Measuring the Impact of Code Dependencies on Software Architecture Recovery Techniques

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

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 accurate symbol dependencies on the accuracy of architecture recovery techniques. In addition, we evaluate other factors of the input dependencies such as the level of granularity and the dynamic-bindings graph construction. Second, we recovered the architecture of a large system, Chromium, that was not available previously. 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 results of our evaluation of nine architecture recovery techniques and their variants suggest that (1) using accurate symbol dependencies has a major influence on recovery quality, and (2) more accurate recovery techniques are needed. Our results show that some of the studied architecture recovery techniques scale to very large systems, whereas others do not.

Index Terms—Software architecture, Empirical software engineering, Maintenance and evolution, Program comprehension

1 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 Unfortunately, documented software architectures are either nonexistent or outdated for many software projects.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 an experienced recoverer to expend an average of 100 hours of work 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]. Such techniques typically leverage code dependencies to determine what implementation-level units (e.g., symbols, files, and modules) form a semantic unit in a software system's architecture. To understand their effectiveness, thorough comparisons of existing architecture recovery techniques are needed. Among the studies conducted to evaluate different architecture recovery techniques [2], [8], [9], the latest study [10], conducted by a subset of this paper's authors, 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 overall accuracy is low.

This previous study used *include dependencies* as inputs to the recovery techniques. These 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 or a variable name. 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, by analyzing a set of real-world systems implemented in Java, C, and C++.

Dependencies can be grouped to different levels of granularity, which can affect the manner in which recovery techniques operate. Generally, dependencies are extracted at the file level. For large projects, dependencies can be grouped to the module level, where a module is a semantic unit defined by system build files. Module dependencies can be used to recover architectures even when finer-grained dependencies do not scale. In this paper, we study the extent to which the granularity of dependencies affects the accuracy of architecture recovery techniques.

Another key factor affecting the accuracy of a recovery technique is whether dependencies utilized as input to a technique are direct or transitive. Transitive dependencies can be obtained from direct dependencies by using a transitive-closure algorithm, and may add relationships between strongly related components, making it easier for recovery techniques to extract such components from the architecture. However, as the number of dependencies increases, the use of transitive dependencies with some recovery techniques may not scale to large projects.

Different symbols can be used (functions, global variables, etc.) to create a symbol dependency graph, but it is unclear which symbols have the most impact on the accuracy of architecture recovery techniques. In this paper, we study the impact of function calls and global variable usage on the quality of architecture recovery techniques. In addition, both C++ and Java offer the possibility of using dynamic-bindings mechanisms. Several techniques exist to build dynamic-bindings graphs [11]–[13] and, despite the existence of two early studies [14], [15] about the impact of call-graph construction algorithms, and the origins of software dependencies on basic architecture recovery, no work has been done to study the effect of dynamic-bindings resolution on recent architecture recovery techniques.

The last question we study pertains to the scalability of existing automatic architecture recovery techniques. While large systems have been studied and their architectures analysed in previous work [16]–[18], the largest software system used in the published evaluations of automatic architecture recovery techniques is Mozilla 1.3, comprising 4MSLOC, and it revealed the scalability limits of several recovery techniques [10]—an old version of Linux was also studied, but its size reported in previous evaluations was only 750KSLOC. The size of software is increasing, and many software projects are significantly larger than 4MSLOC. For example, the Chromium open-source browser contains nearly 10MSLOC. In this paper, we test whether existing automatic 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], as well as two additional baseline algorithms, using eight different types of 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: What is the impact of different input factors, such as the granularity level, the use of transitive dependencies, the use of different symbol dependencies and dynamic-bindings graph construction algorithms, on existing architecture recovery techniques?

RQ3: Can existing automatic 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 eight types of dependencies at different levels of granularity to assess their effects on accuracy. More specifically, we studied the impact of dynamic-bindings resolution algorithms, function calls, and global variable usage on the accuracy of recovery algorithms. We also expand the previous work by studying whether using a higher level of granularity or transitive dependencies improve the accuracy of recovery techniques. This is the first substantial study to the impact of different types of dependencies for architecture recovery.
- We found that 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, other factors to consider for improved recovery accuracy are the dynamic-bindings graph resolution algorithm, the granularity of dependencies, and whether such dependencies are direct or transitive (**RQ2**).
- Our results show that the accuracy is low for all studied techniques, with only one technique (ACDC) consistently producing better results than k-means, a basic machine learning algorithm. This corroborates past results [10] but does so on a different set of subject systems, including one significantly larger system, and for a different set of dependency relationships.
- We recovered the ground-truth architecture of Chromium (svn revision 171054). This ground-truth architecture was not available previously and 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]. All ground-truth architectures have been certified by the developers of the different projects.
- 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 [19], which is used in 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. Working with coarser-grained dependencies and using direct dependencies are two possible solutions to make those techniques scale (**RQ2 and RQ3**).

