A Comparative Analysis of Software Architecture Recovery Techniques

Joshua Garcia* Igor Ivkovic† Nenad Medvidovic*

*Computer Science Department
University of Southern California,
Los Angeles, CA 90089, USA
{joshuaga,neno}@usc.edu

† Wilfrid Laurier University
75 University Avenue West,
Waterloo, ON, N2L 3C5, Canada
iivkovic@wlu.ca

***Abstract*—Many automated techniques of varying accuracy have been developed to help recover the architecture of a software system from its implementation. However, rigorously assessing these techniques has been hampered by the lack of architectural "ground truths". Over the past several years, we have collected a set of eight architectures that have been recovered from open-source systems and independently, carefully verified. In this paper, we use these architectures as ground truths in performing a comparative analysis of six state-of-the-art software architecture recovery techniques. We use a number of metrics to assess each technique for its ability to identify a system's architectural components and overall architectural structure. Our results suggest that two of the techniques routinely outperform the rest, but even the best of the lot has surprisingly low accuracy. Based on the empirical data, we identify several avenues of future research in software architecture recovery.**

I. Introduction

The effort and cost of software maintenance dominate the activities in a software system's lifecycle. Understanding and updating a system's software architecture is a critical facet of maintenance. However, the maintenance of an architecture is challenged by the related phenomena of architectural *drift* and *erosion* [47]. These phenomena are caused by careless, unintended addition, removal, and modification of architectural design decisions. To deal with drift and erosion, sooner or later engineers are forced to *recover* a system's architecture from its implementation. A number of techniques have been proposed to aid architecture recovery [19]. The existing techniques however are known to suffer from inaccuracies, and different techniques typically return different results as "the architecture" for the same system. In turn, this can lead to

- *difficulties in assessing* a recovery technique, which makes it unclear how to identify the best technique for a given recovery scenario;
- *risks in relying* on a particular technique, because its accuracy is unknown; and
- *flawed strategies for improving* a technique, because the needed baseline assessments are missing.

Previous comparative studies [9], [28], [6], [36], [51], [24]—while informative and providing preliminary hints as to the respective quality of different recovery techniques—have been limited in a number of ways. First, these analyses disagree as to which techniques are most accurate. Second, they do not elaborate the conditions under which each technique excels or falters. Third, the quality of the "ground truths" on which these studies base their conclusions is questionable. Fourth, previous comparative analyses have been limited in terms of scale (i.e., the numbers and sizes of systems selected, or the number of techniques evaluated).

To better understand the accuracy of existing architecture recovery techniques and to address the shortcomings encountered in previous comparative studies, this paper presents a *comparative analysis of six automated architecture recovery techniques*: Algorithm for Comprehension-Driven Clustering (ACDC) [48], Architecture Recovery using Concerns (ARC) [22], Bunch [35], scaLable InforMation BOttleneck (LIMBO) [6], Weighted Combined Algorithm (WCA) [28], and a technique developed by Corazza et al. [16], which we will refer to as zone-based recovery (ZBR) in this paper. We have selected these techniques because they have been shown to be accurate based on prior work and have been evaluated in several previous publications. The six selected techniques rely on two kinds of input obtained from implementation-level artifacts: *textual* and *structural*. Textual input refers to the words found in the source code and comments of implementation-level entities. Structural inputs are the control-flow-based and/or data-flow-based dependencies between implementation-level entities. Most work on automated component recovery has focused on structural input. However, recent work has increasingly included textual input as a way of improving the accuracy of architecture recovery techniques.

We assess the accuracy of these techniques on eight architectures from six different open-source systems: ArchStudio [17], Bash [14], Hadoop [4], Linux [13], Mozilla [2], and OODT [29]. In the case of two of the systems—Linux and Mozilla—we use a pair of architectures each, at different levels of detail. These six systems span a number of application domains, are implemented in three different programming languages (C, C++, Java), two different programming paradigms (procedural and object-oriented), and greatly vary in size (from 70KSLOC TO 4MSLOC). The systems' eight architectures comprise our ground truth. Four of the architectures—those of ArchStudio, Bash, Hadoop, and OODT—were verified as correct by one or more architects of the corresponding systems [20]; the remaining four architectures—two each of

978-1-4799-0215-6/13 © 2013 IEEE

ASE 2013, Palo Alto, USA
Experience Track

Linux and Mozilla—are a result of a meticulous recovery process conducted by other researchers and were used in previous evaluations of recovery techniques [6]. Our assessment is performed at the system-wide level and at the level of individual system components.

We performed extensive groundwork to position ourselves to conduct this comparative analysis. Over the span of a decade, we developed, evaluated, and refined a semi-automated software architecture recovery technique, called Focus [18], [33]. We subsequently used Focus to recover the architectures of 18 open-source platforms for scientific computing and processing large datasets [30]. This work informed and aided us in the development of a more recent approach [21] for obtaining the architecture of a software system that can be relied on as its ground truth. We applied that approach in obtaining four of the ground-truth architectures [20] that we use to evaluate the recovery techniques in this paper. Given the limited or no current availability of several recovery techniques' implementations, we reimplemented three of the six techniques used in this study in consultation with their authors.

Our results indicate that two of the selected recovery techniques are superior to the rest along multiple measures. However, the results also show that there is significant room for improvement in all of the studied techniques. In fact, while the accuracy of individual techniques varies across the different subject systems, on the whole the techniques performed surprisingly poorly. We discuss the threats to our study's validity, the possible reasons behind our results, and several possible avenues of future research in automated architecture recovery.

The remainder of the paper is organized as follows. Section II outlines the related work. Section III describes the architecture recovery techniques used in our study. Section IV delineates the details of the study and interprets the obtained results. Section V discusses lessons learned. Section VI describes threats to validity. Finally, Section VII concludes and considers future research opportunities.

II. Related Work

Three recent surveys together provide a detailed overview of software architecture recovery techniques. Ducasse and Pollet [19] provide a practitioner-oriented survey of over thirty existing approaches for software architecture recovery. Their survey is organized according to the goals, processes, inputs, techniques, and outputs of the recovery approaches. Koschke presents a tutorial and survey [25] of architecture recovery organized around *Symphony*, a conceptual framework based on the use of architectural views and viewpoints. Maqbool and Babri present a survey on the use of hierarchical clustering techniques for architecture recovery [28]. They compare different measures and hierarchical clustering algorithms, and discuss the resulting research issues and trends.

Automated software architecture techniques have been around for over three decades [11], [23], [43], [44]. Most of them group implementation-level entities (e.g., files, classes, or functions) into clusters, where each cluster represents a component [25], [28], [50]. Some existing techniques search for specific patterns, usually relating to structural dependencies between entities, to identify components [48], [41], [42], while others have used concept analysis [46], [19] for the same purpose. Another class of techniques tries to partition system entities and their dependencies into components in a manner that maximizes some objective function [35], [38], [24]. Another set of techniques employs hierarchical clustering [6], [28], [36], which allows a recovered architectural view to be shown at multiple levels of abstractions. These techniques iteratively obtain components by combining similar entities at each step.

While a majority of the existing recovery techniques rely on structural input to identify components [19], over time researchers have also attempted to include non-structural information, such as directory paths and file authorship [7], [8], [34], [22]. Recent work has focused on utilizing textual input [15], [16], [34], [22] obtained from source code and comments. The resulting techniques combine textual input with other commonly-used architecture recovery mechanisms, such as objective function maximization or hierarchical clustering.

