Noname manuscript No.
(will be inserted by the editor)

A Large-Scale Study of Architectural Evolution in Open-Source Software Systems

Pooyan Behnamghader * · **Duc Minh Le *** ·
Joshua Garcia · **Daniel Link** ·
Arman Shahbazian · **Nenad Medvidovic**

21 Jul 2016

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, sometimes careless 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 introducing an architecture recovery framework, *ARCADE*, for conducting large-scale replicable empirical studies of architectural change across different versions of a software system. *ARCADE* includes two novel architectural change metrics, which are the key to enabling large-scale empirical studies of architectural change. We utilize *ARCADE* to conduct an empirical study of changes found in software architectures spanning several hundred versions of 23 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 the system's 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.

Keywords Software architecture · architectural change · software evolution · open-source software · architecture recovery

Pooyan Behnamghader · Duc Minh Le · Daniel Link · Arman Shahbazian · Nenad Medvidovic
Computer Science Department
University of Southern California
Los Angeles, CA, USA
E-mail: {pbehnamg,ducmle,dlink,armansha,neno}@usc.edu

Joshua Garcia
Institute for Software Research
University of California, Irvine
Irvine, CA, USA
E-mail: joshua.garcia@uci.edu

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

1 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 (Taylor et al., 2009). 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 (Perry and Wolf, 1992). 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.

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 (Koschke, 2009; Ducasse and Pollet, 2009; Garcia et al., 2013a; Van Deursen et al., 2004; Maqbool et al., 2007), with the shared objective of analyzing a system's implementation in order to extract its architecture as a structured arrangement of clusters that are, in turn, composed of entities such as source files, classes, or methods. 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 (Garcia et al., 2013a). To evaluate their accuracy, we developed (Garcia et al., 2012) and applied (Garcia et al., 2013b) 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 have recently introduced our first attempt at addressing the above problem (Le et al., 2015). In this work, we outlined 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 (Lehman, 1980; Godfrey and Tu, 2000; Kim et al., 2005; Chatzigeorgiou and Manakos, 2010; Eick et al., 2001; Murgia et al., 2009), *ARCADE* represents the first solution of which we are aware that enables investigating such issues at the level of architecture. In this paper, we expand the empirical study, detail *ARCADE*'s architecture and implementation, and present two novel metrics we developed in order to enable the study.

We have employed *ARCADE* as part of an empirical study in which we analyzed several hundred versions of 23 open-source software 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*) (Tzerpos and Holt, 2000) and *Architecture Recovery using Concerns* (*ARC*) (Garcia et al., 2011)—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 (Kruchten, 1995). *ACDC* and *ARC* were chosen because they demonstrated better accuracy and scalability compared to other recovery techniques in our previous empirical evaluation (Garcia et al., 2013a). *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.

ARCADE is designed to conduct large-scale replicable empirical studies. We have developed a support framework, *ARCADE-Controller*, that allows a software architect to define a workflow for architecture recovery analysis and to distribute that analysis over a set of nodes in the computing. *ARCADE-Controller* rapidly recovers the architectures of many versions and revisions of a system using multiple cloud instances simultaneously. *ARCADE-Controller* then transfers the recovered architectures to an analysis server, allowing an engineer to run evolutionary analyses, such as architectural change analysis.

To measure architectural changes across the development history of a software system, we introduce two new architecture similarity metrics: *cvg* and *a2a*. *cvg* is a metric that computes the similarity between two architectures based on the constituent components of each architecture. *a2a* is a system-level similarity metric calculated based on the cost of transforming one architecture to another. In order to compute the minimum transforming cost between an architecture of two different versions of the same software system, we introduce a new architecture distance metric *mto* and present an algorithm to calculate it.

This paper significantly extends our previous empirical study of architectural change of open-source systems (Le et al., 2015). The extensions include the following: the algorithm for computing *a2a*, proofs of mathematical properties of *a2a* and *cvg*, a description of *ARCADE-Controller*, and an expanded empirical study that comprises an additional set of 9 subject systems on top of the 14 systems studied in (Le et al., 2015). The reader is now able to study the introduced metrics, evaluate them for her needs, and extend and modify them if necessary. Furthermore, the addition of large numbers of versions of the 9 new systems enabled us to further confirm many of our previous conclusions, but also to properly qualify some of them. An example, discussed in detail below, is a trend involving a system's pre-releases versus its patch versions: the additional data confirmed the overall relationship we had observed before, but introduced several instances that serve as counter-examples to our previously reported trend. We have qualified the corresponding conclusions.

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 (Ducasse and Pollet, 2009; Garcia et al., 2013a; Koschke, 2005, 2009; Maqbool et al., 2007). 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 (Kruchten, 1995)), 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 2 summarizes the two related research threads that have been brought together to enable the work described in this paper. Section 3 explains *a2a* and provides proofs of its properties. Section 4 introduces the *cvg* metric, its properties, and its implications. Section 5 presents the details of the *ARCADE* workbench. Section 6 describes the setup for our empirical study, Section 7 its key results, and Section 8 the threats to its validity. A discussion of related work (Section 9) and conclusions (Section 10) round out the paper.

This paper describes our work in its entirety. However, it would be possible for a reader to understand the details of the empirical study even if she skipped Sections 3 and 4, which detail the properties of the two metrics. Likewise, a reader may choose to skip Section 5, which describes the architecture and implementation of our workbench used to obtain the data in the study.

2 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 *a2a* and *cvg* in Sections 3 and 4, and the *ARCADE* workbench in Section 5, 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.

2.1 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 similarity metrics for our study: *a2a*, a system-level metric, and *cvg*, a component-level metric. In order to calculate *a2a*, we introduce a new architecture distance metric, *mto*. We will also describe a metric, *c2c* (Garcia et al., 2013a), because it enables the computation of *cvg*. Similarity metrics *a2a* and *cvg* have been recently used in a study of the impact of the granularity of module dependencies on the quality of architecture recovery (Lutellier et al., 2015).

Architecture-to-architecture (*a2a*) is a similarity metric we developed for assessing system-level change. *a2a* was inspired by the widely used MoJo (Tzerpos and Holt, 1999) and MoJoFM metrics (Wen and Tzerpos, 2004). Neither MoJo nor MoJoFM is intended or designed for a study of architectural evolution of the type attempted here: MoJo is a heuristic distance metric intended to determine the similarity between two different architectures with the same set of implementation-level entities (Tzerpos and Holt, 1999); MoJoFM is an effectiveness measure for software clustering algorithms based on MoJo intended to compare a recovered architecture with a ground-truth architecture (Wen and Tzerpos, 2004). MoJoFM proved to be ill-suited for our study because it assumes that the entity sets in the architectures (depending on the recovery method used, entities may be classes, methods or other building blocks of a system) undergoing comparison will be identical; this is unrealistic for systems whose versions are known to have evolved, sometimes substantially. In order to address this shortcoming, we introduce *mto*, a distance metric that measures distance between two architectures with arbitrary entity sets, then normalize it to calculate *a2a*.

Minimum-transform-operation (*mto*) is the minimum number of operations needed to transform one architecture to another:

$$\begin{aligned} mto(A_1,A_2) = {} & remC(A_1,A_2) + addC(A_1,A_2) \\ & + remE(A_1,A_2) + addE(A_1,A_2) + movE(A_1,A_2) \end{aligned} \tag{1}$$

The five operations used to transform architecture A_1 into A_2 comprise 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 (Agnew et al., 1994; Medvidovic, 1996; Oreizy et al., 1998).

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. This is supported by several foundational works on architectural adaptation (e.g., Agnew et al. (1994); Medvidovic (1996); Oreizy et al. (1998)). The underlying intuition is as follows. If we think of the recovered architecture as a set of constituent building blocks (i.e., clusters and entities) and their configurations (i.e., arrangement of entities inside clusters), then there is a difference between (a) simply changing the architectural configuration and (b) also changing the constituent building blocks.

We normalize *mto* to calculate *a2a*, a similarity metric between two architectures with different implementation-level entities:

$$a2a(A_1,A_2) = (1 - \frac{mto(A_1,A_2)}{mto(A_\emptyset,A_1) + mto(A_\emptyset,A_2)}) \times 100\% \tag{2}$$

where $mto(A_{\emptyset},A_i)$ is the number of operations required to transform a "null" architecture $A_{\emptyset}$ into A_i. In other words, the denominator $mto(A_{\emptyset},A_1)+mto(A_{\emptyset},A_2)$ is the number of operations needed to construct architectures A_1 and A_2 from a "null" architecture. This approach is inspired by the foundational work on architectural adaptation cited above, and is further discussed in Section 3.5.

Section 3 presents the algorithm calculating *mto* and consequently *a2a*.

Cluster coverage (*cvg*) is a new similarity metric we have developed to indicate the extent to which two architectures' clusters overlap. In other words, *cvg* allows engineers to determine the extent to which certain components existed in an earlier version of a system or were added in a later version:

$$cvg(A_1,A_2) = \frac{|simC(A_1,A_2)|}{|C_{A_1}|} \times 100\% \tag{3}$$

where $|C_{A_1}|$ is the number of clusters in architecture A_1.

$simC(A_1,A_2)$ returns the subset of A_1 clusters that have at least one "similar" cluster in A_2:

$$simC(A_1,A_2) = \{c_i \mid c_i \in A_1,\ \exists c_j \in A_2,\ c2c(c_i,c_j) > th_{cvg}\} \tag{4}$$

where *c2c* (Garcia et al., 2013a) measures the degree of overlap between the implementation-level entities contained within two clusters. 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. Section 4 explains *cvg* in detail.

2.2 Architecture Recovery Tool Suite

We recently conducted a comparative evaluation of software architecture recovery techniques (Garcia et al., 2013a). 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 (Garcia et al., 2013b). To that end, we implemented a tool suite offering a large set of architecture recovery choices to an engineer.

Our study showed 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 in this paper. These two techniques—*ACDC* (Tzerpos and Holt, 2000) and *ARC* (Garcia et al., 2011)—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* (Tzerpos and Holt, 2000) groups enty ities 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* (Garcia et al., 2011) groups entities that handle similar system concerns into a single cluster. For instance, *ARC* may group together the entities that handle user interface behaviors. In the study described in this paper, 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" (Kruchten, 1995); while not indicative of the actual architecture underlying the system (Taylor et al., 2009), the package structure provides a useful baseline (a "sanity check") for our study.

3 Architecture-to-Architecture (*a2a*)

This section describes the *a2a* metric and its properties. A reader may choose to skip the section if she is not interested in the details contained therein. While the empirical study described in the remainder of the paper was enabled by *a2a* in a critical way, understanding the obtained data and its interpretation does not require understanding the details of the metric itself.

In order to demonstrate that $a2a(A,B)$ is guaranteed to give a value between 0% and 100%, in this section we explain that (1) our algorithm for calculating $mto(A,B)$ is optimal; and (2) *mto* satisfies metric axioms (Shirali and Vasudeva, 2005) guaranteeing that $mto(A_\emptyset,A) + mto(A_\emptyset,B) \geq mto(A,B)$. We also explain that the denominator $mto(A_\emptyset,A) + mto(A_\emptyset,B)$ is not over-weighted, meaning that *a2a* returns a full range of values between 0% and 100% for every architecture. We first introduce a case study in order to illustrate different facets of *a2a*.

3.1 Case Study

Assume V_1 and V_2 are two versions of an evolving software system S. Version V_1 contains n_1 implementation-level entities and A is a recovered architecture of V_1 with m clusters, A=$\{a_1,a_2,...,a_m\}$. Version V_2 contains n_2 entities and B is an architecture of V_2 with l clusters, B=$\{b_1,b_2,...,b_l\}$. An example is shown in Figure 1 to explain the calculation of $a2a(A,B)$. In this example, A is a recovered architecture of V_1 with 14 entities, and B is a recovered architecture of V_2 with 15 entities. During the evolution of S from V_1 to V_2, 10 entities remain in the system, 4 entities are removed, and 5 new entities are introduced (added).

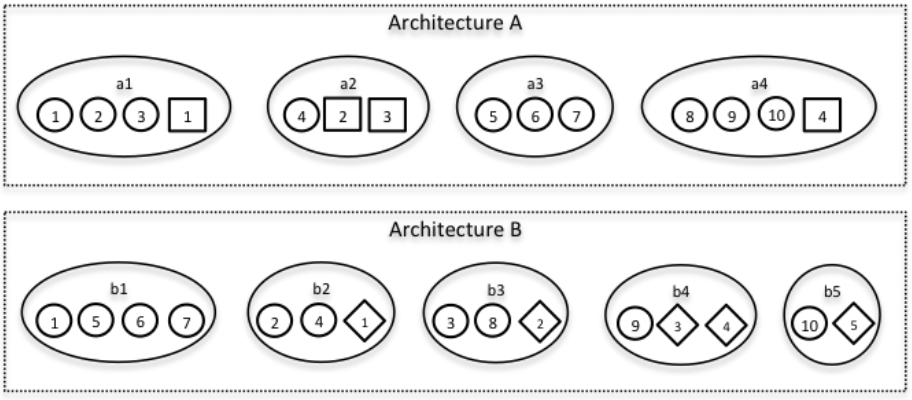

Fig. 1: Architecture A contains 4 clusters and 14 entities. Architecture B contains 5 clusters and 15 entities. To transform A to B, 4 entities (*squares*) are removed, 5 new entities (*diamonds*) are introduced, and 10 entities (*circles*) remain in the system.