2 Related Work

2.1 Comparison of Software Architecture Recovery Techniques

This paper builds on work that was previously reported in [20]. Novelty with respect to this previous work includes a study of the impact of dynamic-bindings resolution algorithms, function calls, and global variable usage on the accuracy of recovery algorithms. We also expand the previous work by studying whether using a higher level of granularity and transitive dependencies improves the accuracy of recovery techniques.

Many architecture recovery techniques have been proposed [2]–[7], [21]. 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], [22].

The results of the different studies are not always consistent. scaLable InforMation BOttleneck (LIMBO) [23], 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) [24], a recovery technique based on hierarchical clustering, outperformed Complete Linkage (CL) [24]. 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 [25]–[27]; other techniques use static and dynamic dependency graphs [2].

Previous work [14] examined the effect of different polymorphic call-graph construction algorithms on automatic clustering. Another work [15] studied the impact of source-code versus object-code-based dependencies on software architecture recovery. They found that dependencies obtained directly from source code are more useful than dependencies obtained from object code. While also studying the impact of dependencies on automatic architecture recovery, we focus on the accuracy and the type of the dependencies (e.g., include and symbol dependencies) independently from the way dependencies were extracted, i.e., from object code or source code.

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.

2.2 Recovery of Ground-Truth Architectures

Ground-truth architectures enable the understanding of implemented architectures and the improvement of automated recovery techniques. Several prior studies invested significant time and effort to recover ground-truth architectures for several systems.

2.2.1 Manual Recovery

Garcia et al. [1] described 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. CacOphoNy [16] is another approach that uses metamodels to aid in manual recovery of software architectures and had been used to reverse engineer a large software system [28].

In his work [29], Laine manually recovered the architecture of the X-Window System to illustrate the importance of software architecture for object-oriented development.

Grosskurth et al. [30] 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.

Bowman et al. [31] and Xiao et al. [32] 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.

2.2.2 Tool-Assisted Architecture Recovery

Several tools have been created to help developers analyze, visualize, and reconstruct software architectures. [33] Those tools can be used in different stages of manual architecture recovery.

Rigi [34] is a tool that can be used to analyse and visualize software dependencies. While it is possible to generate an architectural view of a project with the help of Rigi [35], this requires the intervention of a developer with deep knowledge of the project to manually group similar elements (classes, files, etc.) together. Indeed, for large projects, initial views proposed by Rigi are unreadable due to the large number of nodes and dependencies [36] and manual effort is necessary to recover the architecture of the system. The Portable Bookshelf [37], SHriMP [38], AOVIS [39], LSEdit [40] are other software visualization and analysis tools that can help manual architecture recovery.

Several other tools such as Understand [41], Lattix [42] and Structure101 [43] have been used to ensure the quality of a given architecture and monitor its evolution. However, none of these tools intend to automatically recover an architecture.

3 Approach

Our approach is illustrated in Figure 1. First, we extract different types of dependencies for each projects. Then, we provide those dependencies as input to six different architecture recovery techniques. We also evaluate three additional techniques that take the project's source code as input. Finally, we used K-means results and architectures extracted from the directory structure of the project as a baseline. To evaluate the quality of the architecture recovered from different sets of dependencies, we obtain a ground-truth architecture of each project that was certified by each project's developers or main architect. Then, we measure

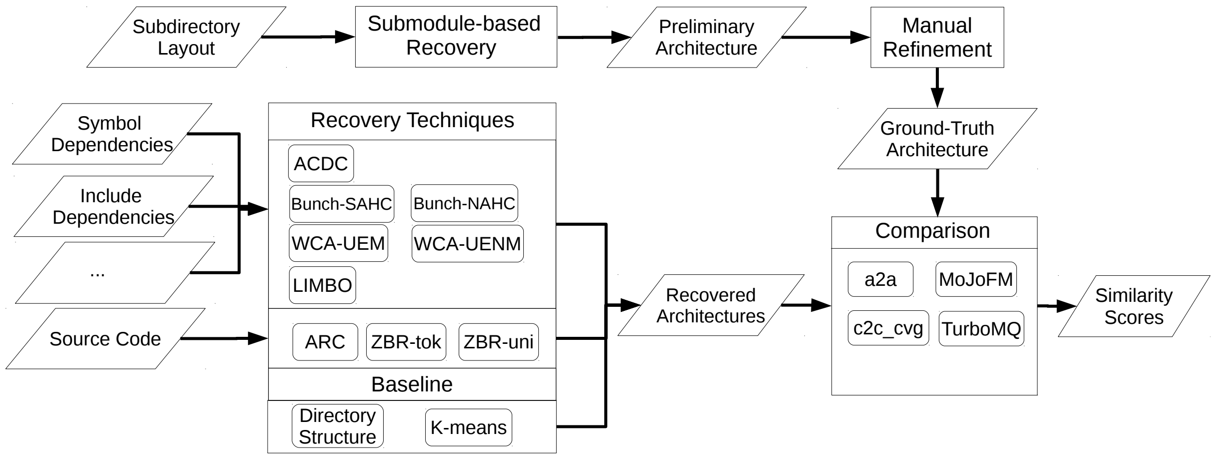

Fig. 1: Overview of our approach

the quality of the architectures recovered automatically by comparing them to the ground-truth architecture using four different metrics.

