Comparing Software Architecture Recovery Techniques Using Accurate Dependencies

Thibaud Lutellier*, Devin Chollak*, Joshua Garcia‡,
Lin Tan*, Derek Rayside*, Nenad Medvidović†, and Robert Kroeger§
*Univ. of Waterloo, Canada †Univ. of Southern California, USA ‡George Mason Univ., USA §Google, Canada

***Abstract*—Many techniques have been proposed to automatically recover software architectures from software implementations. A thorough comparison among the recovery techniques is needed to understand their effectiveness and applicability. This study improves on previous studies in two ways.**

First, we study the impact of leveraging more accurate symbol dependencies on the accuracy of architecture recovery techniques. Previous studies have not seriously considered how the quality of the input might affect the quality of the output for architecture recovery techniques.

Second, we study a system (Chromium) that is substantially larger (9.7 million lines of code) than those included in previous studies. Obtaining the ground-truth architecture of Chromium involved two years of collaboration with its developers. As part of this work we developed a new submodule-based technique to recover preliminary versions of ground-truth architectures.

The other systems that we study have been examined previously. In some cases, we have updated the ground-truth architectures to newer versions, and in other cases we have corrected newly discovered inconsistencies.

Our evaluation of nine variants of six state-of-the-art architecture recovery techniques shows that symbol dependencies generally produce architectures with higher accuracies than include dependencies. Despite this improvement, the overall accuracy is low for all recovery techniques. The results suggest that (1) in addition to architecture recovery techniques, the accuracy of dependencies used as their inputs is another factor to consider for high recovery accuracy, and (2) more accurate recovery techniques are needed.

Our results show that some of the studied architecture recovery techniques scale to the 10M lines-of-code range (the size of Chromium), whereas others do not.

I. Introduction

Software architecture is crucial for program comprehension, programmer communication, and software maintenance. Unfortunately, documented software architectures are either nonexistent or outdated for many software projects. While it is important for developers to document software architecture and keep it up-to-date, it is costly and difficult. Even medium sized projects of 70K to 280K source lines of code (SLOC) require on average 100 hours of work by an experienced recoverer to create an accurate ground-truth architecture [1]. In addition, as software grows in size, it is often infeasible for developers to have complete knowledge of the entire system to build an accurate architecture.

Many techniques have been proposed to automatically or semi-automatically recover software architectures from software code bases [2]–[7]. To understand their effectiveness, thorough comparisons of existing architecture recovery techniques are needed. Several studies have been conducted to evaluate different architecture recovery techniques [2], [8], [9]. The latest study [10], compared nine variants of six existing architecture recovery techniques. This study found that, while the accuracy of the recovered architectures varies and some techniques outperform others, their accuracy is relatively low overall.

This previous study used *include dependencies* as inputs to the existing recovery techniques, which are file-level dependencies established when one file declares that it includes another file. In general, the include dependencies are inaccurate. For example, file `foo.c` may declare that it includes `bar.h`, but may not use any functions or variables declared or defined in `bar.h`. Using include dependencies, one would conclude that `foo.c` depends on `bar.h`, while `foo.c` has no actual code dependency on `bar.h`.

In contrast, *symbol dependencies* are more accurate. A symbol can be a function name or a global variable. For example, consider two files `Alpha.c` and `Beta.c`: file `Alpha.c` contains method `A`; and file `Beta.c` contains method `B`. If method `A` invokes method `B`, then method `A` depends on method `B`. Based on this information, we can conclude that file `Alpha.c` depends on file `Beta.c`.

A natural question to ask is, to what extent would the use of symbol dependencies affect the accuracy of architecture recovery techniques? We aim to answer this question empirically.

The second question we study pertains to the scalability of existing architecture recovery techniques. The largest software system used in the published evaluations of architecture recovery techniques comprises 4MSLOC, and it revealed the scalability limit of several recovery techniques [10]. The size of software is increasing, and many software projects are significantly bigger than 4MSLOC. For example, the Chromium open-source browser contains nearly 10MSLOC. We test whether existing architecture recovery techniques can scale to software of such size.

To this end, this paper compares the *same* nine variants of six architecture recovery techniques from the previous study [10] using symbol dependencies on five software projects to answer the following research questions (RQ):

RQ1: Can more accurate dependencies improve the accuracy of existing architecture recovery techniques?

RQ2: Can existing architecture recovery techniques scale to large projects comprising 10MSLOC or more?

This paper makes the following contributions:

- We compared nine variants of six architecture recovery techniques using two different types of dependencies: symbol and include. To the best of our knowledge, this is the first substantial study to compare different types of dependencies for architecture recovery. (One of us previously conducted a limited study on the effect of different polymorphic call graph construction algorithms on architecture recovery [11].)
- We found the types of dependencies and the recovery algorithms have a significant effect on recovery accuracy. In general, symbol dependencies produce software architectures with higher accuracy than include dependencies (**RQ1**). Our results suggest that, *apart from the selection of the "right" architecture recovery techniques, the accuracy of dependencies is another factor to consider for high recovery accuracy*.
- Our results show that the accuracy is low for all studied techniques, corroborating past results [10].
- We recovered the ground-truth architectures of three open source projects—Chromium, Bash and ArchStudio. At 9.7MSLOC, to the best of our knowledge, Chromium is the largest project used for evaluating architecture recovery techniques to date. The ground-truth architecture of Chromium was not available previously. We obtained it through *two years* of regular discussions and meetings with Chromium developers. We also updated the architectures of Bash and ArchStudio that were reported in [1].
- We propose a new submodule-based architecture recovery technique that combines directory layout and build configurations. The proposed technique was effective in assisting in the recovery of ground-truth architectures. Compared to FOCUS [12] which is used in the previous work [1] to recover ground-truth architectures, the submodule-based technique is conceptually simple. Since the technique is used for generating a starting point, its simplicity can be beneficial; any issues potentially introduced by the technique itself can later be mitigated by the manual verification step.
- We found some recovery techniques do, and some do not, scale to the size of Chromium. (**RQ2**).

II. Related Work

A. Comparison of Software Architecture Recovery Techniques

Many architecture recovery techniques have been proposed [2]–[7]. The most recent study [10] collected the ground-truth architectures of eight systems and used them to compare the accuracy of nine variants of six architecture recovery techniques. Two of those recovery techniques—Architecture Recovery using Concerns (ARC) [4] and Algorithm for Comprehension-Driven Clustering (ACDC) [7]—routinely outperformed the others; however, even the accuracy of these techniques showed significant room for improvement.