3.2 Calculating $mto(A_\emptyset, A)$ and $mto(A_\emptyset, B)$

$mto(A_\emptyset, A)$ is the minimum number of operations required to construct architecture A from a "null" architecture $A_\emptyset$:

$$mto(A_\emptyset, A) = addC(A_\emptyset, A) + addE(A_\emptyset, A)$$

In order to calculate $mto(A_\emptyset, A)$, our algorithm adds all m clusters of A to $A_\emptyset$, $addC(A_\emptyset, A) = m$. Afterwards, it first adds all n_1 entities of A to a "dummy" cluster and then moves each of them to the appropriate cluster, $addE(A_\emptyset, A) = 2 \times n_1$. Therefore, the minimum value for $mto(A_\emptyset, A)$ is:

$$mto(A_\emptyset, A) = m + 2 \times n_1 = |C_A| + 2 \times |E_A|$$

where $|E_A|$ is the number of entities, and $|C_A|$ is the number of clusters in architecture A. Similarly, our algorithm constructs B from $A_\emptyset$ then $mto(A_\emptyset, B) = l + 2 \times n_2 = |C_B| + 2 \times |E_B|$. Figure 2 illustrates the aforementioned steps in constructing architecture A from $A_\emptyset$ in our example. In this case, $mto(A_\emptyset, A) = 4 + 2 \times 14 = 32$ and $mto(A_\emptyset, B) = 5 + 2 \times 15 = 35$.

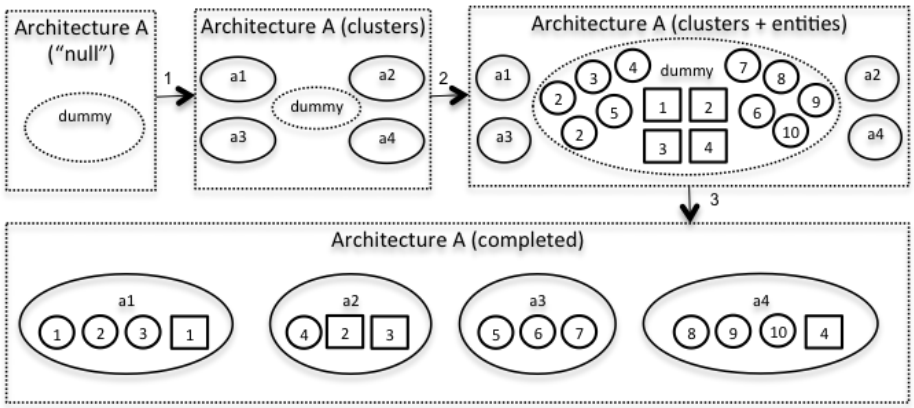

Fig. 2: Constructing architecture A from a "null" architecture: adding (1) clusters and (2) entities, then (3) moving entites into the corresponing clusters.

3.3 Calculating $mto(A, B)$

$mto(A, B)$ is the minimum number of operations required to transform architecture A into architecture B:

$$mto(A, B) = remE(A, B) + remC(A, B) + addC(A, B) + movE(A, B) + addE(A, B)$$

where $remE$, $remC$, $addC$, $movE$, and $addE$ are five operations required to transform one architecture to another. $remE(A, B)$ is the number of operations required to prune architecture A from entities that do not exist in B. $remC(A, B) + addC(A, B)$ is the number of operations required to equalize the number of clusters in A with clusters of B. There might be some entities existing in both A and B, especially when A and B are the architectures of different

versions of the same software system. $movE(A,B)$ is the minimum number of operations required to move those common entities in transforming A to B. Finally, $addE(A,B)$ is the number of operations needed to add entities missing in A but that need to be present in B.

Our algorithm calculates the minimum value for $remE(A,B)$, $remC(A,B)+addC(A,B)$, $movE(A,B)$, and $addE(A,B)$ in order to calculate $mto(A,B)$. Each of these is discussed below.

3.3.1 Ensuring Minimum Value for $remE(A,B)$

In order to transform architecture A into B, we need to remove all entities of A that do not exist in B. Therefore, the minimum value for $remE(A,B)$ is:

$$remE(A,B) = 2 \times |E_A \setminus E_B| \qquad E_A \setminus E_B := \{e \mid e \in E_A \wedge e \notin E_B\}$$

$remE(A,B)$ is obtained by computing the following two operations on each entity e_{rem} in A that needs to be removed: (1) moving e_{rem} out of its current cluster to a "dummy" cluster and (2) deleting e_{rem} from the architecture. If $E_A \subset E_B$—meaning that no entity is removed from the system—then $remE(A,B) = 0$. The highest possible minimum value for $remE(A,B) = 2 \times |E_A|$ when $E_A \cap E_B = \emptyset$. In our example, the mto algorithm removes four entities from A that do not exist in B, so that $remE(A,B) = 2 \times |E_A \setminus E_B| = 2 \times 4 = 8$. Figure 3 illustrates the process of removing entities.

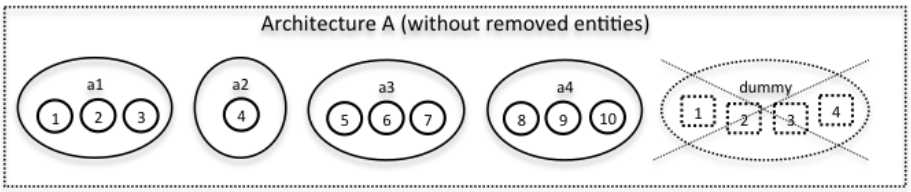

Fig. 3: The mto algorithm first moves four entities to be removed from architecture A to a "dummy" cluster, then removes that cluster and the four entities.

3.3.2 Ensuring Minimum Value for $(remC(A,B)+addC(A,B))$

In the process of transforming architecture A into B, the number of clusters in the transformed architecture should match the number of clusters in B. Therefore, the minimum number of clusters that we need to add or delete for transforming A into B is as follows:

$$remC(A,B)+addC(A,B) = abs(m-l) = abs(|C_A|-|C_B|)$$

If $m < l$, our algorithm adds $l-m$ clusters to A before performing any further entity-related operation ($addE$ and $movE$) in order to guarantee the same number of clusters in A and B. On the other hand if $m > l$ our algorithm removes $m-l$ clusters from A after performing all $addE$ and $movE$ operations. The number of clusters in our example is increased from four clusters in A to five clusters in B. Therefore, $remC(A,B)+addC(A,B) = abs(|C_A|-|C_B|) = abs(4-5) = 1$. Figure 4 shows that the new cluster a_5 is added to architecture A.

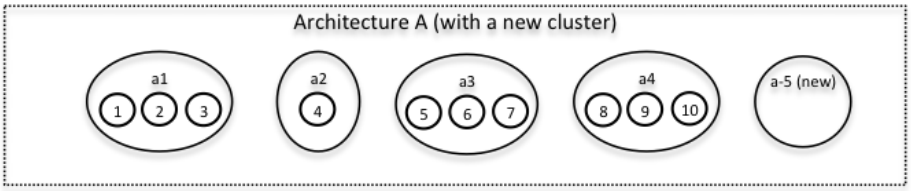

Fig. 4: To have the same number of clusters in A and B, the mto algorithm adds an empty cluster a_5 to A.

3.3.3 Ensuring Minimum Value for $movE(A,B)$

In order to ensure that the minimal number of move operations $movE(A,B)$ is applied in calculating mto, we have implemented an algorithm that finds the largest subset of elements remaining in the same cluster when transforming A into B. Our algorithm constructs a balanced, weighted bipartite graph G that comprises the clusters of A and B. Note that if $m < l$, then our algorithm adds $l-m$ empty clusters $(a_{m+1},...,a_l)$ to A and finds exactly one matching cluster $a_i \in A$ for every cluster $b_j \in B$.

The weight of the edge between two clusters of G is the number of their common entities:

$$G = (U,V,E) \quad U = C_A \cup \{a_{m+1},...,a_l\} \quad V = C_B$$
$$E = \{(a_i,b_j,|a_i \cap b_j|) \mid a_i \in U, b_j \in V\}$$

The mto algorithm uses the Hungarian algorithm (Munkres, 1957) to find the maximum weighted matching in G. The Hungarian algorithm is a combinatorial optimization algorithm which solves the minimum (or maximum) weighted matching problem (i.e., the assignment problem) in polynomial time.

The result of performing our algorithm on G is a set of tuples M:

$$M = \{(a_i,b_j,E_i) \mid a_i \in U, b_j \in V, E_i = a_i \cap b_j\}$$

where a_i and b_j are matched clusters, and E_i is the set of matching entities of a_i that do not move when transforming A into B. In our example, the $a2a$ algorithm constructs a weighted bipartite graph for architectures A and B as follows:

$$G = (U,V,E) \quad U = \{a_1,a_2,a_3,a_4\} \cup \{a_5\} \quad V = \{b_1,b_2,b_3,b_4,b_5\}$$

$$E = \{(a_1,b_1,1),(a_1,b_2,1),(a_1,b_3,1),(a_2,b_2,1),(a_3,b_1,3),(a_4,b_3,1),(a_4,b_4,1),(a_4,b_5,1)\}$$

Figure 5 shows a maximum weighted matching M for G as follows:

$$M = \{(a_1,b_3,\{ce_3\}),(a_2,b_2,\{ce_4\}),(a_3,b_1,\{ce_5,ce_6,ce_7\}),(a_4,b_4,\{ce_9\})\}$$

where ce_i stands for entity $circle_i$.

After finding the matching clusters and entities, our algorithm moves the rest of the entities to their new appropriate clusters. $(a_i \cap E_B) \setminus E_i$ is the set of entities that exist in both a_i and E_B, and move from a_i to another component in transforming A into B. For each $e \in (a_i \cap E_B) \setminus E_i$ there exists $b_e \in V$ and a new appropriate cluster $a_k \in U$ where (1) $e \in b_e$ and (2) a_k is the matching cluster of b_e. For example, for $ce_1 \in (a_1 \cap E_B) \setminus \{ce_3\}$, a_3 is the new appropriate cluster since (1) $ce_1 \in b_1$ and (2) a_3 is the matching cluster of b_1. Our algorithm moves e from a_i to a_k with exactly one operation.

Therefore, the minimum value for $movE(A,B)$ is computed as follows:

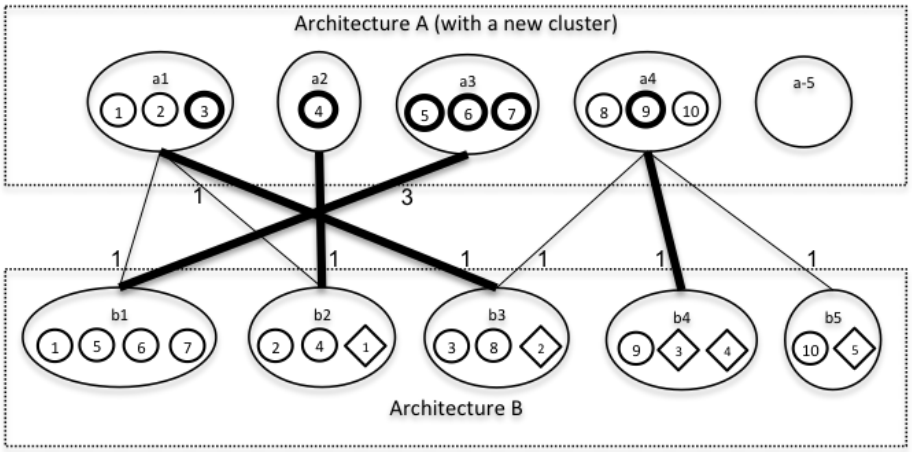

Fig. 5: A maximum weighted matching between clusters of architectures A and B. Bold lines and entities ilustrate matching clusters and entities. No lines means no common entity.

$$(*): \ movE(A,B) = \left| \bigcup_{i=1}^{l} (a_i \cap E_B) \setminus E_i \right| \qquad (**): \ E_i \subset a_i \ \ and \ \ a_i \cap a_j = \emptyset$$

$$(*) \ \wedge \ (**) \ \Rightarrow \quad movE(A,B) = \left| (E_A \cap E_B) \setminus \bigcup_{i=1}^{l} E_i \right| = |E_A \cap E_B| - \left| \bigcup_{i=1}^{l} E_i \right|$$

For simplicity of the formula, we call the set of all matching entities E_{fix} which is the set of all entities in A that do not move in transforming A into B. Therefore, the formula of $movE(A,B)$ is as follows:

$$movE(A,B) = |E_A \cap E_B| - |E_{fix}|$$

In our example:

$$E_A \cap E_B = \{ce_i : 1 \leq i \leq 10\} \quad and \quad E_{fix} = \{ce_3, ce_4, ce_5, ce_6, ce_7, ce_9\}$$

which means that 6 out of 10 circle entities stay in their original clusters, while the other 4 move to new appropriate clusters. Our algorithm moves every entity $e \in (E_A \cap E_B) \setminus E_{fix}$ to its new corresponding cluster with exactly one operation, $movE(A,B) = 10 - 6 = 4$. The new architecture is architecture B minus the new entities. Figure 6 shows the new architecture resulting from the move operations.