In the rest of this section, we describe the manner in which we extract the dependencies we study, and elaborate on our approach for obtaining ground-truth architectures.

3.1 Obtaining Dependencies

Both symbol dependencies and include dependencies represent relationships between files, but the means by which these dependencies are determined vary.

3.1.1 C/C++ Projects

To extract symbol dependencies for C/C++, we use the technique built by our team that scales to software systems comprising millions of lines of code [44]. The technique compiles a project's source files into LLVM 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.

These symbol dependencies are direct dependencies, which may be used at the file level or grouped at the module level. For large projects such as Chromium [1] and ITK [2], many developer teams work independently on different parts of the project. To facilitate this work, developers divided these projects into separated sections (modules) that can be updated independently. To group code-level entities at the module level, we extract module information from the build files of the project provided by the developers (e.g. makefile or equivalent). Transitive dependencies are obtained for all projects using the Floyd-Warshall [45] algorithm on symbol dependencies. Because the Floyd-Warshall algorithm did not scale for Chromium, we also tried to use Crocopat [46] to obtain transitive dependencies for Chromium and encountered similar scalability issues.

To extract include dependencies we use the compiler flag `-MM`. Include dependencies are similar to the dependencies used in prior work [10].

1. https://www.chromium.org/developers/how-tos/chromium-modularization
2. https://itk.org/Wiki/ITK/Release_4/Modularization

3.1.2 Java Projects

To extract symbol dependencies for Java, we leverage a tool that operates at the Java bytecode level and extracts high-level information from the bytecode in a structured and human readable format [47]. This allows for method calls and member access (i.e., relationships between symbols) to be recorded without having to analyze the source code itself. Using this information provides a complete picture of all used and unused parts of classes to be identified. We can identify which file any symbol belongs to, since the Java compiler follows a specific naming convention for inner classes and anonymous classes. With information about usage among symbols and resolving the file location for each symbol, we can build a complete graph of the symbol dependencies for the Java projects. This method accounts only for symbols used in the bytecode and does not account for runtime usage which can vary due to reflective access.

We approximate include dependencies for Java by extracting import statements in Java source code by utilizing a script to determine imports and their associated files. To ensure we capture potential build artifacts, the Java projects are compiled before extracting import statements. 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.

The Java projects studied do not contain well-defined modules. In addition, our ground-truth architecture is finer-grained than the package level. For example, Hadoop ground-truth architecture contains 67 clusters when the part of the project we study contains only 52 packages. Therefore, we cannot use Java packages as an equivalent of C++ modules for our module-level evaluation for those specific projects. When studying larger Java projects (e.g. Eclipse), using Java packages could be a good alternative to modules defined in the configuration files used for C++ projects. Maven or Ant build files could also be use as modules for Java projects.

3.1.3 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 1).

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 use transitive dependencies for header files. Consider an example of three files `A.c`, `A.h`, 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.

3.1.4 Overall Accuracy of Symbol Dependencies

To ensure the symbol dependencies we extracted are accurate, we randomly sampled 0.05% of the symbol dependencies and investigated whether these dependencies are correct. This small sample represents 343 dependencies we manually verified. The sampling was done uniformly across projects and dynamic bindings resolutions (interface-only or class hierarchy analysis). We did not find any incorrectly extracted dependencies in this sample, the margin of error being 5.3% with 95% confidence.

We did not quantitatively check whether all the existing dependencies were extracted, as it would be extremely time-consuming to do. However, when building the tool used for extracting dependencies [48], qualitative sanity checks were done to make sure the tool did not miss obvious dependencies.

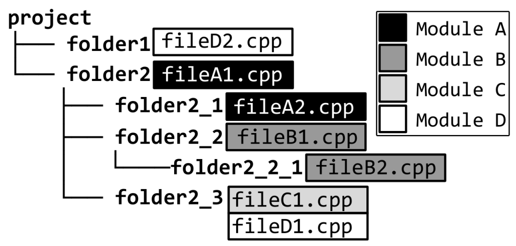

Fig. 2: Example Project Layout

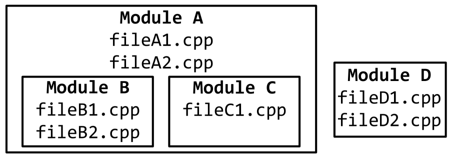

Fig. 3: Example Project Submodules

3.2 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. 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 2, while the module structure that we generate is shown in Figure 3.

- **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`.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2671865, IEEE Transactions on Software Engineering

- **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`.