Architecture recovery techniques have been evaluated against one another in many other studies [2], [5], [8]–[10], [13]. The results of the different studies are not always consistent. scaLable InforMation BOttleneck (LIMBO) [14], a recovery technique leveraging an information loss measure, and ACDC performed similarly in one study [2]; however, in a different study, Weighted Combined Algorithm (WCA) [15], a recovery technique based on hierarchical clustering, outperformed Complete Linkage (CL) [15]. In yet another study, CL is shown to be generally better than ACDC [9]. In the most recent study, ARC and ACDC surpass LIMBO and WCA [10]. Wu et al. [9] compared several recovery techniques utilizing three criteria: stability, authoritativeness, and non-extremity. For this study, no recovery technique was consistently superior to others on multiple measures. A possible explanation for the inconsistent results of these studies is their use of different assessment measures.

The types of dependencies which serve as input to recovery techniques vary among studies: some recovery techniques leverage control and data dependencies [16]–[18]; other techniques use static and dynamic dependency graphs [2]. Previous work [11] examined the effect of different polymorphic call graph construction algorithms on automatic clustering.

None of the papers mentioned above assess the influence of symbol dependencies on recovery techniques when compared to include dependencies. This paper is the first to study (1) the impact of symbol dependencies on the accuracy of recovery techniques and (2) the scalability of recovery techniques to a large project with nearly 10MSLOC.

B. Recovery of Ground-Truth Architectures

Garcia et al. [1] describe a method to recover the ground-truth architectures of four open-source systems. The method involves extensive manual work, and the mean cost of recovering the ground-truth architecture of seven systems ranged from 70KSLOC to 280KSLOC was 107 hours.

Bowman et al. [19] and Xiao et al. [20] recovered the ground-truth architectures of the Linux kernel 2.0 and Mozilla 1.3 respectively. The Linux kernel and Mozilla are large systems, but the evaluated versions are more than a decade old. The version of the Linux kernel recovered was from 1996 and at that time, it contained only 750KSLOC. Mozilla 1.3 is from 2003 with 4MSLOC.

Grosskurth et al. [21] studied the architecture and evolution of web browsers and provide guidance for obtaining a reference architecture for web browsers. Their work does not address the challenges of recovering an accurate ground-truth architecture in general. In addition, it is not clear if their approach is accurate for modern web browsers such as Chromium, which use new design principles such as a modern threading model for tabbed browsing.

Several commercial tools such as Lattix [22] and Structure101 [23] are used to ensure the quality of a given architecture and monitor its evolution. As far as we know, none of those tools claim to generate automatically the ground-truth architecture of a project.

III. Approach

Our approach consists of two parts: the extraction of symbol dependencies and obtaining a ground-truth architecture. In the rest of this section, we describe the manner in which we extract symbol and include dependencies for C/C++ (Section III-A) and Java (Section III-B), discuss why symbol dependencies are more accurate than include dependencies (Section III-C), and elaborate on our approach for obtaining ground-truth architectures (Section III-D).

A. Obtaining Dependencies for C/C++ Projects

To extract symbol dependencies, we use the technique built by our team that scales to software systems comprising millions of lines of code [24]. The technique compiles a project's source files with LLVM into bitcode, analyzes the bitcode to extract the symbol dependencies for all symbols inside the project, and groups dependencies based on the files containing the symbols. At this stage, our extraction process has not considered symbol declarations. As a result, header-file dependencies are often missed because many header files only contain symbol declarations. To ensure we do not miss such dependencies, we augment symbol dependencies by analyzing `#include` statements in the source code.

Include dependencies are transitive dependencies utilized by the Makefile during compilation. As in prior work [10], we use `mkdep` to extract them.

B. Obtaining Dependencies for Java Projects

To extract symbol dependencies, we leverage a tool that operates at the Java bytecode level and extracts high-level information from the bytecode in a human readable format [25]. This allows for method calls and member access (i.e., relationships between symbols) to be recorded without having to analyze the source code. By using this information, it is possible to build a complete graph of the symbol dependencies for the Java projects. This method only accounts for symbols used in the bytecode and does not account for runtime usage which can vary due to reflective access.

We extract include dependencies for Java projects from import statements in Java source code by utilizing a script to determine imports and their associated files. The script used to extract the dependencies detects all the files in a package. Then for every file, it evaluates each import statement and adds the files mentioned in the import as a dependency. When a wildcard import is evaluated, all classes in the referred package are added as dependencies.

C. Relative accuracy of Include and Symbol Dependencies

C/C++ include dependencies tend to miss or over-approximate relationships between files, rendering such dependencies inaccurate. Specifically, include dependencies over-approximate relationships in cases where a header file is included but none of the functions or variables defined in the header file are used (recall Section I).

In addition, include dependencies ignore relationships between non-header files (e.g., .cpp to .cpp files), resulting in a significant number of missed dependencies. For example, consider the case where `A.c` depends on a symbol defined in `B.c` because `A.c` invokes a method defined in `B.c`. Include dependencies will not contain a dependency from `A.c` to `B.c` because `A.c` includes `B.h` but not `B.c`. For example, in Bash, we only identified 4 include dependencies between two non-header files, although there are 1035 actual dependencies between non-header files based on our symbol results. Include dependencies miss many important dependencies since non-header files are the main semantic components of a project.

A recovery technique can treat non-header and header files whose names before their extensions match (e.g., `B.c` and `B.h`) as a single unit to alleviate this problem. However, this remedy does not handle cases where such naming conventions are not followed or when the declarations for types are not in a header file.

Include dependencies generated by `mkdep` use transitive dependencies for header files. Consider an example of four files `A.c`, `A.h`, `B.c`, and `B.h`, where `A.c` includes `A.h` and `A.h` includes `B.h`; `A.c` has an include dependency with `B.h` because including `A.h` implicitly includes everything that `A.h` includes.

For Java projects, include dependencies miss relationships between files because they do not account for intra-package dependencies or fully-qualified name usage. At the same time, include dependencies can represent spurious relationships because some imports are unused and wildcard imports are overly inclusive. Include dependencies are therefore significantly less accurate than symbol dependencies.

D. Obtaining Ground-Truth Architectures

To measure the accuracy of existing software architecture recovery techniques, we need to know the "ground-truth" architecture of a target project. Since it is prohibitively expensive to build architectures manually for large and complex software, such as Chromium, we use a semi-automated approach for ground-truth architecture recovery.

We initially showed the architecture recovered using ACDC to a Chromium developer. He explained that most of the ACDC clusters did not make sense and suggested that we start by considering module organization in order to recover the ground truth.

In response, we have introduced a *simple submodule-based approach* to extract automatically a preliminary ground-truth architecture by combining directory layout and build configurations. Starting from this architecture, we worked with developers of the target project to identify and fix mistakes in order to create a ground-truth architecture.

The submodule-based approach groups closely related modules, and considers which modules are contained within another module. It consists of three steps. First, we determine the module that each file belongs to by analyzing the configuration files of the project.

Second, we determine the submodule relationship between modules. We define a *submodule* as a module that has all of its files contained within the subdirectory of another module.