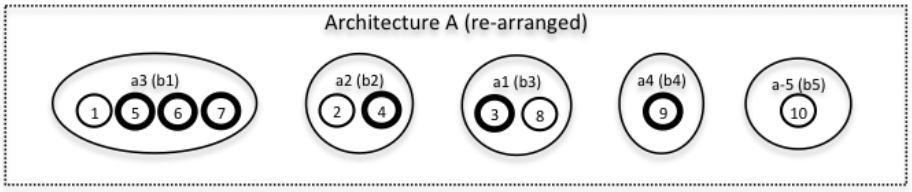

Fig. 6: Performing move operations on the entities of A after finding the appropriate clusters

This matching between clusters of the two architectures maximizes the number of matched elements, and consequently minimizes $movE(A,B)$. Let us assume that there exists

another algorithm to transform A into B with fewer move operations than our algorithm; in fact, it introduces a new set of elements $E_{fix}\prime$ that remain in their original clusters when transforming A into B; and $E_{fix}\prime$ is larger than E_{fix}. This means that the new algorithm introduces a new matching between clusters of A and B with more common entities than our matching, which contradicts our assumption that E_{fix} is the entity matching set resulting from the maximum weighted matching algorithm. We note that the Hungarian algorithm is guaranteed to find the maximum weighted matching in graph G.

3.3.4 Ensuring Minimum Value for $addE(A,B)$

Finally, for each newly introduced entity e_{new} in architecture B, the mto algorithm performs two operations: (1) adding e_{new} to a "dummy" cluster and (2) moving e_{new} to the appropriate cluster. Therefore, the minimum value for $addE(A,B)$ is computed as follows:

$$addE(A,B) = 2 \times |E_B \setminus E_A|$$

If $E_B \subset E_A$—meaning no entity is introduced to the system—then $addE(A,B) = 0$. The highest possible minimum value for $addE(A,B) = 2 \times |E_B|$ when $E_A \cap E_B = \emptyset$.

In our example, the mto algorithm adds every entity $e \in E_B$ that does not exist in E_A to the architecture, $addE(A,B) = 2 \times |E_B \setminus E_A| = 10$. The new architecture shown in Figure 7 is an isomorph of architecture B, i.e., each entity or each cluster in the new architecture has one and only one equivalent in architecture B.

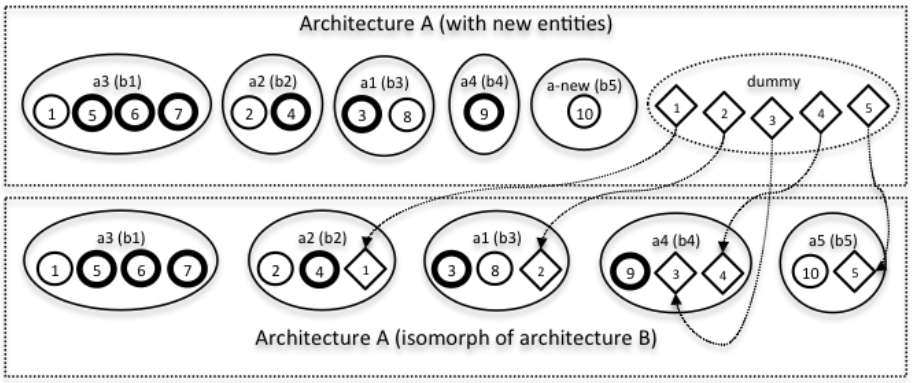

Fig. 7: Adding each new entity to a "dummy" cluster, then moving it to the appropriate cluster. The new architecture is an isomorph of architecture B.

3.3.5 Determining the Value of mto

Based on the minimum values calculated for $remE(A,B)$, $remC(A,B)+addC(A,B)$, $movE(A,B)$, and $addE(A,B)$, $mto(A,B)$ is derived as follows:

$$mto(A,B) = abs(|C_A| - |C_B|) + 2 \times |E_A \setminus E_B| + 2 \times |E_B \setminus E_A| + |E_A \cap E_B| - |E_{fix}|$$

In our example, the minimum cost for transforming architecture A into an isomorph of architecture B in our example is:

$$mto(A,B) = 1+8+10+4 = 23$$

This means that 23 operations are required to transform A into B.

Algorithm 1 presents the pseudocode of mto.

Algorithm 1: $mto(A,B)$

Input : Two architectures $A = \{a_1, a_2, ..., a_m\}$ and $B = \{b_1, b_2, ..., b_l\}$
Output : An architecture resulted from transforming architecture A into B

$m, l \leftarrow |C_A|, |C_B|$
//$remE(A,B)$
foreach $e \in (E_A \setminus E_B)$ **do**
    move e to a "dummy" cluster
    remove e from A
if $m < l$ **then**
    //$addC(A,B)$
    add $\{a_{m+1}, ..., a_l\}$ to A
//The algorithm $movE(A,B)$ is presented in Algorithm 2.
movE(A,B)
//$addE(A,B)$
foreach $e \in (E_B \setminus E_A)$ **do**
    add e to a "dummy" cluster in A
    move e from the "dummy" cluster to the appropriate cluster in A
if $m > l$ **then**
    //$remC(A,B)$
    remove $m - l$ empty clusters from A
return A

Algorithm 2: $movE(A,B)$

Input : Two architecture $A = \{a_1, a_2, ..., a_l\}$ and $B = \{b_1, b_2, ..., b_l\}$ with the same number of clusters
Output : An architecture resulted from moving entities in architecture A

$U, V \leftarrow \{a_1, a_2, ..., a_l\}, \{b_1, b_2, ..., b_l\}$
$E \leftarrow \{ (a_i, b_j, |a_i \cap b_j|) : a_i \in U, b_j \in V \}$
$G \leftarrow (U, V, E)$
//M (Maximum Weighted Matching) is the result of the Hungarian algorithm in the form of $\{ (a_i, b_j, E_i) : a_i \in U, b_j \in V, E_i = a_i \cap b_j \}$
$M \leftarrow HungarianAlgorithm(G, A, B)$
foreach $(a_i, b_j, E_i) \in M$ **do**
    foreach $e \in (a_i \cap E_B) \setminus E_i$ **do**
        //a_k is the matching cluster of b_e, and e belongs to b_e in B
        $a_k \leftarrow$ where $(a_k, b_e, E_k) \in M$ and $e \in b_e$
        move e from a_i to a_k with one operation
return A

3.4 Calculating $a2a(A,B)$

Once $mto(A_\emptyset,A)$, $mto(A_\emptyset,B)$, and $mto(A,B)$ are calculated, $a2a(A,B)$ is computed as follows:

$$a2a(A,B) = (1 - \frac{mto(A,B)}{mto(A_\emptyset,A) + mto(A_\emptyset,B)}) \times 100\%$$

$$= (1 - \frac{abs(|C_A| - |C_B|) + 2 \times |E_A \setminus E_B| + 2 \times |E_B \setminus E_A| + |E_A \cap E_B| - |E_{fix}|}{|C_A| + 2 \times |E_A| + |C_A| + 2 \times |E_B|}) \times 100\%$$

In our example, $a2a(A,B)$ is as follows:

$$a2a(A,B) = (1 - \frac{23}{32+35}) \times 100\% = 65.7\%$$

3.5 Properties of $a2a$ and mto

The similarity metric $a2a$ between architectures A and B is calculated by normalizing the numerator $mto(A,B)$ with the denominator $mto(A_\emptyset,A) + mto(A_\emptyset,B)$. In order for it to be a meaningful similarity metric for our study, we expect $a2a$ to satisfy the following properties. These properties are described in terms of the four axioms of metric space (Shirali and Vasudeva, 2005): non-negativity, coincidence, symmetry, and triangle inequality.

1. $a2a(A,B)$ must be guaranteed to return a value between 0% and 100%.

   $$\forall\, A\ ,\ B\ :\quad 100\% \geq a2a(A,B) \geq 0\%$$

   This means that the numerator $mto(A,B)$ must be positive and the denominator $mto(A_\emptyset,A) + mto(A_\emptyset,B)$ must always be greater than or equal to $mto(A,B)$. We prove that $a2a$ satisfies this property by showing that the **non-negativity**, **symmetry** and **triangle inequality** properties hold for the distance metric mto.
2. For identical architectures, $a2a$ must return 100% similarity. Also, if $a2a$ returns 100% similarity for two architectures, they must be identical.

   $$\forall\, A\ ,\ B\ :\quad a2a(A,B) = 100\% \quad \Leftrightarrow \quad A = B$$

   From the formula, $a2a(A,A)$ equals 100% if and only if $mto(A,A) = 0$. We prove that $a2a$ satisfies this property by showing that the **coincidence** property holds for mto.
3. For any architecture A, $a2a$ must return 0% similarity between a "null" architecture $A_\emptyset$ and A. Also if $a2a$ returns 0% similarity for two architectures, exactly one of them should be $A_\emptyset$.

   $$\forall\, A \neq A_\emptyset\ ,\ B\ :\quad a2a(A,B) = 0\% \quad \Leftrightarrow \quad B = A_\emptyset$$

   We prove that $a2a$ satisfies this property using the symmetry and triangle inequality properties of mto.
4. Similarity metric $a2a$ must be symmetric.

   $$\forall\, A\ ,\ B\ :\quad a2a(A,B) = a2a(B,A)$$

   We prove that $a2a$ satisfies this property using the symmetry property of mto.

In the remainder of this subsection, we show that the four metric space axioms (Shirali and Vasudeva, 2005) hold for mto and that $a2a$ satisfies the desired properties.

3.5.1 Metric Space Axioms for mto

One of the contributions of this paper is to define a meaningful distance metric between architectures of different versions of a software system. Many statistical analysis and machine learning algorithms critically rely on the distance metric given over their inputs (Xing et al., 2002). Defining a good metric that reflects the importance of relationship between data highly affects the performance of learning and data-mining algorithms, as well as the meaning of their results. We describe some basic axioms for metric spaces and distances; and illustrate that *mto* satisfies these axioms.

According to (Shirali and Vasudeva, 2005), a nonempty set X with a map $m : X \times X \to R$ is called a *metric space* if the map m satisfies following properties:

$(P1)$ $m(x,y) \geq 0 \quad x,\ y \in X;$ (non-negativity)
$(P2)$ $m(x,y) = 0 \Leftrightarrow x = y;$ (coincidence)
$(P3)$ $m(x,y) = m(y,x) \quad x,\ y \in X;$ (symmetry)
$(P4)$ $m(x,y) \leq m(x,z) + m(z,y) \quad x,\ y,\ z \in X;$ (triangle inequality)

The map m is called the distance function or metric on X. Introducing such a map m (distance) on set X (metric space) means that distances between all members of X are defined. We illustrate that set $Arcs = \{all\ possible\ architectures\}$ is a metric space, and map *mto* is the distance metric of *Arcs*:

$$mto(A,B) = remE(A,B) + remC(A,B) + addC(A,B) + movE(A,B) + addE(A,B)$$

1. Non-negativity:

$$mto(A,B) \geq 0 \quad A,\ B \in Arcs = \{all\ possible\ architectures\}$$

$$remE(A,B) \geq 0 \ \wedge \ addC(A,B) \geq 0 \ \wedge \ movE(A,B) \geq 0 \ \wedge \ addE(A,B) \geq 0$$
$$\wedge \ remC(A,B) \geq 0 \quad \Rightarrow \quad mto(A,B) \geq 0 \quad (sum\ of\ non-negative\ numbers)$$

2. Coincidence:

$$mto(A,B) = 0 \quad \Leftrightarrow \quad A = B$$

$$(I) \quad if \ A = B \quad \Rightarrow \quad mto(A,B) = addC(A,A) + remC(A,A)$$
$$+ addE(A,A) + remC(A,A) + movE(A,A) = 0+0+0+0+0 = 0$$

$$(II) \quad if \ mto(A,B) = 0 \quad \Rightarrow \quad addC(A,B) = remC(A,B)$$
$$= addE(A,B) = remC(A,B) = movE(A,B) = 0 \quad (all\ non-negative\ numbers)$$
$$\Rightarrow \quad No\ change\ in\ the\ system \quad \Rightarrow \quad A = B$$

3. Symmetry:

$$mto(A,B) = mto(B,A)$$

$$(I) \quad remC(A,B) + addC(A,B) = abs(|C_A| - |C_B|)$$
$$= abs(|C_B| - |C_A|) = addC(B,A) + remC(B,A)$$

$$(II) \quad remE(A,B) + addE(A,B) = 2 \times |E_A - E_B|$$
$$+2 \times |E_B - E_A| = addE(B,A) + remE(B,A)$$

(III) $movE(A,B) = movE(B,A)$ *since the maximum weighted matching is calculated on the same "undirected" graph for* $movE(A,B)$ *and* $movE(B,A)$

Note that the empty clusters added to architecture A_i with smaller number of clusters have no impact on the result of the matching algorithm since they contain no entity.

$$(I) \quad \wedge \quad (II) \quad \wedge \quad (III) \quad \Rightarrow \quad mto(A,B) = mto(B,A)$$

4. Triangle inequality:

$$mto(A,C) \quad \leq \quad mto(A,B) \; + \; mto(B,C)$$

Suppose there exists an architecture B such that $mto(A,B)+mto(B,C) < mto(A,C)$. This means that there exist a new way to transform A into C $(A \rightarrow B \rightarrow C)$ which costs less operations than $mto(A,C)$. This contradicts our assumption that $mto(A,C)$ is the minimum number of operations required to transform A into C, which we already proved.