This preliminary version of the ground-truth architecture does not accurately reflect the "real" architecture of the project and additional manual corrections are necessary. For example, Chromium has two modules `webkit_gpu`, located in the folder `webkit/gpu`, and `content_gpu`, located in the folder `content/gpu`. The two modules are in completely separate folders and are grouped in different clusters by the submodule approach. However, both are involved with displaying GPU-accelerated content and should be grouped together to indicate their close relationship to the `gpu` modules. This is an example where the submodule approach based on folder structure may not accurately reflect the semantic structure of modules and needs to be manually corrected.

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 to the developers the list of clusters containing files and modules in the Rigi Standard Format and a visual representation of how the clusters interact with one another for certification. Multiples rounds of verifications, based on developers' feedback, are necessary to obtain an accurate ground-truth architecture. For the recovery of Chromium, we also had several in-person meetings with a Chromium developer where he explained to us his view of the project's architecture and updated the parts of our preliminary architectures that were inaccurate. During these meetings, the Chromium developer investigated those clusters to see if they make sense (for example, whether the cluster names match with his understanding of Chromium modules and clusters). Then we showed him which files belongs to each cluster using different visualizations (e.g. the "spring" model from Graphviz [49] and a circular view using d3 Javascript library [50]), and he also verified if they were correctly grouped. When he did not agree, we checked if there was some mistakes on our side (i.e. inaccuracy in the submodule technique) or if it was a bug in the Chromium module definition.

It took *two years* of meetings and email exchanges with Chromium developers to obtain the ground truth.

The final ground truth we obtain is a nested architecture. Because most of the architecture recovery techniques produce a flat architecture, we flatten our ground-truth architecture by grouping 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.

Previous work [1], [19] mentioned there might exist different ground-truth architectures for the same project. Despite the fact that our submodule-based approach only recover one ground truth, it is possible to use our approach as a starting point for recovering several ground truths, by having different recoverers and receiving feedback from different developers.

Prior work [1], conducted by a subset of this paper's authors, used a different approach, FOCUS [19], 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]).

4 Selected Recovery Techniques

We select the same nine variants of six architecture recovery techniques as in previous work [10] for our evaluation. We also used 2 baseline clustering algorithms, the K-means algorithm and a directory-based recovery technique. 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.

The view that the techniques we evaluate recover are structural views representing components and their configurations. Such views are fundamental and should be as correct as possible before making other architectural decisions. For example, behavioral or deployment views are still highly dependent on accurate component identification and the configurations among components.

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]. ACDC aims to achieve three goals. First, to help understand the recovered architecture, the clusters produced should have meaningful names. Second, clusters should not contain an excessive number of entities. Third, the grouping is based on identified patterns that are used when a developer describes the components of a software system. 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], [51] 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) [24] is a hierarchical clustering algorithm that measures the inter-cluster distance between software entities and merges them into clusters based on this distance. The algorithm starts with each entity in its own cluster associated with a feature vector. The inter-cluster distance between all clusters is then calculated, and the two most similar clusters are merged. Finally, the feature vector of the new cluster is recalculated. These steps are repeated until WCA reaches the specific number of clusters defined by the user. 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 [23] is a hierarchical clustering algorithm that aims to make the Information Bottleneck algorithm scalable for large data sets. The algorithm works in three phases. Clusters of artefacts are summarized in a Distributational Cluster Feature (DCF) tree. Then, the DCF tree leaves are merged using the Information Bottleneck algorithm to produce a specified number of clusters. Finally, the original artefacts are associated with a cluster. The accuracy of this algorithm was evaluated in several studies. It performed well in most of the experiments [2], [8], except in one 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 considers a program as a set of textual documents and utilizes a statistical language model, Latent Dirichlet Allocation (LDA) [52], to extract concerns from identifiers and comments of the source code. A concern is as a role, concept or purpose of the system studied. The extracted concerns are used to automatically identify clusters and 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 and comments found in the source code. Each file is represented as a textual document and divided into zones. For each word in a zone, ZBR evaluates the term frequency-inverse document frequency (tf-idf) score. Each zone is weighted using the Expectation-Maximization algorithm. ZBR has multiple methods for weighting zones. The initial weights for each zone can be uniform (ZBR-uni), or set to the ratio of the number of tokens in the zone to the number of tokens in the entire system (ZBR-tok). We chose these two weighting variations to ensure consistency with the previous study [10]. The last step of ZBR consists of clustering this representation of files by using group-average agglomerative clustering. ZBR demonstrated accuracy in recovering Java package structure [3] but struggled with memory issues when dealing with larger systems [10].

Previous techniques are clustering algorithms specifically designed for architecture recovery. To obtain an estimate of the quality of the architectures generated by these algorithms, we used two baselines. For the first baseline, we cluster the files using the **K-means** algorithm. Each entity (i.e., a file or module) is represented by a feature vector $\{f_1, f_2, ..., f_n\}$, where n is the number of features. Each of the n features represents a dependency with one of the n entities in the project.