Most recovery techniques are evaluated individually for their accuracy or other quality attributes. In certain cases, techniques have been evaluated against other techniques. Anquetil and Lethbridge [9] compare generic clustering algorithms that are not specifically designed for architecture recovery. They utilize a combination of structural and non-structural input to obtain architectures. Other comparative analyses focus on techniques specifically designed for architecture recovery. WCA [28], [36] has been evaluated against LIMBO [6], [28], [36]. WCA has two variations that differ based on the measures used for determining the similarity between entities: Unbiased Ellenberg (UE) and Unbiased Ellenberg-NM (UENM). LIMBO has been shown to be generally superior to WCA-UE, but has been outperformed by WCA-UENM. LIMBO, ACDC [48], and Bunch [6] have been compared to generic clustering techniques [6]. In this study, LIMBO and ACDC were shown to be the most accurate. Another study compared ACDC, Bunch, and generic clustering techniques [51]. The complete linkage (CL) clustering algorithm was shown to be generally the most accurate, while ACDC was shown to have limited accuracy. WCA was shown to be superior to CL in yet another study [27]. Finally, Software Architecture Finder (SArF) [24], another technique that maximizes an objective function, has recently been compared against ACDC and Bunch [24]. A software system's package structure was used as the ground truth in this comparison. SArF showed better accuracy than ACDC and Bunch in the study.

It can be seen from the above summary that different studies conflict in their conclusions. We postulate that part of the reason for this is that each of these comparative analyses has been limited in one or more important ways. Each study has relied on a very small number of subject systems or recovery techniques, or both. Comparative analyses that have included a wider variety of techniques have done so by relying on generic clustering techniques that are not

specifically designed for architecture recovery. Comparisons involving techniques actually designed for architecture recovery typically only include structural input. Furthermore, previous evaluations have either relied on a few "home-grown" subject systems, which tend to be small, or have had to rely on larger, third-party systems whose actual ("ground-truth") architectures are not fully known. In some cases, researchers have used directory or package structure as a widely available architectural proxy. However, our recent study [20] has shown that in most real systems, architects do not consider the directory or package structure to be an accurate reflection of a system's architecture. Finally, the above studies give little guidance as to the conditions under which a recovery technique excels or falters. Our work described in this paper tries to address each of these shortcomings.

III. Selected Recovery Techniques

For our comparative analysis, we have selected the following six software architecture recovery techniques:

- Algorithm for Comprehension-Driven Clustering (ACDC) [48],
- Weighted Combined Algorithm (WCA) [27], and its two measurements WCA-UE and WCA-UENM [28],
- scaLable InforMation BOttleneck (LIMBO) [6],
- Bunch [26],
- Zone-Based Recovery (ZBR) [15], [16] and two of its variants,
- Architecture Recovery using Concerns (ARC) [22].

The selected techniques have been previously published, they are automated, and they have been designed specifically for architecture recovery. Each technique provides either a reference implementation or the details of its underlying algorithms. Finally, the techniques have been shown to be accurate in previous evaluations. The six techniques provide a broad cross-section in that they differ in the underlying mechanisms employed and type of information utilized to derive architectural details. Note that we have chosen not to include the recent SArF recovery technique [24] because its evaluation to date has not assessed its accuracy beyond recovering the package structure of Java applications. We elaborate more on each selected technique next.

ACDC recovers components using patterns. ACDC was introduced by Tzerpos and Holt [48], who observed that certain patterns for grouping implementation-level entities recur. These patterns include grouping together *(1)* implementation-level entities that reside in the same source file, *(2)* entities in the same directory, *(3)* entities from the associated body and header files (e.g., .h and .c files in C), *(4)* entities that are leaves in a system's graph, *(5)* entities that are accessed by the majority of subsystems, *(6)* entities that depend on a large number of other resources, and *(7)* entities that belong to a subgraph obtained through dominance analysis. This last pattern, called the *subgraph dominator pattern*, clusters entities that are dominated by a given node together with the dominator node itself. ACDC has been included in previous comparative analyses, and its accuracy has been evaluated and confirmed in previous experiments [6], [28], [51].

WCA is a hierarchical clustering technique. In a clustering technique, each entity to be clustered is represented by a feature vector $\mathbf{v} = \{f_i \mid 1 \leq i \leq n\}$, which is an n-dimensional vector of numerical features. n is the number of implementation-level entities in the system. A *feature* f_i is a property of an entity. In WCA, a feature f_i of an entity e_j indicates whether or not e_j depends on another entity e_i. Each feature in a feature vector corresponds to a different entity. Therefore, a feature vector of an entity e in WCA indicates all the entities that e depends on. For example, consider a system with just three entities: e_1, e_2, and e_3. For this system, e_1 will have the feature vector $\{1,0,1\}$ if e_1 depends on itself and e_3 but not on e_2.

A hierarchical clustering technique, like WCA, begins by placing each implementation-level entity in its own cluster, where a cluster represents an architectural component. The technique computes the pair-wise similarity between all the clusters and then combines the two most similar clusters into a new cluster. This is repeated until all elements have been clustered or the desired number of clusters is obtained. When two clusters are merged by WCA, a new feature vector is formed by combining the feature vectors of the two clusters. Two clusters c_i and c_j with feature vectors v_i and v_j, respectively, are combined into a new feature vector $\mathbf{v_{ij}^{wca}} = \{\frac{\mathbf{v_i}+\mathbf{v_j}}{m_i+m_j} = \frac{f_{ik}+f_{jk}}{m_i+m_j}\}, k = 1,..,n$, where m_i and m_j are the numbers of entities in c_i and c_j, respectively.

To compute similarity between clusters or entities, WCA provides two measures, UE and UENM. The UE measure for entities e_1 and e_2 is defined as $UE = \frac{0.5*sumBoth(e_1,e_2)}{0.5*sumBoth(e_1,e_2)+only(e_1,e_2)+only(e_2,e_1)}$. $sumBoth(e_i,e_j)$ is the sum of features present in both entities. $only(e_i,e_j)$ is the number of entities present in e_i but not in e_j. A greater number of shared features between two entities indicates their increased similarity. UENM is defined as $UENM = \frac{0.5*sumBoth(e_1,e_2)}{0.5*sumBoth(e_1,e_2)+2(only(e_1,e_2)+only(e_2,e_1))+numBoth(e_1,e_2)+notBoth(e_1,e_2)}$. $numBoth(e_i,e_j)$ is the number of features present in both entities. $notBoth(e_i,e_j)$ is the number of features absent from both entities. UENM incorporates more information into the measure, which allows it to better distinguish between entities. WCA's accuracy has been evaluated and confirmed (within the corresponding experiment parameters) in previous comparative analyses [28], [36].

LIMBO is another hierarchical clustering technique, but it differs from WCA in three key ways. First, LIMBO uses a mechanism called *Summary Artifacts* (SA) to reduce the computations needed while minimizing accuracy loss. Second, LIMBO uses the Information Loss (IL) measure to compute similarities between entities, which is defined as $\delta I = (p(c_i) + p(c_j)) * D_{js}(\mathbf{v_i}, \mathbf{v_j})$. $p(c) = \frac{m_c}{n}$ is the probability of a cluster c, where m_c is the number of entities in the cluster. D_{js} is the Jensen-Shannon divergence, which computes the symmetric distance between two probability distributions. Each feature vector is a probability distribution over features. D_{js} is defined as $D_{js} = p(c_i)/p(c_{ij}) * D_{kl}(\mathbf{v_i}||\mathbf{v_{ij}}) + p(c_j)/p(c_{ij}) * D_{kl}(\mathbf{v_j}||\mathbf{v_{ij}})$.

c_{ij} is the cluster resulting from combining clusters c_i and c_j. v_{ij} is c_{ij}'s associated feature vector. $D_{kl}(\mathbf{v_i}||\mathbf{v_j})$ is the Kullback-Leibler divergence, which is a non-symmetric distance measure for probability distributions, and is defined as $D_{kl}(\mathbf{v_i}||\mathbf{v_j}) = \sum_{k=1}^{n} f_{ik} * log(\frac{f_{ik}}{f_{jk}})$. Lastly, LIMBO associates a new feature vector for a combined cluster c_{ij} using the following formula: $\mathbf{v_{ij}^{lim}} = \{\frac{m_i\mathbf{v_i}+m_j\mathbf{v_j}}{m_i+m_j}\}$. LIMBO has been included in previous comparative analyses [6], [28], [36] and has demonstrated significant accuracy.