Therefore, the set $Arch = \{\text{all possible architectures}\}$ is a metric space with a distance function (metric), mto.

3.5.2 Proof of desired properties for a2a

We proved that the properties non-negativity, coincidence, symmetry, and triangle inequality hold for mto. Now, we illustrate that $a2a$ satisfies our desired properties:

$$a2a(A,B) = (1 - \frac{mto(A,B)}{mto(A_\emptyset,A) + mto(A_\emptyset,B)}) \times 100\%$$

1. $\forall A, B: \quad 100\% \geq a2a(A,B) \geq 0\%$

$$mto(A_\emptyset,A) + mto(A_\emptyset,B) = mto(A,A_\emptyset) + mto(A_\emptyset,B) \geq mto(A,B) \geq 0$$

$$\Rightarrow \quad 1 \geq \frac{mto(A,B)}{mto(A_\emptyset,A) + mto(A_\emptyset,B)} \geq 0$$

$$\Rightarrow \quad 100\% \geq (1 - \frac{mto(A,B)}{mto(A_\emptyset,A) + mto(A_\emptyset,B)}) \times 100\% \geq 0\%$$

Therefore, $a2a$ is always guaranteed to return a value between 0% and 100%.

2. $\forall A, B: \quad a2a(A,B) = 100\% \quad \Leftrightarrow \quad A = B$

(I) $if \;\; A = B \quad \Rightarrow \quad a2a(A,B) = a2a(A,A)$

$$= (1 - \frac{mto(A,A)}{mto(A_\emptyset,A) + mto(A_\emptyset,A)}) \times 100\% = (1 - \frac{0}{2 \times mto(A_\emptyset,A)}) \times 100\% = 100\%$$

(II) $if \;\; a2a(A,B) = 100\%$

$$\Rightarrow \quad \frac{mto(A,B)}{mto(A_\emptyset,A) + mto(A_\emptyset,B)} = 0 \;\; \Rightarrow \;\; mto(A,B) = 0 \;\; \Rightarrow \;\; A = B$$

Therefore, $a2a$ returns 100% if and only if two architecture are identical.

3. $\forall A \neq A_\emptyset\ ,\ B\ :\quad a2a(A,B) = 0\% \quad \Leftrightarrow \quad B = A_\emptyset$

$(I)\quad if\ \ B = A_\emptyset \quad \Rightarrow \quad a2a(A,B) = a2a(A,A_\emptyset)$

$$= (1 - \frac{mto(A,A_\emptyset)}{mto(A_\emptyset,A) + mto(A_\emptyset,A_\emptyset)}) \times 100\% = (1 - \frac{mto(A_\emptyset,A)}{mto(A_\emptyset,A)}) \times 100\% = 0\%$$

$(II)\quad if\ \ a2a(A,B) = 0\% \quad \Rightarrow \quad mto(A,B) = mto(A_\emptyset,A) + mto(A_\emptyset,B)$

$$= \quad mto(A,A_\emptyset) + mto(A_\emptyset,B) \quad (*)$$

$$\forall e \in E_A \cup E_B \ \ cost\ of\ operations\ on\ e = \begin{cases} 0 & \text{if } e \in E_{fix} \\ 1\ \ (move) & \text{if } e \in (E_A \cap E_B) \setminus E_{fix} \\ 2\ \ (remove) & e \in E_A \setminus E_B \\ 2\ \ (add) & e \in E_B \setminus E_A \end{cases}$$

$$\Rightarrow \ \ cost\ of\ entity\ operations\ in\ mto(A,B) \ \ \leq \ \ 2 \times |E_A \cup E_B|$$

$$\leq \ \ 2 \times |E_A| + 2 \times |E_B| \quad (**)$$

$$if\ \ |C_B| > 0 \quad \Rightarrow \quad abs(|C_A| - |C_B|) < |C_A| + |C_B| \quad (***)$$

$$(**) \wedge (***) \ \ \Rightarrow \ \ cost\ of\ entity\ operations\ in\ mto(A,B) + abs(|C_A| - |C_B|)$$

$$\leq \ \ 2 \times |E_A| + 2 \times |E_B| + |C_A| + |C_B|$$

$$\Rightarrow \ \ mto(A,B) < mto(A,A_\emptyset) + mto(A_\emptyset,B) \quad (****)$$

$$(****) \ \ contradicts\ with \ \ (*) \ \ \Rightarrow \ \ |C_B| = 0 \ \ \Rightarrow \ \ B = A_\emptyset$$

Therefore, $a2a$ returns 0% if and only if one of the architectures equals to $A_\emptyset$.

4. $\forall A\ ,\ B\ :\quad a2a(A,B) = a2a(B,A)$

$$a2a(A,B) = (1 - \frac{mto(A,B)}{mto(A_\emptyset,A) + mto(A_\emptyset,B))}) \times 100\%$$

$$= (1 - \frac{mto(B,A)}{mto(A_\emptyset,B) + mto(A_\emptyset,A))}) \times 100\% = a2a(B,A)$$

Therefore, $a2a$ is symmetric.

It is important to note that the denominator of $a2a$ is neither over- nor under-weighted, meaning that for every architecture A, a2a returns the full range of values between 0% and 100%. This property is a consequence of the first, second, and third properties.

4 Cluster Coverage (*cvg*)

This section describes the *cvg* metric and its properties. A reader may choose to skip the section if she is not interested in the details contained therein. While the empirical study described in the remainder of the paper was enabled by *cvg* in a critical way, understanding the obtained data and its interpretation does not require understanding the details of the metric itself.

From the architectural perspective, a system may not change structurally at the same rate at which its individual components change. This indicates that architectural maintenance issues may occur at different points and may have different scopes. To measure component-level change, we introduce a novel similarity metric, *cvg*, which measures the extent to which components are added or removed as a software system evolves. In order to calculate *cvg*, we use cluster-to-cluster (*c2c*), a metric we developed and applied in our recent work (Garcia et al., 2013a) to assess component-level changes. In the remainder of this section, we recap *c2c*, explain *cvg*, and provide an analysis of *cvg*'s properties.

4.1 Similarity Between Two Clusters

c2c is a similarity metric that measures the degree of overlap between implementation-level entities contained within two clusters:

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

where $c_i \cap c_j$ is the set of common entities between clusters c_i and c_j; and $|c_i|$ and $|c_j|$ are the numbers of entities residing in clusters c_i and c_j respectively. 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.

In architectures *A* and *B* in our example, $c2c(a_1, b_1) = 25\%$ since the leftmost circle with the label "1" is common between these two clusters and the size of the larger cluster (i.e., b_1) is 4. a_1 has no common entity with cluster b_4, hence $c2c(a_1, b_4) = 0\%$. Figure 8 shows *c2c* values between clusters of architecture *A* and architecture *B*.

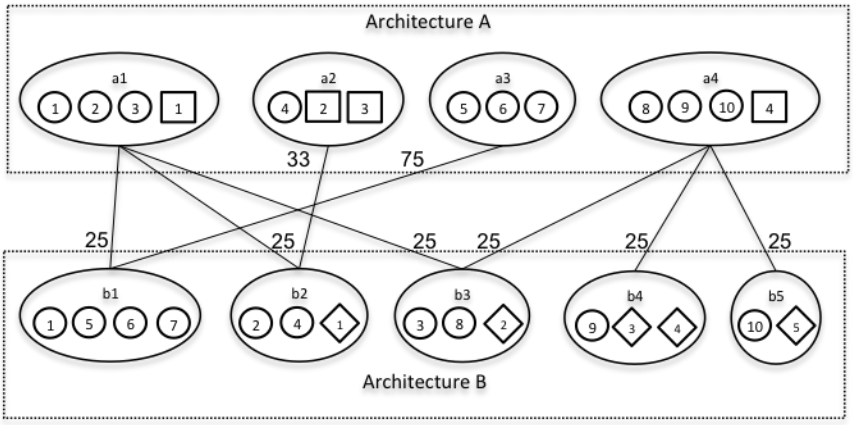

Fig. 8: *c2c* values between clusters of architecture *A* and architecture *B*. If there is no line between clusters a_i and b_j, then $c2c(a_i, b_j) = 0\%$.

4.2 Calculating Cluster Coverage

cvg allows an engineer to determine the extent to which certain components of a system exist in an earlier version or were added in a later version. *cvg* is calculated as follows:

$$cvg(A, B) = \frac{|simC(A, B)|}{|C_A|} \times 100\%$$

where the denominator $|C_A|$ is the number of clusters in architecture *A*.

$simC(A,B)$ returns the set of components of architecture A that have at least one matched component in architecture B. Two components *match* if their similarity, computed using $c2c$, is higher than a specified threshold th_{cvg}.

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

The threshold th_{cvg} determines the level of conservativeness of cvg. A higher th_{cvg} results in a more conservative comparison. In our example from Figure 8, if $th_{cvg} = 80\%$, then none of the $c2c$ values reach the threshold, which results in $cvg(A,B) = 0\%$ and $cvg(B,A) = 0\%$. On the other hand, a very low th_{cvg} results in an unrealistically high cvg. In our example, if $th_{cvg} = 20\%$, then every component in A finds a match in B and vice versa, which results in $cvg(A,B) = 100\%$ and $cvg(B,A) = 100\%$. With $th_{cvg} = 67\%$, only $c2c(a_3,b_1)$ reaches the threshold. As a result, $cvg(A,B) = 25\%$ and $cvg(B,A) = 20\%$, which is more intuitive than the cvg values obtained using the other thresholds we previously considered. We have seen this pattern in a number of systems we have analyzed to date. For this reason, we recommend setting $th_{cvg} = 67\%$, which we use for our empirical study later in this paper. Setting $th_{cvg} = 67\%$, inspired by experiments performed in our prior work Garcia et al. (2013a); Lutellier et al. (2015), ensures that the component remains largely unchanged, while still allowing it to vary to a reasonable degree, when considering component equivalence across versions.

As another 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.

4.3 Properties of Cluster Coverage

cvg exhibits certain mathematical properties that are important for our study of architectural change. These properties are related to the range of values the metric can take, the value it takes for identical architectures, and the value it takes when a non-empty architecture is compared with a "null" architecture. We expand on these properties below:

1. $cvg(A,B)$ returns a value between 0% and 100%.

$$\forall\, A\ ,\ B\ :\quad 100\% \geq cvg(A,B) \geq 0\%$$

   cvg has this property since $simC(A,B)$ and C_A are sets, and $simC(A,B) \subset C_A$.
2. For any two identical architectures, cvg returns a similarity of 100%.

$$\forall\, A\ ,\ B\ :\quad A = B \quad\Rightarrow\quad cvg(A,B) = 100\%$$

   cvg has this property since

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

$$\Rightarrow\ \ \forall\, th_{cvg} < 100\%,\ A:\ \ simC(A,A) = \{c_i \mid c_i \in A,\ \exists c_j \in A,\ c2c(c_i,c_j) > th_{cvg}\} = C_A$$

$$\Rightarrow\ \ cvg(A,B) = cvg(A,A) = \frac{|simC(A,A)|}{|C_A|} \times 100\% = \frac{|C_A|}{|C_A|} \times 100\% = 100\%$$

3. For any non-empty architecture A, *cvg* returns a 0% similarity between A and a "null" architecture $A_{\emptyset}$.

$$\forall A \quad : \quad cvg(A, A_{\emptyset}) = 0\%$$

This property always holds since:

$$cvg(A, A_{\emptyset}) = \frac{|simC(A, A_{\emptyset})|}{|C_A|} \times 100\% = \frac{|\emptyset|}{|C_A|} \times 100\% = 0\%$$

For a given architecture A, *cvg* may returns values that span the complete range of values between 0% and 100%. This property is a consequence of the first, second, and the third properties. However, *cvg* is not symmetric, as explained by the example in Section 4.2. Also note that if $cvg(A,B)$ equals 100%, A and B are not necessarily identical. Furthermore, if $cvg(A,B)$ equals 0%, neither A nor B need be a "null" architecture.