For the second baseline, we used the **directory structure of the project** as an approximation of the architecture of the software. If automatic architecture recovery techniques cannot generate a recovered architecture that is superior to the directory structure of the project, then the recovery technique is not helpful for the specific project. To generate this approximated architecture, we use the same implementation as previous work [53].

5 Experimental Setup

In this section, we describe our experimental environment, how we obtained the ground-truth architectures for each project, the parameters used in our experiments, and the different metrics used to assess the quality of the recovered architectures.

5.1 Projects and Experimental Environment

We conduct our comparative study on five open source projects: Bash, ITK, Chromium, ArchStudio, and Hadoop. Detailed information about these projects can be found in Table 1. We choose those specific version of Bash and Chromium because they were the most recent versions available when we started our ground-truth recovery. For ArchStudio, Hadoop and ITK, we picked those versions because their respective ground-truth architectures were already available.

To run our experiments, we leveraged two machines and parallel processing, due to the large size of some projects. We ran ZBR with the two weight variations described in Section 4 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.

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 takes several hours to days to run. Running all experiments for Chromium would take more than 20 days of CPU time on a single machine. Consequently, we parallelized our experiments.

5.2 Extracted Dependencies

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

TABLE 1: Evaluated projects and architectures. †Cluster denotes the number of clusters in the ground-truth architectures. N/A means the value is not available.

Project	Version	Description	SLOC	File	Cluster†	Inc Dep.	Sym Dep.	Trans Dep.	Mod Dep.
Chromium	svn-171054	Web Browser	10M	18,698	67	1,183,799	297,530	N/A	4,455
ITK	4.5.2	Image Segmentation Toolkit	1M	7,310	11	169,017	30,784	19,281,510	2,700
Bash	4.2	Unix Shell	115K	373	14	2,512	2,481	26,225	N/A
Hadoop	0.19.0	Data Processing	87K	591	67	1,656	3,101	79,631	N/A
ArchStudio	4	Architecture Development	55K	604	57	866	1,697	10,095	N/A

include dependencies for Chromium. This is the result of both transitive and over-approximation of dependencies, detailed in Section 3.1.3.

The number of transitive dependencies shown in Table 1 for ITK is strikingly high. We leverage Class Hierarchy Analysis (CHA) [13] to build the dynamic-bindings dependency graph of symbol dependencies which, in turn, is used for extracting dependencies. When using CHA, we consider that each time a method from a specific class is called, all its subclasses are also called. Depending on how the developers use dynamic bindings, this can generate a large number of dependencies. For example, for ITK, more than 75% of the dependencies extracted are virtual function calls, as opposed to just 11% for Chromium. This high proportion of dynamic bindings also results in an extremely large number of transitive dependencies.

For Chromium, the algorithm to obtain transitive dependencies ran out of memory on our 32GB server. None of the recovery technique scaled for ITK with transitive dependencies. Given that Chromium is around ten times larger than ITK, it is safe to assume that, even if we were able to obtain the transitive dependencies for Chromium, none of the technique would scale to Chromium with transitive dependencies.

5.3 Ground-truth architectures

To assess the effect of different types of dependencies on recovery techniques, we obtained ground-truth architectures for each selected project. Compared to previous work [10], we do not use Linux 2.0.27 and Mozilla 1.3 because our tool that extracts symbol-level dependencies for C++ projects works with LLVM. Making those two projects compatible with LLVM would require heavy manual work. In place of those medium-sized projects, we included ITK. We also included a very large project, Chromium, for which we recovered the ground truth. Due to issues resolving library dependencies with an older version of OODT, for which a ground-truth architecture is available [1], we were unable to use it for our study.

For Chromium, the ground-truth architecture was obtained by manually improving the preliminary architecture extracted using the submodule approach outlined in Section 3.2. After several updates and meetings with a Chromium developer, the ground-truth architecture was certified by one of Chromium's main developers. ITK was refactored in 2013 and its ground-truth architecture, extracted by ITK's developers, is available. We contacted one of ITK's lead developers involved in the refactoring who confirmed that this architecture was still correct for ITK 4.5.2.

The version of Bash used in a recent architecture-recovery 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, who also recently authored a chapter on Bash's idealized architecture [54].

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 ArchStudio's primary architect.

Hadoop, an open-source Java project used in a recent architecture-recovery 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 the class level to the file level for our analysis. For our analysis, we focused on the HDFS, Map-Reduce, and core parts of Hadoop.

5.4 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 different types of dependencies. All input dependencies and output recovered architectures are generated in the Rigi Standard Format [34]. We obtained ACDC and Bunch from their authors' websites. The K-means-based architecture recovery technique was implemented using the scikit-learn python library [55]. For the other techniques, we used our implementation from our previous study [10], [53]. 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 the average results. WCA, LIMBO, ARC and K-means can take varying numbers of clusters as input. We experimented with 20 clusters bellow and above the number of clusters in the ground truth, with an increment of 5 for all cases. For example, for ArchStudio, we ran these algorithms for 40 to 80 clusters. ARC also takes a varying number of concerns as input. We experimented with 10 to 150 concerns in increments of 10. We report the average results for each technique.