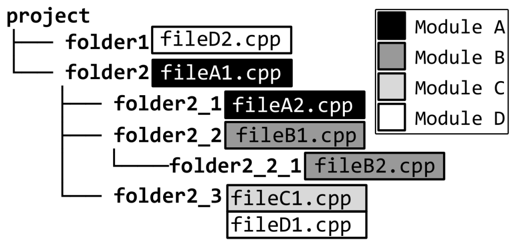

Fig. 1. Example Project Layout

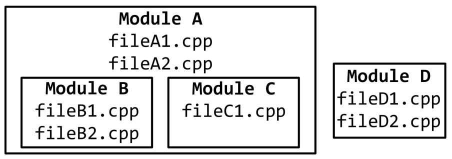

Fig. 2. Example Project Submodules

We first determine a module's *location*, which is defined as the common parent directories that contain at least one file belonging to the module. Then we can determine if a particular module has a relation to another module.

For example, assume a project has four modules named A, B, C, and D. The file structure of the project is shown in Figure 1, while the module structure that we generate is shown in Figure 2.

- **Module A:** contains `fileA1.cpp` and `fileA2.cpp`. Location is `project/folder2`.
- **Module B:** contains `fileB1.cpp` and `fileB2.cpp`. Location is `project/folder2/folder2_2`.
- **Module C:** contains `fileC1.cpp`. Location is `project/folder2/folder2_3`.
- **Module D:** contains `fileD1.cpp` and `fileD2.cpp`. Location is both `project/folder1` and `project/folder2/folder2_3`.

Based on the modules' locations, we determine that module B is a submodule of module A because module B's location `project/folder2/folder2_2` is within module A's location `project/folder2`. Similarly, module C is a submodule of module A. The reason module D has two folder locations is because there is no common parent between the two directories. If module D had a file in the project folder, then its location would simply be `project`. Module D is not a submodule of module A because it has a file located in `project/folder1`.

Last, we group modules that are submodules of one another into a cluster. In the example above, we cluster modules A, B and C into a single cluster and leave module D on its own.

This preliminary version of the ground-truth architecture does not accurately reflect the "real" architecture of the project. Hundreds of hours of manual work are then required to investigate the source code of the system to verify and fix the relationships obtained. When we are satisfied with our ground-truth version, we send it to the developers for certification. Multiples rounds of verifications, based on developers' feedback, are necessary to obtain an accurate ground-truth architecture. For Chromium, it took *two years* of meetings and email exchanges with Chromium developers to obtain the ground truth.

Prior work [1] used a different approach, FOCUS [12], to recover preliminary versions of ground-truth architectures. Compared to FOCUS, the proposed submodule-based technique is conceptually simpler. However, the submodule-based technique uses the same general strategy as FOCUS and can, in fact, be used as one of FOCUS's pluggable elements. This fact, along with the extensive manual verification step, suggests that the strategy used as the starting point for ground-truth recovery does not impact the resulting architecture (as already observed in [1]).

IV. Selected Recovery Techniques

We select the same nine variants of six architecture recovery techniques as in previous work [10] for our evaluation. Four of the selected techniques (ACDC, LIMBO, WCA, and Bunch [6]) use dependencies to determine clusters, while the remaining two techniques (ARC and ZBR [3]) use textual information from source code. We include techniques that do not use dependencies to (1) assess the accuracy of finer-grained, accurate dependencies against these information retrieval-based techniques and to (2) determine their scalability.

Algorithm for Comprehension-Driven Clustering (ACDC) [7] is a clustering technique for architecture recovery. We included ACDC because it performed well in several previous studies [2], [8]–[10]. The main pattern used by ACDC is called the "subgraph dominator pattern". To identify this pattern, ACDC detects a dominator node n_0 and a set of nodes $N = \{n_i \mid i \in \mathbb{N}\}$ that n_0 dominates. A dominator node n_0 dominates another node n_i if any path leading to n_i passes through n_0. Together, n_0, N, and their corresponding dependencies form a subgraph. ACDC groups the nodes of such a subgraph together into a cluster.

Bunch [6], [26] is a technique that transforms the architecture recovery problem into an optimization problem. An optimization function called Modularization Quality (MQ) represents the quality of a recovered architecture. Bunch uses hill-climbing and genetic algorithms to find a partition (i.e., a grouping of software entities into clusters) that maximizes MQ. As in previous work [10], we evaluate two versions of the Bunch hill-climbing algorithms—Nearest and Steepest Ascent Hill Climbing (NAHC and SAHC).

Weighted Combined Algorithm (WCA) [15] is a hierarchical clustering algorithm that measures the inter-cluster distance between software entities and merges them into clusters based on this distance. Two measures are proposed to measure the inter-cluster distance: Unbiased Ellenberg (UE) and Unbiased Ellenberg-NM (UENM). The main difference between these measures is that UENM integrates more information into the measure and thus might obtain better results. In our recent study [10], UE and UENM performed differently depending on the systems tested, therefore, we evaluate both.

LIMBO [14] is a hierarchical clustering algorithm that aims to make the Information Bottleneck algorithm scalable for large data sets. The accuracy of this algorithm was evaluated

TABLE I

EVALUATED PROJECTS AND ARCHITECTURES. †CLUSTER DENOTES THE NUMBER OF CLUSTERS IN THE GROUND-TRUTH ARCHITECTURES.

Project	Version	Description	SLOC	File	Cluster†	Include Dep.	Symbol Dep.
Chromium	svn-171054	Web Browser	9.7M	18,698	67	1,183,799	297,530
ITK	4.5.2	Image Segmentation Toolkit	1M	7,310	11	169,017	30,784
Bash	4.2	Unix Shell	115K	373	14	2,512	2,481
Hadoop	0.19.0	Data Processing	87K	591	67	1,656	3,101
ArchStudio	4	Architecture Development	55K	604	57	866	1,697

in several studies. It performed well in most of the experiments [2], [8], except in our recent study [10] where LIMBO achieved surprisingly poor results.

Architecture Recovery using Concerns (ARC) [4] is a hierarchical clustering algorithm that relies on information retrieval and machine learning to perform a recovery. This technique does not use dependencies and is therefore not used to evaluate the influence of different levels of dependencies. ARC is one of the two best-scoring techniques in our previous evaluation [10] and thus is important to compare against when evaluating for accuracy.

Similar to ARC, **Zone Based Recovery (ZBR)** [3] is a recovery technique based on natural language semantics of identifiers found in the source code. ZBR demonstrated accuracy in recovering Java package structure [3] but struggled with memory issues when dealing with larger systems [10].