5 ARCADE

This section describes the *ARCADE* workbench, its architecture, and its implementation. A reader may choose to skip the section if she is not interested in the details contained therein. While the empirical study described in the remainder of the paper was directly enabled by *ARCADE*, understanding the obtained data and its interpretation does not require understanding the details of the workbench itself.

To study architectural change and decay, *ARCADE* (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 is designed with the requirement of conducting replicable, reusable, and scalable studies. Therefore, we have developed a support framework named *ARCADE-Controller* as part of *ARCADE*. *ARCADE-Controller* helps users to define the workflow of an analysis and to employ the power of cloud computing for distributing the analysis on the cloud. *ARCADE*'s overall architecture and implementation as well as *ARCADE-Controller* will be discussed in the following sub-sections.

5.1 ARCADE's Architecture and Implementation

ARCADE's foundational element is architecture recovery, depicted as the *Recovery Techniques* component in *ARCADE*'s dataflow architecture shown in Figure 9. The architectural views produced by *Recovery Techniques* are directly used for studying architectural changes. *ARCADE* currently provides access to ten recovery techniques; nine techniques use algorithms for clustering implementation-level elements into architectural components Garcia et al. (2013a), while one technique reports the implementation view of a system's architecture (i.e., 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.

For each architecture, *ARCADE* computes the change metrics discussed in Sections 2.1, 3, and 4. 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 9) that is relevant to this

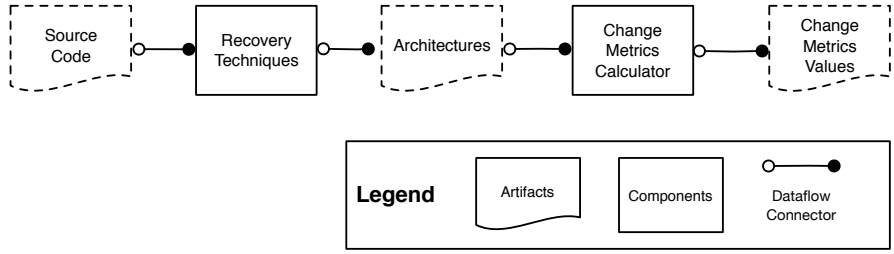

Fig. 9: Architecture recovery and analysis components and artifacts leveraged in this study.

paper.[1] This artifact is then used to interpret the degree of architectural change in the manner discussed in Section 6. *ARCADE* employs our own implementations of the change metrics in a manner that directly reflects the details described in Sections 3 and 4.

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 2.1 is computed using a version pair. We obtained the correct evolution paths for our subject systems by using git-log (Git, 2014) and svn-graph-branches (Mengué, 2014). The two tools provide analogous functionality on the two different types of repositories, Git and SVN. Both tools can parse repository log files and create graphs that show relationships between software versions. For example, if version 1.1 of a system is created from version 1.0, then in the evolution graphs version 1.1 will be linked from version 1.0.

The described features of *ARCADE* are implemented in Java and Python. *ARCADE* is available for download from (ARCADE, 2015).

A critical part of *ARCADE* is the selection of appropriate architecture recovery techniques. Since our previous evaluation (Garcia et al., 2013a) 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 2.2), we focus on them in our study. *ACDC*'s view is based on several common, familiar subsystem patterns found in large systems, such as directory structures and graphical properties of source files (e.g. their out-degrees or subgraph domination) (Tzerpos and Holt, 2000). 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*.

PKG is a straightforward recovery technique, since package information and directory structure is directly available from a software system's implementation. On the other hand, *ARC* and *ACDC* perform more sophisticated analyses to extract an architectural view of a system. Both of these techniques introduce challenges that must be addressed when studying architectural change. In the following sub-sections, we explain *ACDC* and *ARC* in more

[1] The current version of *ARCADE* (ARCADE, 2015) 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.

detail, the challenges arising from using them for studying architectural change, and the solutions we developed to address those challenges.

5.1.1 ACDC

To recover architecture using *ACDC*, we obtained an implementation of the technique from *ACDC*'s authors (Tzerpos and Holt, 2000) and used its default settings. Although *ACDC* relies on a deterministic clustering algorithm, it turned out that its implementation is not deterministic. This initially introduced inaccuracies in our empirical analysis. Specifically, applying the original implementation of *ACDC* on the same source code twice yields architectures that are usually 95% similar according to the *a2a* metric.

We traced the source of *ACDC*'s non-determinism bug to the implementation of 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.

5.1.2 ARC

To represent system-level concerns, ARC leverages probabilistic topic modeling (Blei, 2012), which are machine-learning algorithms for determining thematic topics in text documents. For ARC, each topic obtained using such algorithms represents a system-level concern. In order to represent the topic models needed for *ARC*, we have used MALLET, a machine learning toolkit (McCallum, 2002). The topic-model extraction algorithms implemented by MALLET are non-deterministic. This posed a problem when trying to meaningfully compare two concern-based architectures as required for our study, since 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. Another challenge arising from using topic modeling is that there is no generally agreed-upon or objectively computable number of topics for a given body of text (McCallum, 2002; Blei, 2012). For each system in our study, the number of topics was determined based on our experience with *ARC* from a previous empirical evaluation (Garcia et al., 2013a). We used the resulting multi-version topic model for a system to recover the architectures for all of that system's versions.

In addition, we also computed architectural changes between a large number of pairs consisting of different versions of the same system each 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 a 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.

Our ultimate goal in employing topic modeling is to extract meaning from system code. Natural language texts (e.g., newspaper articles and books) are intended for human readers and use vocabularies with typically well-defined semantics. Text found in code, however, does not necessarily use identifiers or comments that are human-readable. In fact, such code may be intentionally obfuscated to prevent human readability (e.g., to protect intellectual property). To obtain accurate topics from software, developers should use meaningful identifiers or

write intelligible comments. To mitigate such issues, probabilistic topic models rely upon frequencies of words in code sampled to fit a probability distribution known to be representative of textual documents. Furthermore, it is reasonable to expect that ARCADE would be used by an organization that owns the software system under maintenance, precluding the need to resort to measures like obfuscation.

To reduce issues arising from noise that appears in text, we must select appropriate *stop words*, which are words that have low semantics and reduce the quality of obtained topics, for the domain of software. We selected stop words from the English language (e.g., articles like "the"), general computing, programming languages (e.g., keywords in Java), and individual systems (e.g., a system's name). Finally, it is also possible in topic modeling that the resulting topics are not easily understandable by a human unfamiliar with the software system in question. Such topics may be confusing for new engineers still learning about the system; however, those topics may still be meaningful for the original and/or more knowledgeable engineers.

To construct a highly accurate topic model for each subject system, we identified frequently used yet unimportant words (e.g., words about license agreements that appear in many source files). Those words should be ignored when constructing topic models to prevent excessive overlap between topics. To identify those words, we designed a refinement process involving three PhD students. Each student individually identified words involving license agreements, meaningless variable names, and subject-system names. From this set, the participants agreed upon a set of common words that should be ignored. We then supplied the resulting words to MALLET as stop words which, in turn, ignores those words during topic-model construction.

5.2 Automation of the Analysis Workflow via *ARCADE-Controller*

For mining software repositories, the ability to replicate experiments is essential for evaluating different techniques and assessing their findings. In addition, the ability to scale an empirical study, in terms of the number of systems and revisions involved, enhances the generalizability of a study's conclusions. Prior research has shown that many empirical studies in mining software repositories suffer from scalability or replicability issues (Ghezzi and Gall, 2013; Robles, 2010). To address these issues, we have designed and implemented *ARCADE-Controller*, a support framework that plugs into *ARCADE* and helps engineers define and execute complete workflows for architecture recovery and analysis over a set of software systems and their revisions. As depicted in Figure 10, *ARCADE-Controller* augments *ARCADE*'s data flow from Figure 9 in a relatively straightforward but critical way.

To conduct analyses on a large number of software versions, *ARCADE-Controller* distributes a user-defined workflow over multiple cloud instances. Each instance is responsible for downloading the source code of multiple versions from online repositories, compiling the source code, and recovering the architecture of each system version. *ARCADE-Controller* gathers recovered architectures onto an analysis server for further evolutionary studies, such as computing architectural change metrics between the recovered architectures. To enable portability, *ARCADE-Controller* is developed in the bash scripting language, which makes it runnable on Unix-based operating systems (e.g., Linux and Mac OS X).

ARCADE-Controller interacts with cloud infrastructures through their command-line interfaces (e.g., Amazon CLI (Amazon, 2015)). The tool provides cloud management operations, such as launching instances, setting up required software, and terminating instances. In addition, it deploys static and dynamic analyses on each cloud instance and downloads

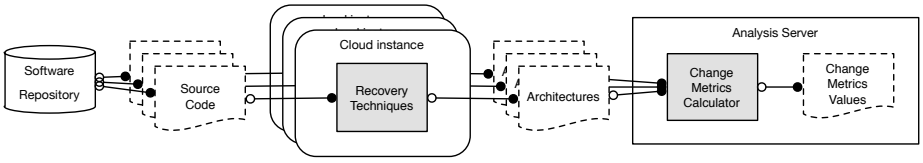

Fig. 10: Deployment of *ARCADE* components onto the cloud using *ARCADE-Controller* for rapid, large-scale architecture recovery and analysis.

the results to an analysis server. In the study described in this paper, we used the Amazon EC2 platform to run our analysis. The layered architecture of *ARCADE-Controller* makes it possible to run analyses on other cloud infrastructures, such as Google Cloud Platform (Google, 2015a).

ARCADE-Controller provides a number of scripts to help users define analysis steps over subject systems. For example, users can set up a cloud instance to download source code from popular repositories (e.g. Github (Git, 2015) and BitBucket (Bitbucket, 2015)). Users are also able to select the version granularity they want (e.g., internal commits or official releases). Currently, *ARCADE-Controller* supports compiling the source code of Ant (Apache, 2015a) and Maven (Apache, 2015b) projects semi-automatically. We have defined many compilation scenarios (e.g., using different Java versions) to minimize manual configuration.

ARCADE-Controller supports incorporating tools outside of *ARCADE* by providing a standard wrapper that serves as an interface to *ARCADE*. *ARCADE-Controller* allows workflows to be defined among these wrapped tools and *ARCADE*'s built-in analyses. This design allows tools and analyses to be re-used in different scenarios. By defining a workflow of architecture-recovery analysis using *ARCADE-Controller*, researchers can conduct large-scale analyses that are replicable with little effort.

We have tested the applicability of this approach by integrating other static and dynamic programming analysis tools, such as PMD (PMD, 2015), a static source code analyzer, and FieryEye (Mahajan et al., 2016), a dynamic web-interface analyzer, into *ARCADE*. In the case of FieryEye, we reduced the execution time of the analysis from a couple of days to several hours by using hundreds of EC2 nodes. It is not mandatory to run *ARCADE*'s analyses using cloud computing. All the tools in section 5.1 can be run on a local machine.

For this paper's analyses, which involved tens of millions of source lines of code (MSLOC), cloud computing significantly reduced the running time. Downloading and compiling the source code of each version of a system, and subsequently recovering the architecture of that version in our workflow, are parallelized on cloud servers.

6 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 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 (Apache, 2014a)). In turn, this should imply a substantial change to the system's architecture. This research question will target our hypothesis, formed after our initial observations, that a system's architecture may experience significant change even though the system remains within the same major version.

In order to answer these research questions, we extended our previous work (Le et al., 2015) by applying *ARCADE* to a total of 720 versions of 18 Apache open-source systems.The largest versions of these systems range between 50 KSLOC and 800 KSLOC. All of these systems are implemented in Java and managed in the Apache Jira repository. Table 1 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.