Bunch uses a hill-climbing algorithm to find a partitioning of entities into clusters that maximizes an objective function. Bunch initially starts with a random partition and stops when it can no longer find a better partition. The objective function used in Bunch is called Modularization Quality (MQ) and is defined as $MQ = \sum_{i=1}^{k} CF_i$. k is a partition's number of clusters. CF_i is the "cluster factor" of cluster i, representing its coupling and cohesion, and is defined as

$$CF_i = \begin{cases} 0 & \mu_i = 0 \\ \frac{2\mu_i}{2\mu_i + \sum_{\substack{j=1 \\ j\neq i}}^{k} (\varepsilon_{i,j} + \varepsilon_{j,i})} & otherwise \end{cases}$$

μ_i is the number of edges within the cluster, which measures cohesion. $\varepsilon_{i,j}$ is the number of edges from cluster i to cluster j, which measures coupling. We analyze two variations of Bunch's hill-climbing algorithm: next ascent hill climbing (NAHC) and steep ascent hill climbing (SAHC). In each step of the NAHC algorithm, the first neighbor of the current partition that improves MQ is chosen. In each step of the SAHC algorithm, the entire set of neighboring partitions to the current partition is examined to find the partition that improves MQ by the largest margin. Bunch has been used in previous comparative analyses [6], [24], [51], although its accuracy has mostly shown to be limited compared to other techniques. For our comparative analysis, we have considered both variants of Bunch—NAHC and SAHC—to obtain the most accurate clustering that this technique can provide.

ZBR, or zone-based recovery, is a technique developed by Corrazza et al. [15], [16], which utilizes textual information, hierarchical clustering, and a weighting scheme for feature vectors. We have selected the latest formulation of Corazza et al.'s technique [16]. The textual information used in ZBR tries to infer a software system's semantics when recovering the system's architecture. ZBR has been shown to work well in recovering the package structure of Java systems [16].

In ZBR, each source file is considered a *document*, i.e., a bag of words, where each word is obtained from program identifiers and comments. Each of these documents is divided into zones, which are regions in which a word may reside. For a source file, there can be up to six zones: class names, attribute names, function names, parameter names, comments, and bodies of functions. Each word is scored a *term frequency (tf) – inverse document frequency (idf)* value, defined as $tf\text{-}idf(t,d) = tf(t,d) \times idf(t,d)$. For a document d, tf for a term t is the number of occurrences of t in d. For a set of D documents, idf is defined as $idf(t,D) = log\frac{|D|}{|\{d\in D|t\in d\}|}$. The denominator of the *log* function of idf gives the number of documents in which t is found. Thus, a *tf-idf* value is greater for a term that appears more frequently in a single document and lower as a term appears in multiple documents.

Each term in a document gets a different tf-idf value for each zone; each zone is weighted using the Expectation-Maximization (EM) algorithm for Gaussian Mixtures [32]. ZBR can vary based on the inputs to EM for the zone weights [16]. In our analysis, we first use uniform zone weights as input to EM (referred to as ZBR-UNI hereafter), and then vary the zone weights by the number of tokens, i.e., word instances, in a zone divided by the total number of tokens in the system (referred to as ZBR-TOK).

Clusters in ZBR consist of source files and each source file's feature vector consists of the tf-idf values for each weighted zone. ZBR computes similarity between entities using cosine similarity. Cosine similarity is defined as $sim_{cos}(\mathbf{v_1},\mathbf{v_2}) = \frac{\mathbf{v_1}\cdot\mathbf{v_2}}{\|\mathbf{v_1}\|\|\mathbf{v_2}\|}$. The numerator of sim_{cos} is the dot product of the two feature vectors; the denominator is the product of the magnitude of the two feature vectors.

ARC is the final architecture recovery technique used in our study. It has been developed as part of our on-going research. ARC recovers concerns of implementation-level entities and uses a hierarchical clustering technique to obtain architectural elements. ARC recovers concerns using the statistical language model LDA [12], which is obtained from the identifiers and comments in a system's source code. LDA allows ARC to compute similarity measures between concerns and identify which concerns appear in a single implementation-level entity. Thus, ARC, like ZBR and unlike the other techniques, attempts to rely on a program's semantics to perform recovery.

Similarly to ZBR, ARC represents a software system as a set of *documents*. A document can have different topics, which are the concerns in ARC. A *topic* z is a multinomial probability distribution over words w. A document d is represented as a multinomial probability distribution over topics z (called the document-topic distribution). Each implementation-level entity is treated as a document where its document-topic distribution is its feature vector. Hierarchical clustering is performed by computing similarities between entities using the Jensen-Shannon divergence (D_{js}), which allows computing similarities between document-topic distributions.

IV. Comparing Recovery Techniques

We aim to meet two objectives with our analysis:

- **OBJ1:** Determine which techniques, if any, generally work better than others.
- **OBJ2:** Determine under what conditions each technique excels or falters.

To that end, we compare recovery techniques in terms of

- their overall accuracy (Section IV-B),
- the extent to which a cluster (i.e., architectural component) produced by a technique resembles a cluster produced by a human expert (Section IV-C), and

- the extent to which a recovery criterion (similarity measure, objective function, or pattern) implemented by a technique is reflected in a given cluster (Section IV-D).

A. Subject Systems and Implementation Details

Table I provides an overview of the eight architectures against which we assessed the selected recovery techniques, and the six systems to which those architectures belong. We consider these to be "ground-truth" architectures because they have been recovered from a system by one or more experts. An expert in this context is an engineer with intimate knowledge of the system, its architectural design, and/or its implementation. Table I summarizes each *System* from which we have obtained an architecture; the system's *Ver*sion; application *Dom*ain; primary implementation *Lang*uage; size in terms of *SLOC*; and the number of *Comp*onents in the architecture. The eight architectures vary by size, domain, implementation language, and programming paradigm (procedural vs. OO).

We produced the ground-truth architectures of ArchStudio, Bash, Hadoop, and OODT as part of our prior work, with the assistance of those systems' architects [21], [20]. We obtained the architectures of Linux and Mozilla from the authors of previous studies involving the two systems' architecture recoveries [6], [48]. For both of these systems, we were provided nested architectures in which individual components comprised subcomponents. Using Shtern and Tzerpos's method [45], we obtained two non-hierarchical variations for Linux and Mozilla: a compact architecture of coarser granularity (denoted by the suffix *-C* in Table I) and a detailed architecture of finer granularity (denoted by the suffix *-D*).

We obtained the implementations of three of the selected architecture recovery techniques—ACDC, Bunch, and ARC—from their original authors. On the other hand, working implementations of WCA, LIMBO, and ZBR were unavailable. We re-implemented those techniques based on their published documentation, with guidance provided by the techniques' authors. ZBR is implemented in Python, while the other techniques are implemented in Java. ZBR uses gensim for computing tf-idf values [40], and scikit-learn [37] for its EM algorithm for Gaussian mixtues. We obtain module dependencies using the Class Dependency Analyzer (CDA) [5] for the subject systems implemented in Java and the `mkdep` tool [3] for the systems implemented in C and C++. Finally, ARC uses topic models from MALLET [31] to represent concerns.

B. Overall Accuracy

We assess the overall accuracy of the selected techniques using a widely-used measure in architecture recovery called MoJoFM [49]. MoJoFM allows us to determine which techniques are generally more accurate than others (OBJ1) since it provides a system-wide measure. Furthermore, by comparing the obtained MoJoFM results, we can determine system-wide conditions under which techniques excel or falter (OBJ2).