V. EXPERIMENTAL METHOD

A. Analyzed Projects

We conduct our comparative study on five open source projects, Bash, ITK, Chromium, ArchStudio and Hadoop. Detailed information on these projects can be found in Table I. For the C/C++ projects, the number of include dependencies is much larger than the number of symbol dependencies, e.g., 297,530 symbol dependencies versus 1,183,799 include dependencies for Chromium. This is the result of both transitive and over-approximation of dependencies, detailed in Section III-C.

Compared to previous work [10], we do not use Linux 2.0.27 and Mozilla 1.3 because our tool extracting symbol-level dependencies for C++ projects works with LLVM. Making those two projects compatible with LLVM would require heavy manual work. Instead, we added ITK to have a medium-sized C/C++ project for evaluation. We also added Chromium, for which we recovered the ground truth. Due to issues around resolving library dependencies with an older version of OODT used in the previous work, we were unable to use it for our study.

For Chromium, the ground-truth architecture was extracted using the submodule approach outlined in Section III-D. ITK was refactored in 2013 and its ground-truth architecture extracted by ITK's developers is available. ITK developers involved in the ITKv4 project confirmed that this architecture was still valid for ITK 4.5.2.

The version of Bash used in the recent study [10] was from 1995. Bash has been changed significantly since then (e.g., from 70KSLOC to 115KSLOC). Therefore, we recovered the ground-truth architecture of the latest version of Bash and used it in our study. Our certifier for Bash is one of Bash's primary developers and its sole maintainer.

The ground-truth architecture for ArchStudio was updated from prior work [1] to be defined at the file-level instead of at the class-level. Additionally, ArchStudio's original ground-truth architecture had a number of inconsistencies and missing files which were verified and corrected by the primary architect.

Hadoop, also an open-source Java project and used in a recent study [10] was the other Java project we evaluated. Its original ground-truth architecture was based on version 0.19.0 and had to be converted from class-level to file-level for our analysis.

B. Architecture Recovery Software and Parameters

To answer the research questions, we compare the clustering results obtained from nine variants of the six architecture recovery techniques, using include and symbol dependencies. We obtained ACDC and Bunch from their authors' websites. For the other techniques, we used our implementation from our previous study [10]. Each of those implementations was shared with the original authors of the recovery techniques and confirmed as correct [10]. Due to the non-determinism of the clustering algorithms used by ACDC and Bunch, we ran each algorithm five times and reported only the best results. WCA, LIMBO, and ARC can take varying numbers of clusters as input. Based on the number of clusters in the ground-truth architectures, we experimented with 5 to 75 clusters as inputs for Bash and ITK, 25 to 75 for Chromium and 30 to 80 for Hadoop and ArchStudio with an increment of 5 for all cases. ARC also takes a varying number of concerns as input. We experimented with 10 to 150 concerns in increments of 10. We report only the best results for each technique.

C. Experimental Environment

For Bash, Hadoop, and ArchStudio, all techniques take a few seconds to a few minutes to run. For large projects such as ITK and Chromium, each technique take several hours or days to run. Running all experiments for Chromium would take more than 20 days of CPU time. This is why we parallelized our experiments by using two machines. We ran ZBR with the two weight variations described in Section IV on a 3.2GHz i7-3930K desktop with 12 logical cores, 6 physical cores, and 48GB of memory. We ran all the other recovery techniques on a 3.3GHz E5-1660 server with 12 logical cores, 6 physical cores, and 32GB memory.

D. Accuracy Measures

There might be multiple ground-truth architectures for a system [1], [19]; that is, experts might disagree. Therefore, a recovered architecture may be different from a ground-truth architecture used in this paper, but close to another ground-truth architecture of the same project. To mitigate this threat, we selected four different metrics to evaluate recovery techniques. One of the metrics—*TurboMQ*—is independent of any ground-truth architecture, which calculates the quality of the recovered architectures. When we use *TurboMQ* to compare different recovery techniques, the threat of multiple ground-truth architectures should not apply. The remaining three metrics—$MoJoFM$, *a2a* and $c2c_{cvg}$—calculate the similarity between a recovered architecture and a ground-truth architecture. If one recovery technique consistently performs well according to all metrics, it is less likely due to the bias of one metric or the particular ground-truth architecture. Although using four metrics cannot eliminate the threat of multiple ground-truth architectures entirely, it should give our results more credibility than using $MoJoFM$ alone.

MoJoFM [27] is defined by the following formula,

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

where $mno(A,B)$ is the minimum number of Move or Join operations needed to transform the recovered architecture A into the ground truth B. This measure allows us to compare the architecture recovered by the different techniques according to their similarity with the ground-truth architecture. A score of 100% indicates that the architecture recovered is the same as the ground-truth architecture. A lower score results in greater disparity between A and B. $MoJoFM$ has been shown to be more accurate than other measures and was used in the latest empirical study of architecture recovery techniques [5], [10].

Architecture-to-architecture [28] (*a2a*) is designed to address some of $MoJoFM$ drawbacks. $MoJoFM$'s Join operation is excessively cheap for clusters containing a high number of elements. This is particularly visible for large projects. This results in high $MoJoFM$ values for architectures with many small clusters. In addition, we discovered that $MoJoFM$ does not properly handle discrepancy of files between the recovered architecture and the ground truth. We tried to reduce this problem by adding the missing files to the recovered architecture into a separate cluster before measuring $MoJoFM$, but this does not entirely solve the issue. In complement of $MoJoFM$, we use a new metric, *a2a*, based on architecture adaptation operations identified in previous work [29], [30]. *a2a* is a distance measure between two architectures:

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

$$\begin{aligned} mto(A_i, A_j) = {} & remC(A_i, A_j) + addC(A_i, A_j) + \\ & remE(A_i, A_j) + addE(A_i, A_j) + movE(A_i, A_j) \\ aco(A_i) = {} & addC(A_\emptyset, A_i) + addE(A_\emptyset, A_i) + movE(A_\emptyset, A_i) \end{aligned}$$

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

mto and *aco* are used to calculate the total numbers of the five operations used to transform one architecture into another: 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.

Cluster-to-cluster coverage ($c2c_{cvg}$) is a metric used in our previous work [31] to assess component-level accuracy. This metric measures the degree of overlap between the implementation-level entities contained in two clusters:

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

where c_i is a technique's cluster; c_j is a ground-truth cluster; and $entities(c)$ is the set of entities in cluster c. 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.

To summarize the extent to which clusters of techniques match ground-truth clusters, we leverage *architecture coverage* ($c2c_{cvg}$). $c2c_{cvg}$ is a change metric from our previous work [31] that indicates the extent to which one architecture's clusters overlap the clusters of another architecture:

$$c2c_{cvg}(A_1, A_2) = \frac{|simC(A_1, A_2)|}{|A_2.C|} \times 100\%$$

$$\begin{aligned} simC(A_1, A_2) = \{c_i \mid \; & (c_i \in A_1, \exists c_j \in A_2) \wedge \\ & (c2c(c_i, c_j) > th_{cvg})\} \end{aligned}$$

A_1 is the recovered architecture; A_2 is a ground-truth architecture; and $A_2.C$ are the clusters of A_2. th_{cvg} is a threshold indicating how high the *c2c* value must be for a technique's cluster and a ground-truth cluster in order to count the latter as covered.