5.5 Accuracy Measures

There might be multiple ground-truth architectures for a system [1], [31]; 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 commonly used in other automatic architecture recovery evaluations to measure the impact of the different inputs on the quality of the recovered architectures.

One of the metrics—normalized TurboMQ—is independent of any ground-truth architecture, which calculates the quality of the recovered architectures. When we use normalized 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.

We used MoJoFM, a2a and $c2c_{cvg}$'s implementations provided by the developers of each technique. For TurboMQ, we used our own implementation based on the technique described by the original authors [56].

MoJoFM [57] 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 [58] (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. This observation corroborates results obtained in recent work [58]. 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 [59], [60]. a2a is a distance measure between two architectures:

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

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

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

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

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. $mto(A_i, A_j)$ is calculated optimally by using the Hungarian algorithm [61] to maximise the weights of a bipartite graph built with the clusters from A_i and A_j.

Cluster-to-cluster coverage ($c2c_{cvg}$) is a metric used in our previous work [62] 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 [62] that indicates the extent to which one architecture's clusters overlap the clusters of another architecture:

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

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

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.

Normalized Turbo Modularization Quality (normalized 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. They are widely accepted metrics which have been used in several studies [63]–[65]. We implemented the TurboMQ version because it has better performance than *BasicMQ* [56].

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:

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

We note that TurboMQ by itself is biased toward architectures with a large number of clusters because the sum of CF_i will be very high if the recovered architecture contains numerous clusters. Indeed, we found that for Chromium, the architecture recovered by ACDC contains thousands of clusters. The TurboMQ value for this architecture was 400 times higher than the TurboMQ values of architectures obtained with other recovery techniques. To address this issue, we normalized TurboMQ by the number of clusters in the recovered architecture.

6 Results

This section presents the results of our study that answer the three research questions, followed by a comparison of our results and those of prior work. Tables 2-21 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 a type of dependency: *Inc*lude, *Sym*bol, *Funct*ion alone, function and global variables (*F-GV*), *Trans*itive, and *Mod*ule-level dependencies. Symbol dependencies may be resolved by ignoring dynamic bindings (No Vir) or using a class hierarchy analysis of dynamic bindings (*S-CHA*) or with interface-only resolution of dynamic bindings (*S-Int*). Bash does not contain dynamic bindings because it is implemented in C, and our tool cannot extract function pointers.

For certain combinations of recovery techniques and systems, a result may not be attainable due to inapplicable combinations (NA), techniques running out of memory (MEM), or timing out (TO). For example, information retrieval-based techniques such as ARC and ZBR do not rely on dependencies. Therefore, normalized TurboMQ results are not meaningful when studying the impact of the different factors of the dependencies. For this reason, we only report normalized TurboMQ for include and symbol dependencies and mark the other combinations as inapplicable.

We do not report results obtained utilizing transitive dependencies for Chromium and ITK because, as discussed above, the use of such dependencies with those projects caused scalability problems. Module-level dependencies are only reported for ITK and Chromium, since they are the only projects that define modules in their documentation or configuration files.

6.1 RQ1: Can accurate dependencies improve the accuracy of recovery techniques?

As explained in section 3.1.3, include dependencies present some issues (e.g. missing relationships between non-header files, etc.) which can be solved by using more accurate dependencies based on symbol interactions. Therefore, to answer this research question, we focus on results obtained using include (Inc) and symbol dependencies (Sym, S-Int, and S-CHA), which are presented in Tables 2-21. In these Tables, we reported the average results for each technique and each type of dependency.

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 recovery technique across all type of dependencies is highlighted in dark gray; the best score between include and symbol dependencies for each technique, when applied to a particular technique, 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 (Tables 7-11) relying on both types of symbol dependencies outperforms relying on include dependencies for all of the combinations of techniques and systems which use dependencies. The only exception is in the case of ITK, where relying on include dependencies outperforms interface-only resolution for dynamic bindings. As ITK contains a large number of dynamic-bindings dependencies (more than 75%), using interface-only resolution likely results in a significant loss of information, making those dependencies inaccurate. When doing a complete analysis of the dynamic-bindings dependencies of ITK (S-CHA), using symbol dependencies with a class hierarchy analysis of dynamic bindings outperforms using include dependencies for all techniques. On average, using symbol dependencies respectively improves the accuracy by 9 percentage points (percentage point, pp, is the unit for the arithmetic difference between two percentages) according to a2a. For a2a, the technique obtaining the greatest improvement from the use of symbol dependencies, as compared to include dependencies, is K-means, followed by Bunch-SAHC, with an average improvement of, respectively, 12 pp and a 10 pp for a2a.

MoJoFM results (Tables 2 to 16) followed a similar trend, with symbol dependencies generally improving the accuracy of the recovered architecture over include dependencies for five of the projects. However, for Bash, include dependencies produce better results than symbol dependencies for all techniques but ACDC.