TABLE I: Summary information about the subject systems

System	Ver	Dom	Lang	SLOC	Comps
ArchStudio	4	IDE	Java	280K	54
Bash	1.14.4	OS Shell	C	70K	25
Hadoop	0.19.0	Data Processing	Java	200K	68
Linux-C	2.0.27	OS	C	750K	7
Linux-D	2.0.27	OS	C	750K	120
Mozilla-C	1.3	Browser	C/C++	4M	10
Mozilla-D	1.3	Browser	C/C++	4M	233
OODT	0.2	Data Management	Java	180K	217

MoJoFM is a distance measure between two architectures expressed as a percentage. This measure is based on two key operations used to transform one architecture into another: moves (*Move*) of entities from one cluster to another cluster and merges (*Join*) of clusters. MoJoFM is defined as:

$$MoJoFM(A,B) = (1 - \frac{mno(A,B)}{max(mno(\forall A,B))}) \times 100\%$$

A is the architecture produced by a given recovery technique. B is the ground-truth architecture against which the technique is being assessed. $mno(A,B)$ is the minimum number of Move and Join operations needed to transform A into B. Therefore, MoJoFM quantifies the amount of effort required to transform one architecture into another. A 100% MoJoFM value indicates full correspondence between A and B, while a 0% MoJoFM value indicates that the two architectures are completely different.

Table II depicts the obtained MoJoFM results. A MoJoFM value is given for each pair comprising a subject *System*'s architecture and a recovery technique. For techniques whose numbers of clusters vary—ARC, WCA, and LIMBO—the table shows the results for the numbers of clusters that maximize the corresponding MoJoFM values. To obtain the maximum MoJoFM value for a given $(architecture, technique)$ pair, we selected a range of clusters and computed MoJoFM values for each resulting architecture. We varied the number of clusters from a single cluster to one half of the number of entities in a particular system. We stopped increasing the number of clusters for a given system once the MoJoFM values for the resulting architectures no longer improved.

ARC can additionally vary in the number of concerns, so in its case we also selected the number of concerns that maximizes the MoJoFM values. We selected as few as 10 concerns and as many as $\frac{1}{3}V$, where V is the number of terms in a system. We stopped generating sets of concerns once we found that increasing the number of concerns no longer improved the MoJoFM values.

We ran Bunch three times for each of its variations since the Bunch algorithm is non-deterministic. From those results, we selected the Bunch results that produced the highest MoJoFM values.

In total, in generating Table II, we computed MoJoFM values for approximately 1,540,000 architectures. The table also shows the average MoJoFM values (*AVG*) for each architecture (rightmost column) and recovery technique (bottom row). Dark gray and light gray highlighting indicates the techniques that obtained, respectively, the highest and second-highest MoJoFM values for a given subject system's architecture.

Some entries indicate MoJoFM errors (*MJE*) or memory errors (*MEM*). WCA and LIMBO produce architectures for

TABLE II: MoJoFM results

System	ARC	ACDC	WCA-UE	WCA-UENM	LIMBO	Bunch-NAHC	Bunch-SAHC	Z-Uni	Z-Tok	AVG
ArchStudio	76.28%	87.68%	49.73%	45.87%	31.20%	59.50%	50.07%	48.53%	39.47%	54.26%
Bash	57.89%	49.35%	41.56%	42.21%	27.27%	47.97%	38.51%	36.97%	36.97%	42.08%
Hadoop	54.28%	62.92%	42.15%	39.57%	19.23%	51.24%	46.95%	36.00%	45.91%	44.25%
Linux-D	51.47%	36.31%	33.51%	32.54%	18.46%	32.54%	31.14%	MEM	MEM	33.71%
Linux-C	75.72%	63.76%	61.98%	59.74%	57.70%	73.65%	75.13%	MEM	MEM	66.81%
Mozilla-D	43.44%	41.20%	MJE	MJE	MJE	40.18%	31.65%	MEM	MEM	39.12%
Mozilla-C	62.50%	60.30%	32.49%	32.40%	34.97%	69.02%	64.29%	MEM	MEM	50.85%
OODT	48.48%	46.01%	43.67%	41.97%	MJE	36.65%	31.56%	30.89%	33.57%	39.10%
AVG	58.76%	55.94%	43.58%	42.04%	31.47%	51.34%	46.16%	38.10%	38.98%	**45.15%**

which MoJoFM does not terminate. We confirmed this error with MoJoFM's creators, but the error has not been resolved at the time of this paper's submission. For the memory errors, ZBR stores a large amount of data that surpasses the memory available on our test hardware (16GB RAM). ZBR must store data of size nzV, where n is the number of entities in the system to be clustered, z is the number of zones, and V is the number of terms. For the other techniques, the data to be stored is typically n^2, where $n \ll V$. Since ZBR must store significantly more data, it does not scale to the two largest systems, Linux and Mozilla.

ARC and ACDC produce the best results in general, with respective average MoJoFM values of 58.76% and 55.94%. Bunch generally obtained higher MoJoFM values with its NAHC variant (51.34% on average) than its SAHC variant (46.16%). Furthermore, several techniques performed well in the cases of the coarse-grained architectures of the two largest systems (Linux-C and Mozilla-C). The average MoJoFM values were 66.81% for Linux-C and 50.85% for Mozilla-C; these values were, respectively, the highest and third-highest across all eight architectures in our study.

The results also indicate that there is still significant room for improvement in each of our selected architecture recovery techniques. For example, the average MoJoFM values in the cases of Linux-D, Mozilla-D, and OODT were all under 40%. These three architectures contain the largest numbers of clusters in our study (between 120 and 233), suggesting that the state-of-the-art recovery techniques generally perform worse at finer granularities. Our recent study indicates that it is particularly important for a recovery technique to handle architectures at finer granularities: while they typically elide many details when representing a system's conceptual architecture, engineers and architects tend to prefer much more detail when that same system's architecture is recovered from the implementation [20].

Even for the recovery techniques that generally obtained better MoJoFM results than others, the range of values varied significantly across the subject systems: 43%–76% for ARC, 36%–88% for ACDC, and 33%–74% for Bunch-NAHC. Combined with the generally poorer results yielded by the remaining three techniques, this degree of variation does not inspire confidence in any one technique: currently, they are simply too unpredictable. We do not recommend trying to optimize these techniques such that they maximize the MoJoFM values for the eight ground-truth architectures we have collected to date, since the number of subject systems is likely too low and not representative of software architectures in general. At the same time, introducing improvements that would minimize the variability across different systems—even the relatively limited sample used in our study—would certainly make a given technique more dependable.

C. Cluster-to-Cluster Comparison

To obtain more specific information about the conditions under which each recovery technique excels or falters (OBJ2), we compare the extent to which each cluster produced by a technique matches each cluster in a ground-truth architecture. We call this comparison a *cluster-to-cluster (c2c)* analysis. To this end, we introduce a measure that indicates the overlap of entities between two clusters:

$$c2c_{meas}(A,B) = \frac{|A \cap B|}{|A|} \times 100\%$$

In our case, A is the set of entities grouped in a cluster by a given technique, while B is the set of entities grouped in a ground-truth cluster. To further assess the overall accuracy of the recovery techniques (OBJ1), we also summarize the results of the c2c analysis across each architecture as a whole.

For ease of analysis, we consider the matching of clusters at three discrete strength levels: moderate, strong, and very strong. A moderate match occurs for $40\% < c2c_{meas} \leq 60\%$; a strong match occurs for $60\% < c2c_{meas} \leq 80\%$; finally, a very strong match occurs for $80\% < c2c_{meas} \leq 100\%$. For example, if a cluster $c1_{arc}$, computed by applying ARC to a system, yields a $c2c_{meas}$ value of 55% when compared to cluster $c5_{gt}$ in the system's ground-truth architecture, then $c1_{arc}$ has a moderate match with $c5_{gt}$.