Turbo Modularization Quality (*TurboMQ*) is the final metric we are using in this paper. Modularization metrics measure the quality of the organization and cohesion of clusters based on the dependencies. We implemented the *TurboMQ* version because it has better performance than *BasicMQ* [32].

To compute *TurboMQ* two elements are required: intra-connectivity, and extra-connectivity. The assumption behind this metric is that architectures with high intra-connectivity are preferable to architectures with a lower intra-connectivity. For each cluster, we calculate a Cluster Factor as followed:

$$CF_i = \frac{\mu_i}{\mu_i + 0.5 \times \sum_j \epsilon_{ij} + \epsilon_{ji}}$$

μ_i is the number of intra-relationships;ϵij + ϵ_{ji} is the number of inter-relationships between cluster i and cluster j. *TurboMQ* is defined as the sum of all the Cluster Factors:

$$TurboMQ = \sum_{i=1}^{k} CF_i$$

TABLE II
$MoJoFM$ RESULTS. †SCORES DENOTE RESULTS FOR INTERMEDIATE ARCHITECTURES OBTAINED AFTER THE TECHNIQUE TIMED OUT.

Algorithm	Bash Inc	Bash Sym	ITK Inc	ITK Sym	Chrom. Inc	Chrom. Sym	ArchS. Inc	ArchS. Sym	Hadoop Inc	Hadoop Sym
ACDC	41	**59**	59	56	63	**70**	60	**77**	24	41
B-NAHC	44	47	37	41	28	31	52	59	29	29
B-SAHC	45	58	33	**62**	$13^{\dagger}$	**$70^{\dagger}$**	62	62	32	40
WCA-UE	28	37	31	32	23	23	32	33	14	17
WCA-UENM	28	34	31	32	23	23	32	33	14	17
LIMBO	27	28	31	31	TO	23	26	25	17	15
ARC	40		59		45		62		**49**	
ZBR-tok	35		MEM		MEM		48		29	
ZBR-uni	39		MEM		MEM		48		38	

TABLE III
$a2a$ RESULTS. †SCORES DENOTE RESULTS FOR INTERMEDIATE ARCHITECTURES OBTAINED AFTER THE TECHNIQUE TIMED OUT.

Algorithm	Bash Inc	Bash Sym	ITK Inc	ITK Sym	Chrom. Inc	Chrom. Sym	ArchS. Inc	ArchS. Sym	Hadoop Inc	Hadoop Sym
ACDC	64	81	67	74	71	73	71	**88**	68	**84**
B-NAHC	66	86	71	80	69	73	71	83	68	81
B-SAHC	68	**87**	69	80	$60^{\dagger}$	$71^{\dagger}$	72	85	69	83
WCA-UE	63	81	74	**82**	70	**75**	71	84	68	81
WCA-UENM	63	81	74	**82**	70	**75**	71	84	68	81
LIMBO	63	80	71	80	TO	71	67	79	68	80
ARC	67		60		56		87		**84**	
ZBR-tok	31		MEM		MEM		85		81	
ZBR-uni	32		MEM		MEM		86		83	

TABLE IV
$c2c_{cvg}$ RESULTS FOR MAJORITY(50%), MODERATE(33%) AND WEAK(10%) MATCHES. † SCORES DENOTE RESULTS FOR INTERMEDIATE ARCHITECTURES OBTAINED AFTER THE TECHNIQUE TIMED OUT.

Algorithm	Bash Inc	Bash Sym	ITK Inc	ITK Sym	Chromium Inc	Chromium Sym	ArchStudio Inc	ArchStudio Sym	Hadoop Inc	Hadoop Sym
ACDC	21 50 71	**36 79 92**	0 0 62	0 0 53	16 30 80	**22 48 92**	9 21 47	**56 77 93**	0 3 43	**7** 18 49
B-NAHC	14 36 71	7 28 85	0 0 30	0 0 61	0 0 7	0 0 26	4 9 33	11 19 61	1 3 35	0 12 49
B-SAHC	14 36 64	21 57 **92**	0 0 0	0 **7 92**	$0^{\dagger}$ $6^{\dagger}$ $19^{\dagger}$	$14^{\dagger}$ $33^{\dagger}$ $80^{\dagger}$	7 16 49	11 19 54	1 3 32	4 10 49
WCA-UE	0 14 64	0 21 **92**	0 0 23	0 0 30	0 0 4	0 0 3	0 9 39	7 18 39	0 7 37	1 15 34
WCA-UENM	0 14 64	0 21 **92**	0 0 23	0 0 23	0 0 4	0 0 3	0 9 39	7 18 39	0 7 37	1 15 34
LIMBO	7 14 64	0 21 78	0 0 23	0 0 23	TO	0 0 0	0 0 77	0 0 **93**	0 0 64	0 0 79
ARC	28 57 86		**7 7** 54		3 7 80		21 49 88		6 **36 84**	
ZBR-tok	0 0 21		MEM		MEM		4 16 65		3 15 69	
ZBR-uni	0 0 7		MEM		MEM		4 23 47		3 19 72	

VI. RESULTS

This section presents the results of our study that answer the two research questions, followed by a comparison of our results and those of prior work. Tables II-V show the results for all four metrics when applied to a combination of a recovery technique and system; and, if applicable for such a combination, the results for include or symbol dependencies. For certain combinations of recovery techniques and systems, a result may not be attainable due to techniques running out of memory (MEM) or timing out.

a) RQ1: Can accurate dependencies improve the accuracy of recovery techniques?: Table II and Table III respectively show the $MoJoFM$ and $a2a$ scores to demonstrate the overall accuracy of recovery techniques. Three recovery techniques—ARC, ZBR-tok, and ZBR-uni—do not rely on dependencies; however, we include them to assess the accuracy of symbol dependencies against these information retrieval-based techniques. The best score obtained for each system across all recovery techniques is highlighted in dark gray; the best score for each technique, when applied to a particular system, is highlighted in light gray.

Our results indicate that symbol dependencies generally improve the accuracy of recovery techniques over include dependencies. According to $a2a$ scores, symbol dependencies outperform include dependencies for all of the combinations of techniques and systems which use dependencies. Similar results are observed for $MoJoFM$, despite two exceptions where include dependencies performed between 1 and 3% better. On average, symbol dependencies respectively improve the accuracy by 12% and 7% according to $a2a$ and $MoJoFM$. According to both $MoJoFM$ and $a2a$, the technique getting the largest improvement by the use of symbol dependencies over include dependencies is Bunch-SAHC (21.4% of improvement on average).