In addition to the Apache subject systems, we have analyzed another 5 systems that are not from the Apache Software Foundation. We use the data from these new systems to further verify our conclusions, and to address a threat to validity, discussed in Section 8. We refer to

Table 1: Apache subject systems analyzed in our study

System	Domain	No. of Ver.	Time span	MSLOC
Accumulo	Data Storage System	10	05/15-09/15	1.59
ActiveMQ	Message Broker	20	08/04-01/07	3.40
Cassandra	Distributed DBMS	127	09/09-09/13	22.0
Chukwa	Data Monitor	7	05/09-02/14	2.20
Hadoop	Data Process	63	04/06-08/13	30.0
HttpClient	HTTP Toolset	88	12/07-09/15	3.31
Ivy	Dependency Manager	20	12/07-02/14	0.40
JackRabbit	Content Repository	97	08/04-02/14	34.2
Jena	Semantic Web	7	06/12-09/13	3.50
JSPWiki	Wiki Engine	54	10/07-03/14	1.20
Log4j	Logging	41	01/01-06/14	2.40
Lucene	Search Engines	21	12/10-01/14	4.90
Mina	Network Framework	40	11/06-11/12	2.30
PDFBox	PDF Library	17	02/08-03/14	2.70
Poi	Java API	20	05/15-09/15	1.68
Struts 2	Web Apps Framework	36	10/06-02/14	6.70
Tika	Content Analysis Toolkit	30	05/10-05/15	0.56
Xerces	XML Library	22	03/03-11/09	2.30
Total		**720**	**01/01-09/15**	**125.33**

them as non-Apache systems. In total, we have analyzed 211 versions of the non-Apache systems, as summarized in Table 2.

Table 2: Non-Apache subject systems analyzed in our study

System	Domain	No. of Ver.	Time span	MSLOC
Druid-core	Alibaba JDBC Library	27	04/12-08/14	4.61
Guava-core	Google Java Library	20	08/12-09/15	1.36
Jackson(JS)-databind	Data Binding Library	56	02/12-10/15	4.23
PgJDBC	PostgreSQL JDBC driver	64	01/05-10/15	2.27
TestNG	Testing Framework	44	07/10-10/15	2.33
Total		**211**	**01/05-10/15**	**14.8**

We applied *ARCADE*'s workflow depicted in Figure 9 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 (ARCADE, 2015).

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. 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 5). Determining the accurate evolution path 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 (Git, 2014) and svn-graph-branches (Mengué, 2014). 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 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 (Struts, 2014) 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.

The version numbering convention adopted by developers in the non-Apache systems is similar, although less consistent, when compared to that in the Apache systems. In the non-Apache systems, the developers tend not to strictly follow the convention, or they tend to have a preference for a single type of system version. For example, Google almost exclusively releases major and beta versions, along with a few patch versions, of the Guava library (Google, 2015b). However, we still apply the above approach of selecting version pairs to the non-Apache systems. This helps us to understand the differences in the version change decisions the subject systems.

7 Results

To shed light on the four research questions about architectural change, we leveraged *ARCADE* to compute the *a*2*a* and *cvg* metrics (recall Section 2.1). For each version pair within each evolution path of a system (recall Section 5), we computed these metrics using the three architectural views produced by *ACDC*, *ARC*, and *PKG*. For ease of comparison, the results obtained from the two sets of subject systems—Apache systems and non-Apache systems—are separated into different tables below. Tables 3 and 4 show the average *a*2*a* values for the two sets of subject systems, while Tables 5 and 6 show the average *cvg* values for each system in the two sets. 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 research question below.

7.1 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. Tables 3 and 4 show average *a2a* values for the five different types of evolution paths we selected across the three architectural views.

Table 3: Average *a2a* values between versions of Apache subject systems.

	ACDC					*ARC*				
System	Major	MinMaj	Minor	Patch	Pre	Major	MinMaj	Minor	Patch	Pre
Accumulo	-	-	84	99	-	-	-	84	98	-
ActiveMQ	62	69	95	100	99	61	66	93	100	98
Cassandra	42	80	77	99	99	32	75	70	98	99
Chukwa	-	-	78	-	95	-	-	71	-	92
Hadoop	17	73	86	98	-	19	74	83	95	-
HttpClient	-	-	87	98	98	-	-	85	97	97
Ivy	50	67	91	98	99	28	52	86	95	97
JackRabbit	38	76	84	91	98	26	75	84	99	95
Jena	-	-	88	99	-	-	-	89	95	-
JSPWiki	18	30	86	98	99	10	25	72	98	99
Log4j	9	13	64	97	85	5	6	68	99	86
Lucene	12	8	96	98	94	11	9	97	100	93
Mina	28	30	92	99	88	15	16	93	99	89
Poi	-	-	90	99	98	-	-	86	100	94
Struts2	-	-	90	99	-	-	-	94	99	-
Tika	60	97	94	-	100	54	96	92	-	100
Xerces	21	54	92	83	-	18	54	91	94	-
AVG	**32**	**55**	**87**	**97**	**96**	**25**	**50**	**85**	**98**	**95**
DEV	**19**	**30**	**8**	**4**	**5**	**17**	**31**	**9**	**2**	**4**

	PKG				
System	Major	MinMaj	Minor	Patch	Pre
Accumulo	-	-	85	99	-
ActiveMQ	62	71	94	100	98
Cassandra	36	79	74	99	99
Chukwa	-	-	79	-	94
Hadoop	14	81	91	100	-
HttpClient	-	-	90	99	98
Ivy	35	57	89	98	99
JackRabbit	30	82	92	100	99
Jena	-	-	94	99	-
JSPWiki	8	13	87	99	100
Log4j	1	2	61	98	91
Lucene	1	1	97	99	90
Mina	13	13	98	100	86
PDFBox	-	-	97	100	-
Poi	-	-	92	100	99
Struts2	-	-	93	99	-
Tika	60	98	95	-	100
Xerces	15	63	91	90	-
AVG	**21**	**49**	**89**	**96**	**96**
DEV	**22**	**36**	**10**	**2**	**5**

- Value unit is percentage.
- Lower numbers mean more change.
- Empty table cells indicate versions that do not exist for a given system.
- The second bottom-most row is the average-of-averages.
- The bottom-most row is the standard deviation.

In Table 3, we observed a consistent trend for system-level architectural change among the three views of the Apache systems. 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. From the table, we can see a prevalent overall trend:

$$a2a_{Pre} \approx a2a_{Patch} > a2a_{Minor} > a2a_{MinMaj} > a2a_{Major}$$

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 that sometime require more changes than patch versions. Although the averages of the *Patch* and *Pre* columns are approximately equal, we observed significant changes between pre-release versions in some cases. For example, in Log4j, the $a2a$(1.3-alpha-6, 1.3-alpha-7) value is 49%, and the $a2a$(2.0-rc1, 2.0-rc2) value is 72%. The prevalent overall trend can be observed from a side-by-side depiction of three representative systems' evolutions, shown in Figure 11.

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% for *ACDC*. Hadoop had more than twenty minor versions between versions 0.1.0 and 0.20.x, before releasing version 1.0.0 (Apache, 2014b). 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.

Incremental changes between consecutive minor versions need not always result in higher architectural similarity between the last minor version and the subsequent major version, and may be dwarfed by the changes a major version introduces. This is illustrated by the case of Lucene, whose pair $(a2a_{Major}, a2a_{MinMaj})$ is (12%, 8%) for *ACDC*, while its $a2a_{Min}$ is 96% for its six minor releases from versions 3.0.0 through 3.6.0. In fact, the new major version 4.0.0 has no significant similarity to the previous major version (3.0.0) or its most proximate minor version (3.6.0). Looking into the code history of Lucene, we found that multiple changes between minor versions are related to backward-compatibility issues. For example, Lucene 3.6.0 contains packages that are added to support backward-compatibility

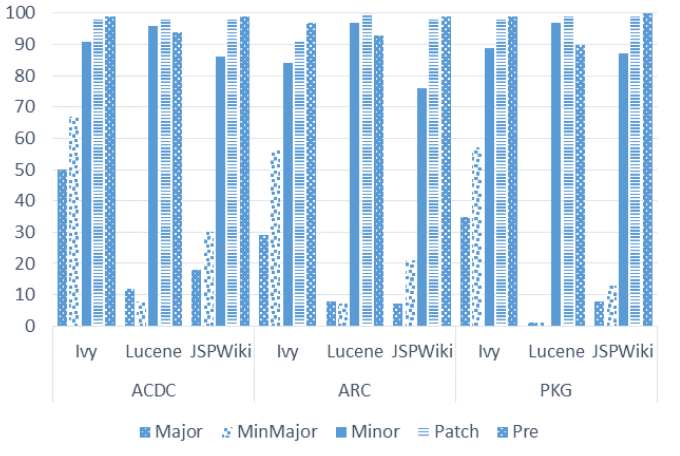

Fig. 11: Average a2a values in three subject systems

Table 4: Average *a2a* values between versions of non-Apache software systems.

	ACDC					*ARC*				
System	Major	MinMaj	Minor	Patch	Pre	Major	MinMaj	Minor	Patch	Pre
Druid	82	-	-	99	-	79	-	-	96	-
Guava	94	-	-	100	100	93	-	-	100	100
JS-databind	-	-	95	99	100	-	-	92	99	99
PgJDBC	69	89	95	99	-	64	86	91	99	-
TestNG	-	-	91	99	-	-	-	87	99	-
AVG	**76**	**89**	**94**	**99**	**100**	**76**	**86**	**91**	**99**	**100**
DEV	**10**	**-**	**2**	**0**	**0**	**11**	**-**	**2**	**1**	**0**

	PKG				
System	Major	MinMaj	Minor	Patch	Pre
Druid	86	-	-	99	-
Guava	93	-	-	100	100
JS-databind	-	-	96	99	100
PgJDBC	71	90	96	99	-
TestNG	-	-	91	99	-
AVG	**83**	**90**	**94**	**99**	**100**
DEV	**9**	**-**	**2**	**0**	**0**

- Value unit is percentage.
- Lower numbers mean more change.
- Empty table cells indicate versions that do not exist for a given system.
- The second bottom-most row is the average-of-averages.
- The bottom-most row is the standard deviation.

with versions 3.1.x and 3.3.x. At the time version 4.0.0 was released, a substantial number of changes happened in the backward-compatibility policy. Subsequently, those packages were removed from version 4.0.0.

Obtaining the consistent trends across the recovered architectural views that are shown in Table 3 at times required that 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 *a*2*a* values. On the other hand, *ARC* performs clustering of code entities (for the Java code evaluated here, these are classes and enums) 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 7.2), our analysis of the *a*2*a* metric's results provided the first indication that *PKG* may not always accurately reflect architectural change. Namely, the *a*2*a* values for the architectures suggested by *PKG* are uniformly higher than corresponding values in the *ACDC* and *ARC* views. 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.

For all three architectural views, the standard deviation values (the bottom-most "DEV" row) show a consistent trend. The DEV value of $a2a_{MinMaj}$ is larger than the one of $a2a_{Major}$. This reflects that developers tend to increase a major version number depending on the accumulated architectural change during the entire major version, rather than on the degree of architectural change since the last minor version. This is reflective of practices expected of well-organized development teams, which usually define a long-term road map of future features and associate those with system versions.

The *a2a* values of non-Apache systems in Table 4 follow the observed trend in the Apache systems. For example, the $a2a_{Major}$ values are the lowest among the five evolution paths, indicating that the most significant architectural change occurs between two major versions of a non-Apache system. Compared to the Apache systems, the non-Apache systems have higher *a2a* values with smaller gaps between different evolution paths. For example, in the ACDC view, the average values of $a2a_{Major}$, $a2a_{Minor}$ and $a2a_{Patch}$ are, respectively, 32%, 87%, and 97% for the Apache systems, and 76%, 94%, and 99% for the non-Apache systems. This can be explained by the different versioning style used in non-Apache systems. For example, as explained in Section 6, developers of Guava almost exclusively use major numbers.

7.2 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 2.1) 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 5 and 6 depict 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.

We first discuss the result of Apache subject systems in Table 5. 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:

$$cvg_{Pre} \approx cvg_{Patch} > cvg_{Minor} > cvg_{MinMaj} > cvg_{Major}$$

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 5) 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 5) 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

Table 5: Average *cvg* values between versions of Apache subject systems.