By comparing matches at these three different strength levels, we aim to determine conditions under which each technique's accuracy varies significantly (OBJ2). It is important to note that a cluster computed by a given technique can match multiple clusters of a ground-truth architecture. Although the converse is also the case, in our analysis we focus primarily on the ability of a technique to group the same entities as a human expert would. As a hypothetical example, consider two ground-truth clusters, $c1_{gt}$ with entities $\{e_1, e_2, e_3\}$ and $c2_{gt}$ with entities $\{e_4, e_5, e_6\}$. Assume that ACDC computes three clusters: $c1_{acdc}$ with entities $\{e_1, e_2\}$, $c2_{acdc}$ with entities $\{e_3, e_4\}$, and $c3_{acdc}$ with entities $\{e_5, e_6\}$. $c2c_{meas}$ of $c1_{acdc}$ and $c1_{gt}$ is 100%, which is a very strong match; the $c2c_{meas}$ values of $c2_{acdc}$ and, both, $c1_{gt}$ and $c2_{gt}$ are 50%, which are moderate matches; finally, $c2c_{meas}$ of $c3_{acdc}$ and $c2_{gt}$ is 100%, which is another very strong match.

TABLE III: Cluster-to-Cluster Analysis, Moderate Match

System	ARC	ACDC	WCA-UE	WCA-UENM	LIMBO	Bunch	Z-Uni	Z-Tok	AVG
ArchStudio	26% (14/54)	33% (18/54)	33% (18/54)	19% (10/54)	11% (6/54)	15% (8/54)	13% (7/54)	11% (6/54)	20%
Bash	20% (5/25)	20% (5/25)	12% (3/25)	8% (2/25)	36% (9/25)	12% (3/25)	20% (5/25)	20% (5/25)	19%
Hadoop	31% (21/68)	13% (9/68)	0% (0/68)	47% (32/68)	1% (1/68)	10% (7/68)	25% (17/68)	22% (15/68)	19%
Linux-D	33% (40/120)	22% (26/120)	13% (15/120)	30% (36/120)	0% (0/120)	9% (11/120)	MEM	MEM	18%
Linux-C	71% (5/7)	57% (4/7)	29% (2/7)	71% (5/7)	14% (1/7)	57% (4/7)	MEM	MEM	50%
Mozilla-D	33% (78/233)	40% (93/233)	2% (5/233)	3% (6/233)	0% (0/233)	16% (37/233)	MEM	MEM	16%
Mozilla-C	100% (10/10)	100% (10/10)	70% (7/10)	70% (7/10)	0% (0/10)	100% (10/10)	MEM	MEM	73%
OODT	29% (62/217)	9% (20/217)	43% (93/217)	28% (61/217)	MJE	4% (9/217)	23% (50/217)	18% (39/217)	22%
AVG	32% (235/734)	25% (185/734)	19% (143/734)	22% (159/734)	3% (17/517)	12% (89/734)	22% (79/364)	18% (65/364)	**19.13%**

TABLE IV: Cluster-to-Cluster Analysis, Strong Match

System	ARC	ACDC	WCA-UE	WCA-UENM	LIMBO	Bunch	Z-Uni	Z-Tok	AVG
ArchStudio	17% (9/54)	33% (18/54)	13% (7/54)	24% (13/54)	15% (8/54)	6% (3/54)	6% (3/54)	7% (4/54)	15%
Bash	12% (3/25)	4% (1/25)	4% (1/25)	4% (1/25)	8% (2/25)	12% (3/25)	4% (1/25)	4% (1/25)	7%
Hadoop	26% (18/68)	7% (5/68)	0% (0/68)	26% (18/68)	0% (0/68)	9% (6/68)	9% (6/68)	9% (6/68)	11%
Linux-D	20% (24/120)	18% (21/120)	3% (3/120)	8% (10/120)	0% (0/120)	5% (6/120)	MEM	MEM	9%
Linux-C	57% (4/7)	86% (6/7)	71% (5/7)	71% (5/7)	0% (0/7)	29% (2/7)	MEM	MEM	52%
Mozilla-D	13% (31/233)	20% (47/233)	0% (0/233)	0% (0/233)	0% (0/233)	6% (13/233)	MEM	MEM	7%
Mozilla-C	80% (8/10)	100% (10/10)	30% (3/10)	30% (3/10)	0% (0/10)	80% (8/10)	MEM	MEM	53%
OODT	9% (19/217)	6% (13/217)	14% (30/217)	13% (29/217)	MJE	0% (1/217)	4% (9/217)	3% (7/217)	7%
AVG	16% (116/734)	16% (121/734)	7% (49/734)	11% (79/734)	2% (10/517)	6% (42/734)	5% (19/364)	5% (18/364)	**8.50%**

TABLE V: Cluster-to-Cluster Analysis, Very Strong Match

System	ARC	ACDC	WCA-UE	WCA-UENM	LIMBO	Bunch	Z-Uni	Z-Tok	AVG
ArchStudio	37% (20/54)	24% (13/54)	17% (9/54)	13% (7/54)	7% (4/54)	2% (1/54)	11% (6/54)	9% (5/54)	15%
Bash	52% (13/25)	44% (11/25)	64% (16/25)	64% (16/25)	0% (0/25)	0% (0/25)	24% (6/25)	24% (6/25)	34%
Hadoop	37% (25/68)	9% (6/68)	0% (0/68)	31% (21/68)	0% (0/68)	4% (3/68)	16% (11/68)	13% (9/68)	14%
Linux-D	48% (58/120)	47% (56/120)	43% (51/120)	44% (53/120)	0% (0/120)	2% (2/120)	MEM	MEM	31%
Linux-C	86% (6/7)	57% (4/7)	86% (6/7)	86% (6/7)	0% (0/7)	43% (3/7)	MEM	MEM	60%
Mozilla-D	67% (156/233)	46% (107/233)	0% (0/233)	0% (0/233)	0% (0/233)	6% (15/233)	MEM	MEM	20%
Mozilla-C	100% (10/10)	100% (10/10)	10% (1/10)	10% (1/10)	0% (0/10)	90% (9/10)	MEM	MEM	52%
OODT	57% (124/217)	8% (17/217)	35% (77/217)	28% (61/217)	MJE	0% (0/217)	7% (16/217)	7% (16/217)	20%
AVG	56% (412/734)	31% (224/734)	22% (160/734)	22% (165/734)	1% (4/517)	4% (33/734)	11% (39/364)	10% (36/364)	**19.63%**

Note that a 100% value of $c2c_{meas}$ need not mean that the two clusters are identical. As defined, $c2c_{meas}$ only reflects a recovery technique's ability to group the appropriate implementation-level entities on a piecemeal (cluster-by-cluster) basis. The measure does not ensure that the granularity of each recovered cluster matches the ground truth. In the above example, the clusters recovered by ACDC are finer-grained than those in the ground-truth architecture. In that case, even though some of the recovered clusters may yield the maximum $c2c_{meas}$, others will not. For this reason, we also accumulate the $c2c_{meas}$ values for each subject system, to determine which techniques are generally more accurate than others (OBJ1).

Tables III, IV, and V, illustrate the matching for the moderate, strong, and very-strong levels, respectively. Each table shows how many components (i.e., clusters) of a ground-truth architecture are matched by the respective technique at the given strength level, both as a numerical ratio and as a percentage value. As in the previous analysis, the highlighted cells are used to denote the techniques with the highest (dark gray) and second-highest (light gray) values for a particular architecture. The optimal result for any technique would be to have a 100% average match rate across all architectures at the very strong level. Higher match rates at higher strength levels are preferred over higher match rates at lower strength levels. We focus our attention on the bottom-most row in each table, indicating the average values for each technique computed across all eight architectures.

For all three strength levels, ARC and ACDC are again shown as the most accurate techniques overall (OBJ1). ARC provides the highest accuracy at the moderate and very-strong levels (Tables III and V), while ACDC slightly outperforms ARC at the strong level (Table IV). Overall, the recovery techniques perform very poorly at matching components in the strong ("middle") range: the mean value of $c2c_{meas}$ across all eight techniques was only 8.5% (the bottom-right cell in Table IV). By comparison, the techniques more than double the match rate in both the moderate (19.13%) and very strong (19.63%) ranges. This was an unexpected trend. We are in the process of performing additional analysis, and are also contacting the authors of the different recovery techniques used in our study, to help shed light on the reasons behind these results.