Table IV shows $c2c_{cvg}$ for three different values of th_{cvg}, i.e., 50%, 33%, and 10%, (from left to right) for each combination of technique and dependency type. The first value depicts $c2c_{cvg}$ for $th_{cvg} = 50\%$ which we refer to as a *majority* match. We select this threshold to determine the extent to which clusters produced by techniques mostly resemble clusters in the ground truth. The other two $c2c_{cvg}$ scores show the portion of *moderate* matches (33%) and *weak* matches (10%). Dark gray cells show the highest $c2c_{cvg}$ for a system across all recovery techniques, while light gray cells show the highest $c2c_{cvg}$ that each technique obtains for each system for a specific threshold th_{cvg}. Several rows do not have any highlighted cells; such rows indicate that $c2c_{cvg}$ is identical for include and symbol dependencies. We observe significant improvement when using symbol dependencies over include dependencies, even for $th_{cvg} = 50\%$. For example, for ACDC on ArchStudio, the $c2c_{cvg}$ for $th_{cvg} = 50\%$ for include dependencies is 9%, while using symbol dependencies increased it to 56%. Overall, Table IV indicates that (1) the use of symbol dependencies generally produces more accurate clusters (majority matches); and that (2) $c2c_{cvg}$ is low regardless of the types of dependencies used.

Table V presents the $TurboMQ$ results, which measure the organization and cohesion of clusters independent of ground-truth architectures. Symbol dependencies obtain higher $TurboMQ$ scores than include dependencies. In other words, symbol dependencies help recovery techniques produce architectures with better organization and internal component cohesion than include dependencies. $TurboMQ$ results of the

TABLE V
TurboMQ RESULTS. † SCORES DENOTE RESULTS FOR INTERMEDIATE ARCHITECTURES OBTAINED AFTER THE TECHNIQUE TIMED OUT.

Algorithm	Bash		ITK		Chrom.		ArchS.		Hadoop	
	Inc	Sym	Inc	Sym	Inc	Sym	Inc	Sym	Inc	Sym
ACDC	4.77	**16.9**	**503**	422	183	**443**	19.2	**51.6**	13.0	21.8
B-NAHC	3.44	4.69	6.01	27.1	2.51	7.13	7.73	21.9	12.4	10.7
B-SAHC	3.33	7.09	5.20	193	$141^{\dagger}$	$291^{\dagger}$	15.5	22.0	10.9	14.7
W.-UE	0.11	1.64	0.51	1.71	0.06	1.02	0.45	17.8	0.70	7.24
W.-UENM	0.11	1.57	0.51	1.47	0.06	1.02	0.45	17.8	0.70	7.24
LIMBO	1.11	4.78	1.83	2.48	TO	1.10	1.29	25.1	1.02	15.4
ARC	2.71	8.52	0.09	0.33	7.89	10.2	13.8	37.3	5.33	**25.4**
ZBR-tok	0.32	1.24	MEM		MEM		3.54	15.3	3.16	13.0
ZBR-uni	0.64	0.66	MEM		MEM		3.15	14.9	4.03	16.5

summation of individual scores for each cluster in the architecture make it biased toward architectures with an extremely high number of clusters. For example, ACDC for Chromium, with more than 2000 clusters, obtains $TurboMQ$ scores with one to two orders of magnitude larger than the other metrics.

The overall conclusion from applying these four metrics is that symbol dependencies allow recovery techniques to increase their accuracy for all systems in almost every case, independently of the metric chosen. The different metrics sometimes contradict one another. For example, for ITK, according to $TurboMQ$, include dependencies are better than symbol dependencies for ACDC, while it is the opposite according to $a2a$. This difference is likely to be due to the difference of perspective from which the metrics measure an architecture.

Despite the accuracy improvement of using symbol dependencies over include dependencies, $c2c_{cvg}$ results for majority match are low. This indicates that these techniques' clusters are significantly different from clusters in the corresponding ground truth. It suggests that improvement is needed for all the evaluated recovery techniques.

b) RQ2: Can existing architecture recovery techniques scale to large projects comprising 10MSLOC or more?: Overall, ACDC is the most scalable technique. It took only 70-120 minutes to run ACDC on Chromium on our server. The WCA variations and ARC have a similar execution time (8 to 14 hours), with WCA-UENM slightly less scalable than WCA-UE. Bunch-NAHC is the last technique which was able to terminate on Chromium for both kinds of dependencies, taking 20 to 24 hours depending on the kind of dependencies used. LIMBO only terminated for symbol dependencies after running for 4 days on our server.

Bunch-SAHC timed out after 24 hours for both include and symbol dependencies. We report here the intermediate architecture recovered at that time. Bunch-SAHC investigates all the neighboring architectures of a current architecture and selects the architecture that improves MQ the most; Bunch-NAHC selects the first neighboring architecture that improves MQ. Bunch-SAHC's investigation of all neighboring architectures makes it less scalable than Bunch-NAHC.

LIMBO failed to terminate for include dependencies after more than 4 days running on our server. Two operations performed by LIMBO, as part of hierarchical clustering, result in scalability issues: construction of new features when clusters are merged and computation of the measure used to compare entities among clusters. Both of these operations are proportional to the size of clusters being compared or merged, which is not the case for other recovery techniques that use hierarchical clustering (e.g., WCA).

ZBR needs to store data of the size nzV, where n is the number of files being clustered, z is the number of zones, and V is the number of terms. For large software (i.e., ITK and Chromium), with thousands of files and millions of terms, ZBR ran out of memory after using more than 40GB of RAM.

The use of symbol dependencies improves the recovery techniques' scalability over include dependencies for large projects (i.e., ITK and Chromium). The main reason for this phenomenon is that include dependencies generated by `mkdep` are transitive dependencies, while symbol dependencies contain direct dependencies.

c) Comparison with the Prior Work: As previously mentioned, three of our subject systems were also used in our previous study [10]. It is difficult to compare our results with the prior study for several reasons explained in Section V-A. When using the same type of dependencies (Inc) as in our previous study, we observe that the $MoJoFM$ scores drop by 6% on average for all techniques over the scores reported in [10]. It is possible that the newer version of Bash is more complex and its architecture harder to automatically recover. In the cases of Hadoop and ArchStudio, our previous study used a different level of granularity (class level), which makes comparison with current work irrelevant.

VII. Threats to Validity

There is not necessarily a unique, correct architecture for a system [1], [19]. Recovering ground-truth architectures require heavy manual work from experts. Therefore, it is challenging to obtain different certified architectures for the same system. As we are using only one ground-truth architecture, there is a threat that our study is biased toward that specific architecture. To reduce this threat, we use four different metrics, including one independent of the ground-truth architecture. Two of the metrics used in this study were developed by some authors of this paper, which might have caused a bias in this study. However, all four metrics, including metrics developed independently, follow the same trend—symbol dependencies are better than include dependencies—which mitigates some of the potential bias.