Tables 17-21 show $c2c_{cvg}$ 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 each recovery technique across all type of dependencies. Light gray cells show the highest $c2c_{cvg}$ between include and symbol dependencies for each technique, when applied to a particular technique 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, in Table 20, for ACDC on ArchStudio, the $c2c_{cvg}$ for $th_{cvg} = 50\%$ for include dependencies is 9%, while using symbol dependencies increased it to 56% with symbol dependencies and interface-only resolution. Overall, Tables 17 to 21 indicate 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.

	Bash				
Algo.	Inc.	Sym.	Trans.	Funct.	F-GV
ACDC	52	57	38	49	50
Bunch-NAHC	53	43	34	49	46
Bunch-SAHC	57	52	34	43	49
WCA-UE	34	24	24	29	30
WCA-UENM	34	24	24	31	30
LIMBO	34	27	27	22	22
K-means	59	55	49	47	46
ARC	43				
ZBR-tok	41				
ZBR-uni	29				
Dir. Struc.	57				

TABLE 2: MoJoFM results for Bash.

	Chromium						
Algo.	Inc.	S-CHA	S-Int	No DyB	Mod.	Funct.	F-GV
ACDC	64	70	73	71	62	71	71
Bunch-NAHC	28	31	24	29	52	29	35
Bunch-SAHC	12†	71†	43†	42	57	39	29
WCA-UE	23	23	23	27	76	29	29
WCA-UENM	23	23	23	27	73	29	29
LIMBO	TO	23	3	26	79	27	27
K-means	40	42	43	43	78	45	45
ARC	54						
ZBR-tok	MEM						
ZBR-uni	MEM						
Dir. Struc.	69						

TABLE 3: MoJoFM results for Chromium. † Scores denote results for intermediate architectures obtained after the technique timed out.

	ITK						
Algo.	Inc.	S-CHA	S-Int	No DyB	Mod.	Funct.	F-GV
ACDC	52	55	52	48	35	60	60
B.-NAHC	37	36	35	35	46	45	47
B.-SAHC	32	46	43	41	51	54	53
WCA-UE	30	31	44	45	64	36	36
WCA-UENM	30	31	44	45	61	36	36
LIMBO	30	31	44	38	60	36	35
K-means	38	42	39	43	68	60	61
ARC	24						
ZBR-tok	MEM						
ZBR-uni	MEM						
Dir. Struc.	59						

TABLE 4: MoJoFM results for ITK.

	ArchStudio						
Algo.	Inc.	S-CHA	S-Int	No DyB	Trans.	Funct.	F-GV
ACDC	60	60	77	78	71	75	74
Bunch-NAHC	48	40	49	47	40	53	46
Bunch-SAHC	54	39	53	40	38	53	54
WCA-UE	30	30	32	45	32	31	31
WCA-UENM	30	30	32	45	33	31	31
LIMBO	23	23	24	25	24	24	23
K-means	44	37	39	41	43	39	38
ARC	56						
ZBR-tok	48						
ZBR-uni	48						
Dir. Struc.	88						

TABLE 5: MoJoFM results for ArchStudio.

	Hadoop						
Algo.	Inc.	S-CHA	S-Int	No DyB	Trans.	Funct.	F-GV
ACDC	24	29	41	41	28	41	41
Bunch-NAHC	23	21	24	24	17	26	26
Bunch-SAHC	24	26	28	26	20	29	28
WCA-UE	13	12	15	28	17	17	17
WCA-UENM	13	12	15	28	17	17	17
LIMBO	15	13	14	14	13	13	14
K-means	30	25	29	28	29	29	29
ARC	35						
ZBR-tok	29						
ZBR-uni	38						
Dir. Struc.	63						

TABLE 6: MoJoFM results for Hadoop.

	Bash				
Algo.	Inc.	Sym.	Trans.	Funct.	F-GV
ACDC	65	80	80	41	41
Bunch-NAHC	68	84	83	41	41
Bunch-SAHC	69	84	83	40	41
WCA-UE	65	81	81	40	40
WCA-UENM	65	81	81	39	40
LIMBO	63	79	79	38	37
K-means	67	84	84	41	40
ARC	67				
ZBR-tok	71				
ZBR-uni	70				
Dir. Struc.	64				

TABLE 7: a2a results for Bash.

	ITK						
Algo.	Inc.	S-CHA	S-Int	No DyB	Mod.	Funct.	F-GV
ACDC	67	74	63	58	84	48	48
Bunch-NAHC	71	78	68	58	85	47	47
Bunch-SAHC	69	78	66	57	85	48	47
WCA-UE	74	82	47	39	89	48	48
WCA-UENM	74	82	47	39	88	48	48
LIMBO	70	78	44	36	87	46	46
K-means	74	82	71	61	89	51	51
ARC	54						
ZBR-tok	MEM						
ZBR-uni	MEM						
Dir. Struc.	61						

TABLE 8: a2a results for ITK.