The results of the c2c analysis further reinforce the observation made in the case of the MoJoFM analysis: simply put, the state-of-the-art architecture recovery techniques need to improve. On the whole, the techniques performed poorly in obtaining individual clusters that match ground-truth architectures. The average accuracy of each technique is below 33%, with the exception of ARC at the very-strong level, which matched slightly over half (56%) of the ground-truth clusters. Furthermore, as already mentioned above, the mean values across all eight techniques are uniformly under 20% (the bottom-right cells in Tables III, IV, and V).

Finally, similarly to the MoJoFM analysis, we observe higher

accuracy measurements for coarse-grained architectures. The average accuracy across all recovery techniques and all three strength levels is 54% for Linux-C and 59% for Mozilla-C. On the other hand, the average accuracy is well under 20% for all remaining architectures.

D. Recovery Criteria Indicators

As part of meeting OBJ1 and OBJ2, we analyze the accuracy of the recovery criteria implemented by the different techniques. The selected techniques use three types of recovery criteria (recall Section III): (1) establishing a similarity measure, (2) maximizing an objective function, or (3) identifying a particular pattern involving a given system's modules. In support of OBJ2, we determine the extent to which each criterion is reflected in each cluster in the ground-truth architectures. This analysis can suggest possible ways of combining criteria for future recovery techniques. We analyze each criterion across all techniques for a given architecture, and across all architectures for a given technique. Analyzing recovery criteria in this manner also allows us to further assess OBJ1 and determine the extent to which the results of the recovery-criterion analysis are consistent with the results of the MoJoFM analysis. This recovery-criterion analysis is similar to a previous analysis of coupling between Java classes in packages [10].

For each of the three types of recovery criteria, we have defined functions that quantify the extent to which a criterion is indicated by a cluster. We refer to these functions as *criteria indicators*. Each criterion indicator gives a value between 0 and 1. Values of 1 signify that the cluster is completely indicated by the recovery criterion, while a value of 0 signifies that the recovery criterion does not indicate the cluster at all. For simplicity of analysis in this paper, we select the criterion-indicator values (v_{ci}) within two different ranges: *strong* for $0.6 < v_{ci} \leq 1$, and *very strong* for $0.8 < v_{ci} \leq 1$.

Bunch is the only technique among the six selected recovery techniques that uses an objective function as a recovery criterion. Bunch's objective function *MQ* is computed using the "cluster factor" CF_i for each cluster c_i (recall Section III). Therefore, we directly use CF_i as the criterion indicator in the case of Bunch.

For the hierarchical clustering techniques—ARC, LIMBO, WCA, and ZBR—we compute the extent to which the similarity measure that each technique uses is an indicator of the clusters in the ground-truth architectures. To this end, we compute the following criterion indicator, $crit_{hier}$, for each cluster c in a ground-truth architecture:

$$crit_{hier} = \begin{cases} 0 & E_c = 1 \\ \frac{1}{E_c} \sum\limits_{\substack{i,j \\ i \neq j}}^{E_c} sim(c_i, c_j) & E_c > 1 \end{cases}$$

E_c is the number of entities in cluster c. *sim* is the similarity measure of a technique, which is computed for two clusters c_i and c_j. For example, *sim* can be D_{js} for ARC or UE for WCA (recall Section III). Thus, $crit_{hier}$ computes the average pair-wise similarity between entities in a cluster.

ACDC is the lone technique that uses patterns. We focus on ACDC's main pattern, the subgraph dominator pattern. As our analysis will show, this pattern is overwhelmingly responsible for ACDC's accuracy.

To obtain a criterion indicator for the subgraph dominator pattern, we need to compute the extent to which a cluster can be characterized as having a single entity that dominates the cluster's remaining entities. To this end, each cluster of an architecture must be represented as a rooted directed graph, RDG_c, whose entities are nodes and their dependencies are edges. A node n_1 in a graph dominates another node n_2 if every path from the root node to n_2 must pass through n_1.

It is possible that, in an actual component, disjoint subsets of entities in a cluster are dominated by distinct entities. For example, a cluster c may have two disjoint subsets of entities EN_1 and EN_2, where no entity in EN_1 has any dependencies on any entity in EN_2, and vice versa. In that case, no single entity dominates all other entities in the cluster. Thus, to create a criterion for the subgraph dominator pattern, we compute a minimum spanning forest (MSF), which is a set of disjoint minimum spanning trees. A minimum spanning tree (MST) of a connected graph G is a connected subgraph SG of G that connects all p nodes of G with $p-1$ of G's edges.

To understand how an MSF helps us to quantify the subgraph dominator pattern, consider the case of an MST that contains all entities of a cluster c. Then, RDG_c is indicated by the subgraph dominator pattern since all of RDG_c's entities are dominated by its root entity. If a single MST cannot include all entities of c, we can compute an MSF using a variation of Prim's algorithm [1], [39]. Once an MST is computed that does not contain all nodes of RDG_c, one of the remaining nodes is selected and another MST is computed. MSTs are continuously computed in this manner until are nodes are accounted for.

Thus, by computing an MSF for a cluster c, the following criterion indicator can be used for ACDC's subgraph dominator pattern:

$$crit_{sdp} = \frac{numEnt(maxMST(MSF_c))}{E_c}$$

maxMST returns the MST from an MSF that has the largest number of entities. *numEnt* is the number of entities in an MST. As the number of entities in the largest MST of a cluster's MSF increases, so does the value for $crit_{sdp}$. In turn, a higher value for $crit_{sdp}$ means that the cluster is more highly indicative of ACDC's subgraph dominator pattern.

Tables VI and VII depict the criterion-indicator analysis for the strong and very-strong levels, respectively. Each table depicts the average criterion-indicator values across all clusters of an architecture. Dark gray and light gray cells denote the techniques with the highest and second-highest criterion-indicator values for a given architecture.

As with the MoJoFM and c2c analyses, ARC and ACDC showed the best results for the criterion-indicator analysis. However, in this case ACDC yielded markedly better results than ARC. At the strong level, ACDC's cluster criterion is indicated in 56% of the cases across the eight architectures,

TABLE VI: Criterion-Indicator Analysis, Strong Level

System	ARC	ACDC	WCA-UE	WCA-UENM	LIMBO	Bunch	Z-Uni	Z-Tok	AVG
ArchStudio	44% (24/54)	81% (44/54)	2% (1/54)	0% (0/54)	0% (0/54)	0% (0/54)	0% (0/54)	0% (0/54)	16%
Bash	40% (10/25)	16% (4/25)	16% (4/25)	0% (0/25)	0% (0/25)	8% (2/25)	0% (0/25)	4% (1/25)	11%
Hadoop	43% (29/68)	68% (46/68)	3% (2/68)	0% (0/68)	1% (1/68)	1% (1/68)	7% (5/68)	7% (5/68)	16%
Linux-D	30% (36/120)	53% (64/120)	4% (5/120)	0% (0/120)	3% (4/120)	1% (1/120)	MEM	MEM	15%
Linux-C	100% (7/7)	57% (4/7)	0% (0/7)	0% (0/7)	0% (0/7)	0% (0/7)	MEM	MEM	26%
Mozilla-D	61% (141/233)	46% (107/233)	3% (6/233)	0% (0/233)	0% (0/233)	5% (12/233)	MEM	MEM	19%
Mozilla-C	100% (10/10)	100% (10/10)	0% (0/10)	0% (0/10)	0% (0/10)	30% (3/10)	MEM	MEM	38%
OODT	8% (18/217)	61% (133/217)	7% (15/217)	0% (0/217)	0% (1/217)	0% (0/217)	3% (6/217)	2% (4/217)	10%
AVG	37% (275/734)	56% (412/734)	4% (33/734)	0% (0/734)	1% (6/734)	3% (19/734)	3% (11/364)	3% (10/364)	**13.38%**