The metrics chosen in this paper measure the similarity and quality of an architecture at different levels—the system level (measured by $MoJoFM$ and $a2a$), the component level (measured by $c2c_{cvg}$) and the dependency-cohesion level (measured by $TurboMQ$). In future work, we intend to measure the accuracy of an architecture from an additional perspective, by analyzing whether the architecture contains undesirable patterns or follows good design principles.

We have evaluated recovery techniques on only five systems. To mitigate this threat, we selected systems of different sizes, functionalities, architecture paradigms, and languages.

VIII. Lessons Learned

a) Granularity Affects Scalability: Architecture recovery techniques and their associated metrics sometimes have difficulty scaling to larger software systems. Grouping elements in a coarser manner before performing extraction can improve scalability. For example, developers of large projects such as ITK and Chromium often define large entities of files called modules which can be used as input elements for recovery.

b) Granularity Affects Accuracy: Working at a coarser input level mitigates the scalability issue but creates new accuracy challenges. For example, if elements of the ground truth are files, and the recovered architecture is at the module level, there are two possible approaches to compare them: (1) create the ground truth at the module level, based on ground truth at the file level, which requires significant manual work and domain expertise; or (2) replace each module in the recovered architecture by the files belonging to it. We have conducted experiments using the second approach and found that none of the existing metrics are suitable for comparing the architectures recovered from file-level dependencies and the architectures recovered from module-level dependencies. For example, the $TurboMQ$ values are higher for architectures at the file level because architectures at the file level typically contain more clusters. These metrics are designed without the intent to accommodate different levels of granularity.

c) Architectures With Many Small Clusters Expose Limitations of Metrics: Chromium's ground truth architecture contains 67 clusters, whereas ACDC produces an architecture for Chromium with over 2000 clusters. Despite this intuitive mismatch, ACDC obtains the best scores for Chromium on all metrics except $a2a$, where it scores a close second.

d) File Mismatches Expose a Limitation of MoJoFM: The dependencies are often incomplete. For example, include dependencies generally contains fewer files than the ground-truth architecture. The reasons were explained in Section III-C, including the fact that non-header-file to non-header-file dependencies are missing. Unfortunately, one of the most commonly used metrics, $MoJoFM$, assumes that the two architectures under comparison contain the same elements. Given this limitation, one can create a recovery technique that achieves 100% $MoJoFM$ score easily but completely artificially. The technique would simply create a filename that does not exist in a project, and place it in a single-node architecture. The $MoJoFM$ score between the single-node architecture and the ground truth will be 100%. By contrast, the $a2a$ metric is specifically designed to compare architectures containing different sets of elements.

e) Direct Dependencies versus Transitive Dependencies: The `mkdep` tool that extracts include dependencies produces transitive dependencies, whereas the tool we used for symbol dependencies produces only direct dependencies. We conducted a preliminary investigation with Bash that suggests that this is another dimension of architecture recovery that deserves further study. Our preliminary investigation compared extraction using direct or transitive dependencies for Bash at the symbol level, and found using direct dependencies to be both more accurate and more scalable.

f) Dependencies Matter for Evaluating Architecture Recovery Techniques: This paper explores whether the type of dependencies used affects the quality of the architecture recovered, and answers in the affirmative: each recovery technique gets better if more detailed input dependencies are used. This paper has not focused on the question of which architecture recovery technique is best. The results in this paper show, however, that any attempted evaluation of architecture recovery techniques must be careful about dependencies: $MoJoFM$ would select a different best technique in four out of five cases with different input dependencies; $a2a$ in 3/5 cases; $c2c$ in 2/5 cases; and $TurboMQ$ in 1/5 cases.

g) Which metrics and recovery techniques to use?: Using only one metric is not enough to assess the quality of architectures. However, some metrics are better than others depending on the context. When working on software evolution, the architectures being compared will likely include a different set of files. In this case, $a2a$, $c2c$, and $TurboMQ$ are more appropriate than $MoJoFM$, which assumes that no files are added or removed across versions. If the architectures being compared contain the same files (e.g., comparing different techniques with the same input), $a2a$ will give results with a small range of variations, making it difficult to differentiate the results of each technique. In this case, $MoJoFM$ results are easier to analyze than the ones obtained with $a2a$.

We do not claim that one recovery technique is better than the others. However, we can provide some guidelines to help practitioners choose the right recovery technique for their specific needs. According to our scalability study, ACDC, ARC, WCA, and Bunch-NAHC are the most adapted to recover large software architectures. When trying to recover the low-level architecture of a system, practitioners should favor ACDC, as it generally produces a high number of small clusters. If a different level of abstraction is needed, WCA, LIMBO, and ARC allow the user to choose the number of clusters of the recovered architecture. Those techniques will be more helpful for developers who already have some knowledge of their project architecture.

IX. Conclusions

The paper evaluates the impact of using more accurate symbol dependencies, versus the less accurate include dependencies used in previous studies, on the accuracy of automatic architecture recovery techniques. We studied nine variants of six architecture recovery techniques on five open-source systems. In general, each recovery technique extracted a better quality architecture when using symbol dependencies instead of the less-detailed include dependencies.

In some sense this general conclusion that quality of input affects quality of output is not surprising: the principle has been known since the beginning of computer science. Butler et al. [33] attribute it to Charles Babbage, and note that the acronym 'GIGO' was popularized by George Fuechsel in the 1950's. What is surprising is that this issue has not previously been explored in greater depth in the context of architecture recovery. Our results show that not only does each recovery technique produce better output with better input, but also that the highest scoring technique often changes when the input changes.

There are other dimensions of the input data for architecture recovery that are worthy of future exploration, such as: the granularity of the entities; direct versus transitive dependencies; and the resolution of function pointers and virtual calls. More empirical work is also needed to explore the idea of multiple ground-truth architectures for a given system. One possible direction is to do ground-truth extraction with different groups of engineers on the same system. Another direction would be to have system engineers develop 'ground-truth' architectures starting from automatically recovered architectures. The ground-truth architecture is an important input into this kind of evaluation and deserves greater examination.

The results presented here clearly demonstrate that there is room for more research both on architecture recovery techniques and on metrics for evaluating them.

Acknowledgment

We thank Eric Dashofy for his assistance with updating the ground-truth architecture for ArchStudio 4. We thank Tamara Munzner for the help with graph visualization. This work has been supported by the Natural Sciences and Engineering Research Council of Canada, a Google Faculty Research Award, Ontario Ministry of Research and Innovation, the U.S. National Science Foundation under award numbers 1117593, 1218115, and 1321141, and Infosys Technologies, Ltd.