	ACDC										*ARC*									
	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)
Accumulo	-	-	-	-	71	64	99	99	-	-	-	-	-	-	77	66	99	98	-	-
ActiveMQ	28	19	61	48	95	92	100	100	99	99	37	27	57	54	91	85	99	99	94	92
Cassandra	5	4	59	53	52	46	98	99	98	98	39	19	71	61	65	55	97	96	95	95
Chukwa	-	-	-	-	63	54	-	-	86	86	-	-	-	-	54	44	-	-	72	72
Hadoop	0	0	54	46	83	74	95	98	-	-	20	3	71	55	82	73	96	96	-	-
HttpClient	-	-	-	-	65	64	97	97	96	95	-	-	-	-	74	70	93	94	93	92
Ivy	6	4	46	45	67	57	100	96	100	96	7	5	46	42	47	41	82	75	93	95
JackRabbit	16	7	53	57	87	81	98	97	96	96	49	15	66	65	85	78	99	99	92	92
Jena	-	-	-	-	81	74	96	96	-	-	-	-	-	-	81	77	92	92	-	-
JSPWiki	0	0	0	0	38	35	85	84	98	98	0	0	39	9	41	33	90	87	98	98
Log4j	0	0	0	0	29	21	94	93	85	82	7	2	5	2	57	42	87	85	81	79
Lucene	0	0	0	0	87	84	98	98	99	99	0	0	0	0	92	91	99	99	85	91
Mina	4	2	4	2	78	78	99	99	87	80	10	6	12	8	85	85	99	99	82	77
PDFBox	-	-	-	-	94	92	95	94	-	-	-	-	-	-	88	85	99	97	-	-
Poi	-	-	-	-	86	83	100	100	94	95	-	-	-	-	84	78	100	100	88	89
Struts2	-	-	-	-	79	83	96	96	-	-	-	-	-	-	74	78	96	95	-	-
Tika	25	17	100	100	80	77	-	-	100	100	52	26	83	85	82	78	-	-	100	100
Xerces	0	0	20	16	83	81	86	83	-	-	12	5	26	29	85	79	86	82	-	-
AVG	**7**	**5**	**36**	**33**	**73**	**69**	**96**	**95**	**95**	**94**	**21**	**10**	**43**	**37**	**72**	**74**	**96**	**94**	**89**	**89**
DEV	**10**	**7**	**33**	**32**	**19**	**20**	**5**	**5**	**6**	**7**	**19**	**10**	**29**	**29**	**16**	**18**	**6**	**7**	**8**	**8**

	PKG									
	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)
Accumulo	-	-	-	-	82	72	100	99	-	-
ActiveMQ	33	27	60	67	96	93	100	100	100	97
Cassandra	29	18	77	69	69	60	98	100	99	99
Chukwa	-	-	-	-	76	67	-	-	93	93
Hadoop	0	0	54	46	95	85	100	99	-	-
HttpClient	-	-	-	-	83	81	99	99	98	98
Ivy	14	11	48	46	81	65	100	97	100	97
JackRabbit	28	12	65	73	93	86	99	98	98	98
Jena	-	-	-	-	97	93	99	99	-	-
JSPWiki	0	0	25	5	63	51	97	96	100	100
Log4j	0	0	0	0	69	54	99	97	92	88
Lucene	0	0	0	0	88	85	99	99	70	89
Mina	8	4	8	4	96	96	97	96	91	83
PDFBox	-	-	-	-	97	97	98	97	-	-
Poi	-	-	-	-	94	90	100	100	97	98
Struts2	-	-	-	-	91	95	98	98	-	-
Tika	30	20	100	100	93	90	-	-	100	100
Xerces	7	3	20	10	85	83	90	88	-	-
AVG	**13**	**9**	**42**	**38**	**86**	**80**	**98**	**97**	**95**	**95**
DEV	**13**	**10**	**33**	**35**	**10**	**15**	**2**	**3**	**8**	**5**

- Value unit is percentage.
- Lower numbers mean more change.
- Empty table cells indicate versions that do not exist for a given system.
- The second bottom-most row is the average-of-averages.
- The bottom-most row is the standard deviation

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.

Table 6: Average *cvg* values between versions for non-Apache subject systems.

	ACDC										*ARC*									
	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)
Druid	57	50	-	-	-	-	97	96	-	-	77	67	-	-			94	93	-	-
Guava	89	85	-	-	-	-	100	100	100	100	89	85	-	-	-	-	100	100	100	100
JS-databind	-	-	-	-	89	88	100	100	99	99	-	-	-	-	83	81	100	99	98	98
PgJDBC	31	21	76	68	87	83	98	98	-	-	50	34	79	74	83	78	98	97	-	-
TestNG	-	-	-	-	83	81	99	100	-	-	-	-	-	-	80	77	99	99	-	-
AVG	**59**	**52**	**76**	**68**	**86**	**84**	**99**	**98**	**99**	**99**	**72**	**62**	**79**	**74**	**82**	**79**	**98**	**98**	**99**	**99**
DEV	**24**	**26**	**-**	**-**	**2**	**3**	**1**	**2**	**1**	**1**	**16**	**21**	**-**	**-**	**11**	**2**	**2**	**3**	**1**	**1**

	PKG									
	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)
Druid	82	72	-	-	-	-	99	98	-	-
Guava	94	88	-	-	-	-	100	100	100	100
JS-databind	-	-	-	-	97	96	100	100	97	97
PgJDBC	87	59	95	86	99	92	99	100	-	-
TestNG	-	-	-	-	89	87	99	100	-	-
AVG	**88**	**73**	**95**	**86**	**95**	**92**	**99**	**100**	**99**	**99**
DEV	**5**	**12**	**-**	**-**	**4**	**4**	**0**	**1**	**2**	**2**

- Value unit is percentage.
- Lower numbers mean more change.
- Empty table cells indicate versions that do not exist for a given system.
- The second bottom-most row is the average-of-averages.
- The bottom-most row is the standard deviation

The prevalent trend of *cvg* values remains intact across all five evolution paths of the non-Apache systems, as shown in Table 6. We note that the *cvg* values in Table 6 are markedly higher than the corresponding values in the Apache subject systems in Table 5. For example, in the ACDC view, the average values ("AVG") of $cvg_{Major}(s,t)$ and $cvg_{Major}(t,s)$ are, respectively, 7% and 5% for the Apache systems, and 59% and 52% for the non-Apache systems. Therefore, not only are the non-Apache systems more stable at the system-level, but also at the component-level.

We can conclude that *a2a* and *cvg* mostly display consistent trends for both the Apache and non-Apache systems. In Section 7.3, we will discuss instances in which *a2a* and *cvg* show different aspects of architectural change that buck this trend. However, this is apparent in some of the larger Apache systems, but was not observed in the non-Apache systems. Another noticeable difference between the Apache and non-Apache systems is that the architectural changes in non-Apache systems are much smaller when moving to a new major version (*MinMaj*). This again reiterates the importance of stability and backward compatibility in non-Apache systems.

7.3 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* (architecture-level) and *cvg* (component-level) change metrics. 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*. The reason for that is that *a2a* and *cvg* measure two different aspects of architectural evolution. *a2a* measures the similarity between two architectures in terms of the number of operations required to transform one architecture to the other, while *cvg* measures the similarity between two architectures in terms of the number of

components that persist in the course of evolution. In JSPWiki, *cvg* yielded 0% because no two components in the major versions were "sufficiently" similar based on the chosen threshold. On the other hand, *a2a* found some shared entities among the major versions, resulting in a non-zero degree of similarity.

Another rełvealing 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 (Apache, 2014c). 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).

In Apache systems, 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 *ACDC* and *PKG* yield as compared to *ARC* suggest that the semantics-based perspective of *ARC* yields more architectural changes at the system-level as compared to the structure-based perspectives of *ACDC* and *PKG*.

To illustrate the difference in architectural similarity yielded by the structural views—*ACDC* and *PKG*—as compared to *ARC*, Figures 12–14 depict architectural changes among minor versions of Ivy: Figure 12 depicts the *a2a* values; Figure 13 depicts the *cvg* values for each version pair (s,t); and Figure 14 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 12–14.

Figure 12 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 13 and 14 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

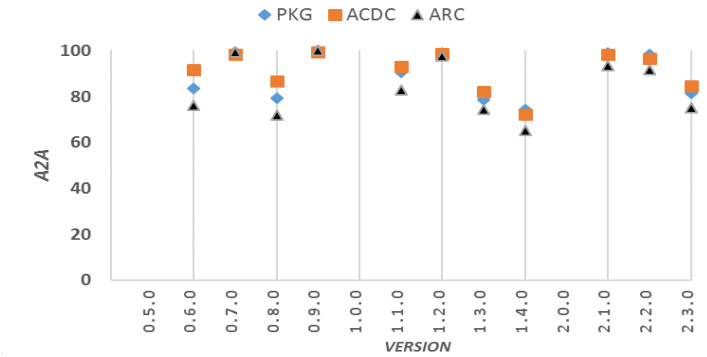

Fig. 12: *a2a* values between minor versions of Ivy

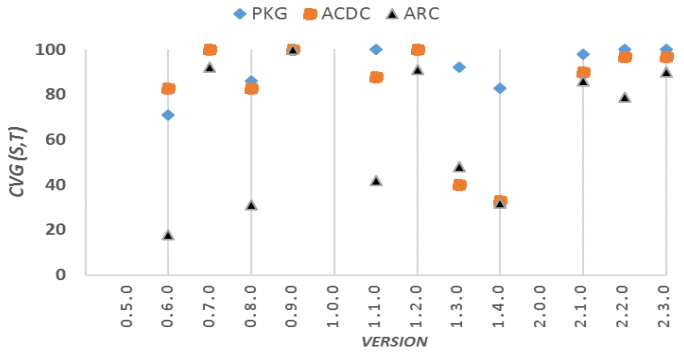

Fig. 13: $cvg(s,t)$ values between minor versions of Ivy

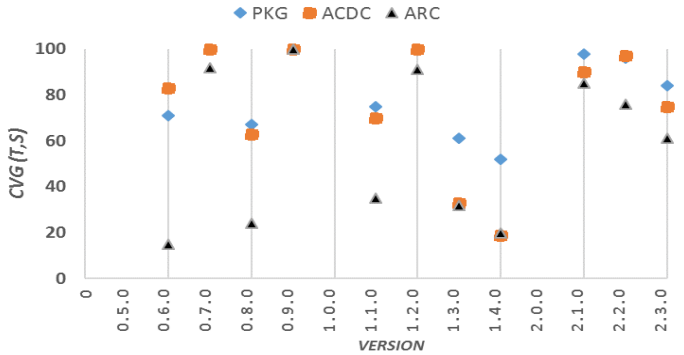

Fig. 14: $cvg(t,s)$ values between minor versions of Ivy

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 taken into account by *ARC* than in either of the structural views.

On the other hand, in the non-Apache systems used in this study, we did not find any instances in which the values of the *a2a* and *cvg* metrics led to contradicting conclusions. We attribute this to the desired high stability of non-Apache systems, which is also reflected in their architectures.

7.4 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 3 and 5. 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 7: Minimum *a2a* values between minor versions

(a) Apache subject systems

System	*ACDC*	*ARC*	*PKG*
Accumulo	83	83	84
ActiveMQ	86	78	84
Cassandra	60	55	49
Chukwa	72	73	76
Hadoop	57	51	72
HttpClient	71	70	74
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
Poi	72	67	70
Struts2	79	80	83
Tika	85	80	86
Xerces	41	37	48
AVG	**74**	**70**	**73**
DEV	**15**	**15**	**14**

(b) Non-Apache subject systems

System	*ACDC*	*ARC*	*PKG*
Druid	-	-	-
Guava	-	-	-
Jackson-databind	94	89	95
PgJDBC	91	80	91
TestNG	77	71	78
AVG	**87**	**80**	**88**
DEV	**7**	**7**	**7**

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 7a and 7b show the *a2a* results of that analysis on architectures produced by *ACDC*, *ARC* and *PKG*. In our dataset, Druid and Guava do not have minor versions. Therefore, the cells of those systems in Table 7b are empty.

We first discuss the Apache systems, shown in Table 7a. Several values in the table indicate that considerable architectural change can indeed occur between two minor versions (e.g., 47% for *ACDC* in JSPWiki; 37% for *ARC* in Xerces). 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*; 75% for *ARC*; 79% for *PKG*, as shown in Table 3). 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. In addition to the substantial system changes that are likely at early stages of development, this may also have occurred due to the lack of clear and consistent versioning guidelines.

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 do not always base their versioning schemes on the architectural impact of their changes, be it because they are not aware of said impact or because they do not consider it relevant enough to be reflected in the versioning scheme.

The minimum a2a values for the non-Apache subject systems are shown in Table 7b. Although the architectures of these systems are highly stable, in some instances big architectural changes are noticeable (e.g., 71% for *PKG* in TestNG). However, the minimum $a2a_{Minor}$ values are relatively high compared to the Apache systems. In addition, the standard deviation values of non-Apache systems (7% in Table 7b) are smaller than standard deviation values of Apache systems (14%-15% in Table 7a). This reinforces the inference that developers of those three systems, all of which are libraries, care about stability and backward compatibility, and are likely to maintain the system's architecture stable across a single major version.

8 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).

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 (Garcia et al., 2013a). Furthermore, we complemented these techniques with *PKG*, which implements an objective measure of a system's "implementation architecture" (Kruchten, 1995). These three techniques are developed independently of one another and use very different strategies for recovering an architecture. This characteristic of our work, 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 (i.e., *cvg*) based on a previously validated metric (Garcia et al., 2013a). We have evaluated *a2a* by applying it on a large number of scenarios; manually inspecting its results; and comparing it to the widely used MoJoFM metric, especially in those cases when MoJoFM yielded counterintuitive results.