TABLE VII: Criterion-Indicator Analysis, Very Strong Level

System	ARC	ACDC	WCA-UE	WCA-UENM	LIMBO	Bunch	Z-Uni	Z-Tok	AVG
ArchStudio	11% (6/54)	70% (38/54)	0% (0/54)	0% (0/54)	0% (0/54)	0% (0/54)	0% (0/54)	0% (0/54)	10%
Bash	16% (4/25)	16% (4/25)	0% (0/25)	0% (0/25)	0% (0/25)	8% (2/25)	0% (0/25)	0% (0/25)	5%
Hadoop	6% (4/68)	56% (38/68)	1% (1/68)	0% (0/68)	0% (0/68)	0% (0/68)	4% (3/68)	4% (3/68)	9%
Linux-D	3% (4/120)	33% (40/120)	1% (1/120)	0% (0/120)	0% (0/120)	1% (1/120)	MEM	MEM	6%
Linux-C	14% (1/7)	14% (1/7)	0% (0/7)	0% (0/7)	0% (0/7)	0% (0/7)	MEM	MEM	5%
Mozilla-D	19% (45/233)	25% (58/233)	1% (2/233)	0% (0/233)	0% (0/233)	2% (4/233)	MEM	MEM	8%
Mozilla-C	90% (9/10)	80% (8/10)	0% (0/10)	0% (0/10)	0% (0/10)	10% (1/10)	MEM	MEM	30%
OODT	0% (0/217)	53% (116/217)	3% (6/217)	0% (0/217)	0% (0/217)	0% (0/217)	1% (2/217)	0% (0/217)	7%
AVG	10% (73/734)	41% (303/734)	1% (10/734)	0% (0/734)	0% (0/734)	1% (8/734)	1% (5/364)	1% (3/364)	**6.88%**

as compared to ARC's 37%. The difference is even more pronounced at the very-strong level, where ACDC yields a 41% average across the eight architectures, as compared to ARC's 10%.

It is interesting to note that the MoJoFM and c2c analyses—where ACDC was matched or slightly outperformed by ARC—incorporated all patterns implemented by ACDC. On the other hand, the criterion-indicator analysis—where ACDC is clearly the most accurate technique—reflects only ACDC's subgraph dominator pattern. While further study is required to understand all of their underlying causes, our results indicate that the other patterns ACDC uses in recovering an architecture may, in fact, result in reducing its accuracy.

Finally, the criterion-indicator results show that the criteria employed by techniques other than ACDC and ARC do not indicate clusters in the ground-truth architectures. These remaining techniques achieve strong criteria indicator values for at most 4% of ground-truth clusters, and very-strong criteria indicator values for at most 1% of ground-truth clusters. We consider both of these values to be negligible. Our results clearly show that clustering criteria used in much of the current research on software architecture recovery are, simply put, wrong and that they need to be carefully reconsidered.

V. Discussion

Two techniques—ARC and ACDC—routinely outperformed the remaining four techniques across all eight ground-truth architectures and all three analyses (MoJoFM, c2c, and recovery-criteria). However, on the whole, all of the studied techniques performed poorly. For the MoJoFM analysis, the techniques obtained values between 38% and 59%. For the three strength levels of the c2c analysis, the techniques on average obtained matches for under 20% of ground-truth clusters. Finally, for the recovery-criterion analysis, criterion-indicator values varied from 0% to 56% on average for the two strength levels.

The particularly poor performance of techniques other than ACDC and ARC across all three analyses suggest that their recovery criteria do not reflect the way engineers actually map entities to components. WCA and LIMBO focus on grouping together entities with similar structural dependencies. Bunch attempts to optimize the coupling and cohesion between entities. ZBR attempts to maximize term similarity weighted by zones in a cluster. None of these criteria seem to be appropriate for architecture recovery.

Given the significantly greater accuracy of ACDC and ARC, techniques that employ structural patterns and system concerns as criteria for recovery may be a fruitful starting point for improving existing and developing new recovery techniques. In particular, the subgraph dominator pattern, or similar patterns, may be particularly beneficial. At the same time, the relative accuracy of the subgraph dominator pattern when compared to the combination of patterns used in ACDC (recall Section IV-D) suggests that researchers creating new recovery techniques should be careful when including multiple criteria in a technique: additional criteria are likely to render a technique more complex and may, at the same time, be detrimental to its accuracy.

We also observe that the results of the recovery-criterion analysis were generally consistent with the MoJoFM and c2c analyses. This suggests that, in order to test any newly developed or selected criteria for a recovery technique, it may be beneficial to perform a criteria-indicator analysis first.

Finally, the results of even the two most accurate techniques varied greatly. ARC's average results varied by 40% for the c2c analysis across the three strength levels, by nearly 30% for the two strength levels of the recovery-criterion analysis, and by over 30% for the MoJoFM analysis. Similarly, ACDC's average results varied by 15% for both the c2c and recovery-criterion analyses, and by over 50% for the MoJoFM analysis. Thus, relying upon even the top-performing techniques alone is insufficient to reliably perform an architecture's recovery in general. This unpredictability of existing techniques, in concert with their overall unreliability, suggest that effective architecture

recovery is likely to require extensive manual intervention—the very thing automated techniques have aimed to eliminate.

VI. Threats To Validity

We have collected a very large amount of data, and used it to draw a number of conclusions about the recovery techniques. However, there are certain issues that potentially undermine the validity of our results and our confidence in them. We highlight the three most important issues.

First, we selected a total of eight variants of six different techniques from a much larger body of research. Clearly, including additional techniques would strengthen our results. However, already discussed, other techniques (1) do not have implementations available, (2) are not targeted at software architectures, (3) have been shown to be inferior to the techniques we have selected, or (4) have not been sufficiently evaluated to meet the threshold we deemed reasonable. We have tried to mitigate the risk by obtaining techniques that employ different kinds of input (textual and structural), use different underlying recovery mechanisms (optimization criteria, similarity measures, and pattern matching), and have been previously shown to be accurate. Furthermore, even though we had to re-implement three of the techniques, we regularly consulted their authors in order to ensure the correctness of our implementations. Our confidence in the veracity of our results is also helped by the fact that two techniques–ACDC and ARC—routinely outperformed the other four techniques and that all techniques performed poorly across all studies.

Second, we have evaluated the techniques on architectures drawn from only six software systems. This is a very limited sample and additional systems are needed to further validate our results. However, we were restricted to systems that can be relied on as ground-truths and, to our knowledge, the eight architectures we have collected, recovered, validated, and used in our study are the largest such set readily available. We have mitigated this specific threat to validity in our study by selecting subject systems that vary across domain, size, implementation paradigm, and implementation language. Thus, although they are limited in number, our subject systems are likely to be representative of a broad class of software systems.

Third, it is widely recognized that there is no single "correct" architecture for a given system. Therefore, even though we invested significant effort in recovering and verifying the ground-truth architectures of four of our subject systems (ArchStudio, Bash, Hadoop, and OODT), and other researchers did so for the remaining two systems (Linux and Mozilla), it is possible that different architectures—both at the level of individual clusters and their overall configurations—could have been recovered by another set of researchers and been deemed "correct" by the systems' architects. Clearly, that would have changed our analysis results. However, it is very likely that the alternative set of ground-truth architectures would be highly similar to the architectures we used in our study, and would therefore yield similar results. Even if we allow for some non-trivial discrepancies, most of the recovery techniques did so poorly along all three measures we used that it is very difficult to foresee a scenario where a different set of ground truths would have changed those results substantially.

VII. Conclusions and Future Work

Clustering software entities is an almost uniformly employed method for automated architecture recovery. In order to improve its accuracy, it is important to provide a careful and reliable comparison of the current state-of-the-art software architecture recovery techniques. The work described in this paper is a step in the direction of creating a set of baseline measurements that can be used as a foundation for future research in the area.

