. At the same time, a significant segment of the research of software architecture, and in particular the research of architecture recovery, has focused on system structure. Along with the results of our recent evaluation of recovery techniques [15], this suggests that there is both a need and an opportunity for investigating more effective approaches to architecture recovery.

ARCADE provides a powerful foundation for studying a wide variety of architectural phenomena as software systems evolve. Besides including additional subject systems, we are working to extend ARCADE to support other architectural constructs (e.g., component types, software connectors [34], their interfaces, and their concerns). We are currently complementing the study described in this paper with an analysis of the decay [33] found in architectures as they change over time. To this end, we have recently added six new metrics to *ARCADE* for measuring different aspects of architectural decay. We intend to use the analysis of decay as a springboard for improving our understanding of the relationship between architectural change and decay on the one hand, and the reported implementation issues on the other hand. Our long-term goal is to leverage *ARCADE* to enable *prediction* of architectural decay and major architectural change based on available implementation-level information.

REFERENCES

[1] apache-portable-runtime-versioning. http://apr.apache.org/versioning.html, 2014.

[2] arcade:start [USC SoftArch Wiki]. http://softarch.usc.edu/wiki/doku.php?id=arcade:start, 2014.

[3] git-log. http://git-scm.com/docs/git-log, 2014.

[4] hadoop-releases. http://hadoop.apache.org/releases.html#News, 2014.

[5] lucene-wiki. http://en.wikipedia.org/wiki/Lucene, 2014.

[6] struts-wiki. http://en.wikipedia.org/wiki/Apache_Struts, 2014.

[7] svn-graph-branches. https://code.google.com/p/svn-graph-branches/, 2014.

[8] E. Bouwers, J. P. Correia, A. van Deursen, and J. Visser. Quantifying the analyzability of software architectures. In *Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on*, pages 83–92. IEEE, 2011.

[9] E. Bouwers, A. v. Deursen, and J. Visser. Evaluating usefulness of software metrics: an industrial experience report. In *ICSE*, pages 921–930. IEEE Press, 2013.

[10] E. Bouwers, A. van Deursen, and J. Visser. Dependency profiles for software architecture evaluations. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 540–543. IEEE, 2011.

[11] A. Chatzigeorgiou and A. Manakos. Investigating the evolution of bad smells in object-oriented code. In *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*. IEEE, 2010.

[12] M. D'Ambros, H. Gall, M. Lanza, and M. Pinzger. *Analysing software repositories to understand software evolution*. Springer, 2008.

[13] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE TSE*, 2009.

[14] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE TSE*, 2001.

[15] J. Garcia, I. Ivkovic, and N. Medvidovic. A comparative analysis of software architecture recovery techniques. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 486–496, 2013.

[16] J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic. Obtaining ground-truth software architectures. *ICSE*, 2013.

[17] J. Garcia, I. Krka, N. Medvidovic, and C. Douglas. A framework for obtaining the ground-truth in architectural recovery. In *Joint Working IEEE/IFIP Conference on Software Architecture & 6th European Conference on Software Architecture (WICSA/ECSA)*, pages 292–296. IEEE, 2012.

[18] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai. Enhancing architectural recovery using concerns. In *ASE*, 2011.

[19] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Software Maintenance, 2000. Proceedings. International Conference on*. IEEE, 2000.

[20] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *ESEC/FSE*. ACM, 2005.

[21] R. Koschke. What Architects Should Know About Reverse Engineering and Rengineering. In *Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2005.

[22] R. Koschke. Architecture reconstruction. *Software Engineering*, 2009.

[23] P. B. Kruchten. The 4+ 1 view model of architecture. *Software, IEEE*, 1995.

[24] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 1980.

[25] T. Lutellier, D. Chollack, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger. Comparing software architecture recovery techniques using accurate dependencies. In *Proceedings of the 37th International Conference on Software Engineering (ISCE 2015), Software Engineering in Practice Track*, 2015.

[26] O. Maqbool and H. Babri. Hierarchical clustering for software architecture recovery. *IEEE TSE*, 2007.

[27] A. McCallum. Mallet: A machine learning for language toolkit. 2002.

[28] N. Medvidovic. Adls and dynamic architecture changes. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, 1996.

[29] J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, March 1957.

[30] A. Murgia, G. Concas, S. Pinna, R. Tonelli, and I. Turnu. Empirical study of software quality evolution in open source projects using agile practices. In *Proceedings of the First International Symposium on Emerging Trends in Software Metrics 2009*. Lulu. com, 2009.

[31] T. Nakamura and V. R. Basili. Metrics of software architecture changes based on structural distance. *Software Metrics, IEEE International Symposium on*, 0:8, 2005.

[32] M. N. Oreizy, P. and R. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on SoftwareEngineering (ICSE'98)*, 1998.

[33] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT SEN*, 1992.

[34] R. Taylor, N. Medvidovic, and E. Dashofy. Software Architecture: Foundations, Theory, and Practice. 2009.

[35] V. Tzerpos and R. Holt. ACDC: an algorithm for comprehension-driven clustering. In *Working Conference on Reverse Engineering (WCRE)*, 2000.

[36] A. Van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: View-driven software architecture reconstruction. In *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*. IEEE, 2004.

[37] Z. Wen and V. Tzerpos. An effectiveness measure for software clustering algorithms. In *International Workshop on Program Comprehension (IWPC)*. IEEE, 2004.