Availability: The ground-truth architectures are available at http://asset.uwaterloo.ca/ArchRecovery.

References

[1] J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic, "Obtaining Ground-truth Software Architectures," in *Proc. ICSE*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 901–910.

[2] P. Andritsos and V. Tzerpos, "Information-Theoretic Software Clustering," *IEEE Trans. on Softw. Eng.*, vol. 31, no. 2, February 2005.

[3] A. Corazza, S. Di Martino, V. Maggio, and G. Scanniello, "Investigating the use of lexical information for software system clustering," in *Proc. CSMR*. IEEE, 2011, pp. 35–44.

[4] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing Architectural Recovery Using Concerns," in *ASE*, P. Alexander, C. S. Pasareanu, and J. G. Hosking, Eds., 2011, pp. 552–555.

[5] K. Kobayashi, M. Kamimura, K. Kato, K. Yano, and A. Matsuo, "Feature-gathering Dependency-based Software Clustering using Dedication and Modularity," *Proc. ICSM*, vol. 0, pp. 462–471, 2012.

[6] S. Mancoridis, B. S. Mitchell, Y.-F. Chen, and E. R. Gansner, "Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures," in *Proc. ICSM*, 1999.

[7] V. Tzerpos and R. C. Holt, "ACDC : An Algorithm for Comprehension-Driven Clustering," in *Proc. WCRE*. IEEE, 2000, pp. 258–267.

[8] O. Maqbool and H. Babri, "Hierarchical Clustering for Software Architecture Recovery," *IEEE Trans. Softw. Eng.*, vol. 33, no. 11, pp. 759–780, Nov. 2007.

[9] J. Wu, A. E. Hassan, and R. C. Holt, "Comparison of Clustering Algorithms in the Context of Software Evolution," in *Proc. ICSM*, 2005, pp. 525–535.

[10] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," in *Proc. ASE*. IEEE, 2013, pp. 486–496.

[11] D. Rayside, S. Reuss, E. Hedges, and K. Kontogiannis, "The Effect of Call Graph Construction Algorithms for Object-Oriented Programs on Automatic Clustering," in *Proc. IWPC*. IEEE CS Press, 2000, pp. 191–200.

[12] L. Ding and N. Medvidovic, "Focus: A Light-Weight, Incremental Approach to Software Architecture Recovery and Evolution," in *Proc. WICSA*. Washington, DC, USA: IEEE CS Press, 2001, pp. 191–200.

[13] T. Lethbridge and N. Anquetil, "Comparative Study of Clustering Algorithms and Abstract Representations for Software Remodularization," *IEE Proceedings - Software*, vol. 150, no. 3, pp. 185–201, 2003.

[14] P. Andritsos, P. Tsaparas, R. J. Miller, and K. C. Sevcik, "LIMBO: Scalable Clustering of Categorical Data," in *Adv. Database Technol. - EDBT 2004*, 2004, pp. 531–532.

[15] O. Maqbool and H. A. Babri, "The Weighted Combined Algorithm: A Linkage Algorithm for Software Clustering," in *Proc. CSMR*. IEEE CS Press, 2004, pp. 15–24.

[16] R. Fiutem, G. Antoniol, P. Tonella, and E. Merlo, "ART: an Architectural Reverse Engineering Environment," *Journal of Software Maintenance*, vol. 11, no. 5, pp. 339–364, 1999.

[17] R. Fiutem, P. Tonella, G. Antoniol, and E. Merlo, "A Clich-Based Environment to Support Architectural Reverse Engineering," in *Proc. ICSM*. IEEE CS Press, 1996, pp. 319–328.

[18] P. Tonella, R. Fiutem, G. Antoniol, and E. Merlo, "Augmenting Pattern-Based Architectural Recovery with Flow Analysis: Mosaic -A Case Study," in *Proc. WCRE*, 1996, pp. 198–207.

[19] I. T. Bowman, R. C. Holt, and N. V. Brewster, "Linux As a Case Study: Its Extracted Software Architecture," in *Proc. ICSE*. New York, NY, USA: ACM, 1999, pp. 555–563.

[20] C. Xiao and V. Tzerpos, "Software Clustering Based on Dynamic Dependencies," in *Proc. CSMR*, ser. CSMR '05. Washington, DC, USA: IEEE CS Press, 2005, pp. 124–133.

[21] A. Grosskurth and M. W. Godfrey, "A Case Study in Architectural Analysis: The Evolution of the Modern Web Browser," 2007.

[22] F. Waldman, "Lattix LDM," in *8th International Design Structure Matrix Conference, Seattle, Washington, USA, October 24-26*, 2006.

[23] C. Chedgey, P. Hickey, P. O'Reilly, and R. McNamara, *Structure101*. [Online]. Available: https://structure101.com/products/#products=0

[24] P. Wang, J. Yang, L. Tan, R. Kroeger, and D. Morgenthaler, "Generating Precise Dependencies For Large Software," in *Proc. MTD*, May 2013, pp. 47–50.

[25] D. Rayside and K. Kontogiannis, "Extracting Java Library Subsets for Deployment on Embedded Systems," *Sci. Comput. Program.*, vol. 45, no. 2-3, pp. 245–270, Nov. 2002.

[26] S. Mancoridis, B. S. Mitchell, and C. Rorres, "Using Automatic Clustering to Produce High-Level System Organizations of Source Code," in *Proc. IWPC*, 1998, pp. 45–53.

[27] Z. Wen and V. Tzerpos, "An Effectiveness Measure for Software Clustering Algorithms," in *Proc. IWPC*, 2004, pp. 194–203.

[28] D. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic, "An Empirical Study of Architectural Change in Open-Source Software Systems," in *Technical Report USC-CSSE-2014-509, Center for Systems and Software Engineering, University of Southern California*, 2014.

[29] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-based Runtime Software Evolution," in *Proc. ICSE*. Washington, DC, USA: IEEE CS Press, 1998, pp. 177–186.

[30] N. Medvidovic, "ADLs and Dynamic Architecture Changes," in *Proc. ISAW*, ser. ISAW '96. New York, NY, USA: ACM, 1996, pp. 24–27.

[31] J. Garcia, D. Le, D. Link, A. S. Pooyan Behnamghader, E. F. Ortiz, and N. Medvidovic, "An empirical study of architectural change and decay in open-source software systems," USC-CSSE, Tech. Rep., 2014.

[32] B. S. Mitchell, "A Heuristic Approach to Solving the Software Clustering Problem," in *Proc. ICSM*. IEEE CS Press, 2003, pp. 285–288.

[33] J. Butler, W. Lidwell, and K. Holden, *Universal Principles of Design*, 2nd ed. Gloucester, MA: Rockport Publishers, 2010.