To this end, we have presented a comparative analysis of eight different variants of six state-of-the-art, automated architecture recovery techniques. We assessed their accuracy on eight architectures derived from real software systems. In order to conduct the analysis, we complemented the previously available MoJoFM metric with two new metrics for assessing architecture recovery techniques: c2c and criterion analysis. We perform the analysis, both, at the system-wide level and at the level of individual system components.

As part of our on-going work, we are studying additional open-source software systems. This includes extracting the ground-truth architecture of Google's Chromium, which, at 12MSLOC, has presented a challenging task. We are also trying to isolate the key factors needed to recover an accurate architecture. The recovery techniques may have performed poorly in our study because different criteria may be effective at identifying different kinds of system components. To test this hypothesis, we have begun to identify characteristics that may be recognized in different types of components, including (1) components that implement a system's *core* business logic, (2) components that provide *utility* functionality that is intended to be reused across multiple systems, and (3) components that mediate the interactions among the core and utility components (i.e., the system's *connectors*).

Once we have identified these three types of components in our ground-truth architectures, we will analyze whether and to what extent the accuracy of the recovery techniques we used in this study varies across component types. Additionally, identifying the characteristics of different component types has the potential to suggest new, more targeted clustering criteria that will, in turn, result in the composition of more accurate recovery techniques.

Acknowledgements

This work has been supported by the U.S. National Science Foundation under award numbers 1117593, 1218115, and 1321141, and by Infosys Technologies, Ltd. The authors would like to thank Periklis Andritsos, Valerio Maggio, Spiros Mancoridis, Onaiza Maqbool, and Vassilios Tzerpos for their help with using or implementing their tools or techniques. We would also like to acknowledge Anita Singh for her help with the recovery-criterion analysis.

REFERENCES

[1] "MinimumSpanningForest (jung2 2.0.1 API)." [Online]. Available: http://jung.sourceforge.net/site/apidocs/edu/uci/ics/jung/algorithms/shortestpath/MinimumSpanningForest.html

[2] "Mozilla home of the mozilla project mozilla.org." [Online]. Available: http://www.mozilla.org/en-US/

[3] (1993) mkdep. [Online]. Available: http://linuxcommand.org/man_pages/mkdep1.html

[4] (2012) Apache™Hadoop™. [Online]. Available: http://hadoop.apache.org/

[5] (2012) Programmer's Friend - Class Dependency Analyzer. [Online]. Available: http://www.dependency-analyzer.org/

[6] P. Andritsos and V. Tzerpos, "Information-theoretic software clustering," *IEEE TSE*, 2005.

[7] N. Anquetil and T. Lethbridge, "File clustering using naming conventions for legacy systems," in *Conference of the Centre for Advanced Studies on Collaborative Research*, 1997.

[8] ——, "Recovering software architecture from the names of source files," *Journal of Software Maintenance: Research and Practice*, 1999.

[9] ——, "Comparative study of clustering algorithms and abstract representations for software remodularisation," *IEE Proceedings-Software*, 2003.

[10] F. Beck and S. Diehl, "On the congruence of modularity and code coupling," in *ESEC/FSE*, 2011.

[11] L. Belady and C. Evangelisti, "System partitioning and its measure," *Journal of Systems and Software (JSS)*, 1981.

[12] D. Blei, A. Ng, and M. Jordan, "Latent dirichlet allocation," *The Journal of Machine Learning Research*, 2003.

[13] D. P. Bovet and M. Cesati, *Understanding the Linux kernel*. O'Reilly Media, 2008.

[14] A. Brown and G. Wilson, *The Architecture of Open Source Applications*. Lulu. com, 2011, vol. 1.

[15] A. Corazza, S. Di Martino, and G. Scanniello, "A probabilistic based approach towards software system clustering," in *European Conference on Software Maintenance and Reengineering (CSMR)*, 2010.

[16] A. Corazza, S. D. Martino, V. Maggio, and G. Scanniello, "Investigating the use of lexical information for software system clustering," in *European Conference on Software Maintenance and Reengineering (CSMR)*, 2011.

[17] E. Dashofy, H. Asuncion, S. Hendrickson, G. Suryanarayana, J. Georgas, and R. Taylor, "Archstudio 4: An architecture-based meta-modeling environment," in *Companion to ICSE*, 2007.

[18] L. Ding and N. Medvidovic, "Focus: A light-weight, incremental approach to software architecture recovery and evolution," in *Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2001.

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

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

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

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

[23] D. H. Hutchens and V. R. Basili, "System structure analysis: Clustering with data bindings," *IEEE TSE*, 1985.

[24] K. Kobayashi, M. Kamimura, K. Kato, K. Yano, and A. Matsuo, "Feature-gathering dependency-based software clustering using dedication and modularity," in *International Conference on Software Maintenance (ICSM)*, 2012.

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

[26] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner, "Bunch: A clustering tool for the recovery and maintenance of software system structures," in *International Conference on Software Maintenance (ICSM)*, 1999.

[27] O. Maqbool and H. Babri, "The weighted combined algorithm: A linkage algorithm for software clustering," in *European Conference on Software Maintenance and Reengineering (CSMR)*, 2004.

[28] ——, "Hierarchical clustering for software architecture recovery," *IEEE TSE*, 2007.

[29] C. Mattmann, D. J. Crichton, N. Medvidovic, and S. Hughes, "A software architecture-based framework for highly distributed and data-intensive scientific applications," in *ICSE*, 2006.

[30] C. Mattmann, J. Garcia, I. Krka, D. Popescu, and N. Medvidovic, "The anatomy and physiology of the grid revisited," in *Joint Working IEEE/IFIP Conference on Software Architecture & 6th European Conference on Software Architecture (WICSA/ECSA)*, 2009.

[31] A. McCallum, "Mallet: A machine learning for language toolkit," 2002.

[32] G. J. McLachlan and T. Krishnan, *The EM algorithm and extensions*. Wiley-Interscience, 2007, vol. 382.

[33] N. Medvidovic and V. Jakobac, "Using software evolution to focus architectural recovery," *Automated Software Engineering*, 2006.

[34] J. Misra, K. Annervaz, V. Kaulgud, S. Sengupta, and G. Titus, "Software clustering: Unifying syntactic and semantic features," in *Working Conference on Reverse Engineering (WCRE)*, 2012.

[35] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *IEEE TSE*, 2006.

[36] R. Naseem, O. Maqbool, and S. Muhammad, "Improved similarity measures for software clustering," in *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2011.

[37] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, 2011.

[38] K. Praditwong, M. Harman, and X. Yao, "Software module clustering as a multi-objective search problem," *IEEE TSE*, 2011.

[39] R. C. Prim, "Shortest connection networks and some generalizations," *Bell system technical journal*, 1957.

[40] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the International Conference on Language Resources and Evaluation (LREC) 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, http://is.muni.cz/publication/884893/en.

[41] K. Sartipi, "Alborz: a query-based tool for software architecture recovery," in *International Workshop on Program Comprehension (IWPC)*, 2001.

[42] ——, "Software architecture recovery based on pattern matching," *International Conference on Software Maintenance (ICSM)*, 2003.

[43] R. W. Schwanke, "An intelligent tool for re-engineering software modularity," in *ICSE*, 1991.

[44] R. W. Schwanke and S. J. Hanson, "Using neural networks to modularize software," *Machine Learning*, 1994.

[45] M. Shtern and V. Tzerpos, "A framework for the comparison of nested software decompositions," in *Working Conference on Reverse Engineering (WCRE)*. IEEE, 2004.

[46] M. Siff and T. Reps, "Identifying modules via concept analysis," *IEEE TSE*, 1999.

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

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

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

[50] T. A. Wiggerts, "Using clustering algorithms in legacy systems remodularization," in *Working Conference on Reverse Engineering (WCRE)*, 1997.

[51] J. Wu, A. Hassan, and R. Holt, "Comparison of clustering algorithms in the context of software evolution," in *International Conference on Software Maintenance (ICSM)*, 2005.