9 Related Work

Software evolution has been studied extensively at the code level, dating back several decades (e.g., Lehman's laws (Lehman, 1980)). We will highlight a number of examples that have influenced our work. Godfrey and Tu (Godfrey and Tu, 2000) discovered that Linux's already large size did not prevent it from continuing to grow quickly. Eick et al. (Eick et al., 2001) found a reduction in modularity over the 15-year evolution of software for a telephone

switching system. Murgia et al. (Murgia et al., 2009) 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. In our recent work Le et al. (2016), and (Langhammer et al., 2016), we have shown the benefit of studying architectural evolution by proposing a framework to correlate a software system's qualities with the evolution of its architecture.

Several studies (D'Ambros et al., 2008), (Nakamura and Basili, 2005), (Holt and Pak, 1996), (Wettel and Lanza, 2008), (Tu and Godfrey, 2002) 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. (D'Ambros et al., 2008) 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. We quantitatively study architectural evolution by introducing metrics that compare a pair of architectural artifacts recovered from the implementation of two different versions of a software system and produce a single similarity value for each pair.

Nakamura et al. (Nakamura and Basili, 2005) 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. While the metric allows nodes to be assigned to entities as small as individual statements, they advise the use of larger units, such as files, classes, interfaces or methods.

Holt and Pak (Holt and Pak, 1996) present an approach for visualizing architectural evolution, and apply it to 11 versions of an industrial system. Their study represents architectural changes by visualizing the difference between architectural facts (subsystems and relations) of different versions of a software systems. Unlike our work, they study architectural evolution from a prescriptive approach (as-conceived) instead of descriptive approach (as-implemented).

Wettel et al. (Wettel and Lanza, 2008) utilize visualization techniques to render coarse-grained (at the level packages and classes) and fine-grained (at the level of methods methods) structural evolution in object-oriented software systems. Unlike our work, they study software evolution as reflected in the structure of implementation level entities (packages, classes, and methods) rather than a system's architecture.

Tu and Godfrey (Tu and Godfrey, 2002) present an integrated approach, Beagle, to analyze the evolution of a software system's architecture. They introduce two techniques to implement "Origin Analysis" for detecting the structural changes between two releases of a system: (1) Bertillonage analysis detects the similarity of code fragments based on the euclidean distance of metrics representing and classifying those fragments, and (2) dependency analysis is based on identifying changes in function call patterns. This study can be viewed as complementary to ours, as the metrics used Tu and Godfrey are code-based evolution metrics including basic (e.g., lines of code, lines of comments, and cyclomatic complexity) as well as composite metrics such as S-complexity and D-complexity. Those metrics are not designed to measure the distance between a pair of architectural facts (i.e., the recovered architectures in our study).

A group of studies by Bouwers et al. (Bouwers et al., 2011a, 2013, 2011b) 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 (Garcia et al., 2013a,b) 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 (Garcia et al., 2013b). 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 (Kruchten, 1995), in which packages belong to a system's implementation view.

10 Conclusion and Future Work

This paper has presented the largest study of architectural recovery and architectural evolution to date. The study's scope is reflected in the number of subject systems (23), the total number of examined system versions (931), the total amount of analyzed code (140 MSLOC), the number of applied architecture-recovery techniques (3) resulting in distinct architectural views produced for each system, the number of analyzed architectural models (2793, yielded by the three recovered views per system version), and the number of architectural change metrics (3) applied to each of the 2793 architectural models. This scope was enabled by *ARCADE*, a novel automated workbench for software architecture recovery and analysis. This paper has significantly extended our previous work (Le et al., 2015). Through *ARCADE-Controller*, our approach has the ability to employ cloud-computing power to run large-scale analyses in a reasonable amount of time.

As part of the extensions to our previous work, this paper introduces a comprehensive discussion of two architectural metrics, *a2a* and *cvg*, that enable the study of architectural change at the overall system-level and component-level. This foundation includes (1) the algorithm that computes *a2a* and (2) the mathematical properties of both *a2a* and *cvg* that are relevant for studying architectural change. *a2a* overcomes a critical shortcoming of the widely used MoJoFM metric (Wen and Tzerpos, 2004) that makes it ill-suited for a study of this kind.

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 (Garcia et al., 2013b) that the gross-level organization of a system's implementation (i.e., PKG) is, by itself, not an adequate representation of the system's architecture (as represented by *a2a*). This is especially magnified in cases where the overall implementation architecture (i.e., *cvg* for PKG) remained very stable while, in fact, the system experienced significant growth (*a2a* for PKG). 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 (Garcia et al., 2013a), 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 (Taylor et al., 2009), their interfaces, and their concerns). We are currently complementing the study described in this paper with an analysis of the decay (Perry and Wolf, 1992) found in architectures as they change over time. To this end, we have recently added six new metrics and a dozen architectural smell types 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. In an ongoing research thread, we aim to leverage *ARCADE* to understand and predict a system's non-functional properties by converting its recovered architecture into analyzable models used in DomainPro (Shahbazian et al., 2016). Furthermore, we intend to enrich *ARCADE*'s tool set by adding various existing code-level analyses to it. This integration will provide *ARCADE* users with additional, large-scale program analyses that are deployable onto the cloud. 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

Agnew B, Hofmeister C, Purtilo J (1994) Planning for change: A reconfiguration language for distributed systems. Distributed Systems Engineering 1(5):313

Amazon (2015) Amazon command line interface. https://aws.amazon.com/cli/

Apache (2014a) Apache portable runtime versioning. http://apr.apache.org/versioning.html

Apache (2014b) Hadoop releases. http://hadoop.apache.org/releases.html#News

Apache (2014c) Lucene wiki. http://en.wikipedia.org/wiki/Lucene

Apache (2015a) Apache ant. http://ant.apache.org/

Apache (2015b) Apache maven. http://maven.apache.org/

ARCADE (2015) arcade:start [USC SoftArch Wiki]. http://softarch.usc.edu/wiki/doku.php?id=arcade:start

Bitbucket (2015) Bitbucket. https://bitbucket.org

Blei DM (2012) Probabilistic topic models. Communications of the ACM 55(4):77–84

Bouwers E, Correia JP, van Deursen A, Visser J (2011a) Quantifying the analyzability of software architectures. In: Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on, IEEE, pp 83–92

Bouwers E, van Deursen A, Visser J (2011b) Dependency profiles for software architecture evaluations. In: Software Maintenance (ICSM), 2011 27th IEEE International Conference on, IEEE, pp 540–543

Bouwers E, Deursen Av, Visser J (2013) Evaluating usefulness of software metrics: an industrial experience report. In: ICSE, IEEE Press, pp 921–930

Chatzigeorgiou A, Manakos A (2010) 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, pp 106–115

D'Ambros M, Gall H, Lanza M, Pinzger M (2008) Analysing software repositories to understand software evolution. In: Software Evolution, Springer, pp 37–67

Ducasse S, Pollet D (2009) Software architecture reconstruction: A process-oriented taxonomy. Software Engineering, IEEE Transactions on 35(4):573–591

Eick SG, Graves TL, Karr AF, Marron JS, Mockus A (2001) Does code decay? assessing the evidence from change management data. Software Engineering, IEEE Transactions on 27(1):1–12

Garcia J, Popescu D, Mattmann C, Medvidovic N, Cai Y (2011) Enhancing architectural recovery using concerns. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, pp 552–555

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

Garcia J, Ivkovic I, Medvidovic N (2013a) A comparative analysis of software architecture recovery techniques. In: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, IEEE, pp 486–496

Garcia J, Krka I, Mattmann C, Medvidovic N (2013b) Obtaining ground-truth software architectures. In: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, pp 901–910

Ghezzi G, Gall HC (2013) Replicating mining studies with sofas. In: Proceedings of the 10th Working Conference on Mining Software Repositories, IEEE Press, pp 363–372

Git (2014) Git log. `http://git-scm.com/docs/git-log`

Git (2015) Github. `https://github.com`

Godfrey MW, Tu Q (2000) Evolution in open source software: A case study. In: Software Maintenance, 2000. Proceedings. International Conference on, IEEE, pp 131–142

Google (2015a) Google cloud platform. `https://cloud.google.com`

Google (2015b) Guava. `https://code.google.com/p/guava-libraries/`

Holt R, Pak JY (1996) Gase: visualizing software evolution-in-the-large. In: Reverse Engineering, 1996., Proceedings of the Third Working Conference on, IEEE, pp 163–167

Kim M, Sazawal V, Notkin D, Murphy G (2005) An empirical study of code clone genealogies. In: ACM SIGSOFT Software Engineering Notes, ACM, vol 30, pp 187–196

Koschke R (2005) What architects should know about reverse engineering and rengineering. In: null, IEEE, pp 4–10

Koschke R (2009) Architecture reconstruction. In: Software Engineering, Springer, pp 140–173

Kruchten PB (1995) The 4+ 1 view model of architecture. Software, IEEE 12(6):42–50

Langhammer M, Shahbazian A, Medvidovic N, Reussner R (2016) Automated extraction of rich software models from limited system information. In: Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture (WICSA), IEEE

Le DM, Behnamghader P, Garcia J, Link D, Shahbazian A, Medvidovic N (2015) An empirical study of architectural change in open-source software systems. In: Proceedings of the 12th Working Conference on Mining Software Repository (MSR 2015)

Le DM, Carrillo C, Capilla R, Medvidovic N (2016) Relating architectural decay and sustainability of software systems. In: Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture (WICSA), IEEE

Lehman MM (1980) Programs, life cycles, and laws of software evolution. Proceedings of the IEEE

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

Mahajan S, Li B, Behnamghader P, Halfond WG (2016) Using visual symptoms for debugging presentation failures in web applications. In: Proceeding of the 9th IEEE International Conference on Software Testing, Verification, and Validation (ICST)

Maqbool O, Babri H, et al. (2007) Hierarchical clustering for software architecture recovery. Software Engineering, IEEE Transactions on 33(11):759–780

McCallum A (2002) Mallet: A machine learning for language toolkit

Medvidovic N (1996) Adls and dynamic architecture changes. In: Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints' 96) on SIGSOFT'96 workshops, ACM, pp 24–27

Mengué O (2014) Svn graph branches. `https://code.google.com/p/svn-graph-branches/`

Munkres J (1957) Algorithms for the assignment and transportation problems. Journal of the Society for Industrial and Applied Mathematics 5(1):32–38

Murgia A, Concas G, Pinna S, Tonelli R, Turnu I (2009) 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

Nakamura T, Basili VR (2005) Metrics of software architecture changes based on structural distance. In: Software Metrics, 2005. 11th IEEE International Symposium, IEEE, pp 24–24

Oreizy P, Medvidovic N, Taylor RN (1998) Architecture-based runtime software evolution. In: Proceedings of the 20th international conference on Software engineering, IEEE Computer Society, pp 177–186

Perry DE, Wolf AL (1992) Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes 17(4):40–52

PMD (2015) Pmd documentation. `http://pmd.sourceforge.net`

Robles G (2010) Replicating msr: A study of the potential replicability of papers published in the mining software repositories proceedings. In: Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on, IEEE, pp 171–180

Shahbazian A, Edwards G, Medvidovic N (2016) An end-to-end domain specific modeling and analysis platform. In: Software Engineering (ICSE), 2016 IEEE/ACM 38th IEEE International Conference on, IEEE

Shirali S, Vasudeva HL (2005) Metric spaces. Springer Science & Business Media

Struts (2014) Struts wiki. `http://en.wikipedia.org/wiki/Apache_Struts`

Taylor R, Medvidovic N, Dashofy E (2009) Software Architecture: Foundations, Theory, and Practice

Tu Q, Godfrey MW (2002) An integrated approach for studying architectural evolution. In: Program Comprehension, 2002. Proceedings. 10th International Workshop on, IEEE, pp

127–136
Tzerpos V, Holt RC (1999) Mojo: A distance metric for software clusterings. In: Reverse Engineering, 1999. Proceedings. Sixth Working Conference on, IEEE, pp 187–193
Tzerpos V, Holt RC (2000) Acdc: An algorithm for comprehension-driven clustering. In: wcre, IEEE, p 258
Van Deursen A, Hofmeister C, Koschke R, Moonen L, Riva C (2004) Symphony: View-driven software architecture reconstruction. In: Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on, IEEE, pp 122–132
Wen Z, Tzerpos V (2004) An effectiveness measure for software clustering algorithms. In: Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on, IEEE, pp 194–203
Wettel R, Lanza M (2008) Visual exploration of large-scale system evolution. In: Reverse Engineering, 2008. WCRE'08. 15th Working Conference on, IEEE, pp 219–228
Xing EP, Jordan MI, Russell S, Ng AY (2002) Distance metric learning with application to clustering with side-information. In: Advances in neural information processing systems, pp 505–512