	Chromium						
Algo.	Inc.	S-CHA	S-Int	No DyB	Mod.	Funct.	F-GV
ACDC	71	73	74	64	82	62	62
Bunch-NAHC	69	73	76	66	81	63	63
Bunch-SAHC	60†	71†	66†	66	83	64	62
WCA-UE	70	75	78	68	84	66	66
WCA-UENM	70	75	78	68	82	66	66
LIMBO	TO	70	73	64	83	61	61
K-means	71	74	77	67	86	65	65
ARC	54						
ZBR-tok	MEM						
ZBR-uni	MEM						
Dir. Struc.	60						

TABLE 9: a2a results for Chromium. † Scores denote results for intermediate architectures obtained after the technique timed out.

	ArchStudio						
Algo.	Inc.	S-CHA	S-Int	No DyB	Trans.	Funct.	F-GV
ACDC	71	86	88	83	92	87	88
Bunch-NAHC	69	80	81	75	80	81	81
Bunch-SAHC	70	80	82	74	80	81	82
WCA-UE	70	83	84	81	83	82	83
WCA-UENM	70	83	84	81	84	82	83
LIMBO	67	79	79	74	78	77	78
K-means	70	81	82	77	83	81	82
ARC	84						
ZBR-tok	85						
ZBR-uni	86						
Dir. Struc.	87						

TABLE 10: a2a results for ArchStudio.

	Hadoop						
Algo.	Inc.	S-CHA	S-Int	No DyB	Trans.	Funct.	F-GV
ACDC	68	81	84	79	80	84	84
Bunch-NAHC	67	79	80	76	78	80	80
Bunch-SAHC	67	80	81	76	79	81	81
WCA-UE	68	80	80	78	81	81	81
WCA-UENM	68	80	80	78	81	81	81
LIMBO	67	79	79	75	79	78	79
K-means	70	81	81	77	82	82	82
ARC	82						
ZBR-tok	81						
ZBR-uni	83						
Dir. Struc.	88						

TABLE 11: a2a results for Hadoop.

Algo.	Bash				
	Inc.	Sym.	Trans.	Funct.	F-GV
ACDC	9	22	6	29	29
Bunch-NAHC	25	31	20	33	28
Bunch-SAHC	30	30	20	28	28
WCA-UE	0	7	7	10	10
WCA-UENM	0	7	7	5	10
LIMBO	6	13	8	7	7
K-means	0	17	6	14	16
ARC	5	11	NA		
ZBR-tok	2	8	NA		
ZBR-uni	2	5	NA		
Dir. Struc.	1	4	NA		

TABLE 12: Normalized TurboMQ results for Bash.

Algo.	Chromium						
	Inc.	S-CHA	S-Int	No DyB	Mod.	Funct.	F-GV
ACDC	15	19	18	20	46	24	24
Bunch-NAHC	4	24	9	26	51	16	19
Bunch-SAHC	2†	30†	11†	23	45	29	11
WCA-UE	0	2	2	2	36	2	2
WCA-UENM	0	2	2	2	37	2	3
LIMBO	TO	2	2	2	34	2	2
K-means	0	17	13	19	35	22	22
ARC	2	5	NA				
ZBR-tok	MEM						
ZBR-uni	MEM						
Dir. Struc.	2	5	NA				

TABLE 13: Normalized TurboMQ results for Chromium. † Scores denote results for intermediate architectures obtained after the technique timed out.

Algo.	ITK						
	Inc.	S-CHA	S-Int	No DyB	Mod.	Funct.	F-GV
ACDC	33	24	18	32	47	40	40
Bunch-NAHC	15	23	23	22	50	34	37
Bunch-SAHC	10	29	23	21	50	44	37
WCA-UE	3	9	3	2	51	11	9
WCA-UENM	3	9	3	2	47	10	9
LIMBO	7	11	5	1	35	9	9
K-Means	13	24	15	13	37	31	25
ARC	9	33	NA				
ZBR-tok	MEM						
ZBR-uni	MEM						
Dir. Struc.	9	9	NA				

TABLE 14: Normalized TurboMQ results for ITK.

Algo.	ArchStudio						
	Inc.	S-CHA	S-Int	No DyB	Trans.	Funct.	F-GV
ACDC	66	41	76	84	71	72	74
Bunch-NAHC	72	42	74	85	35	74	75
Bunch-SAHC	71	41	76	85	50	72	74
WCA-UE	1	11	22	65	15	10	19
WCA-UENM	1	11	22	65	15	10	19
LIMBO	2	12	31	38	7	24	27
K-means	13	21	38	51	29	35	39
ARC	18	25	NA				
ZBR-tok	6	17	NA				
ZBR-uni	5	15	NA				
Dir. Struc.	1	26	NA				

TABLE 15: Normalized TurboMQ results for ArchStudio.

Algo.	Hadoop						
	Inc.	S-CHA	S-Int	No DyB	Trans.	Funct.	F-GV
ACDC	48	28	59	65	29	57	58
Bunch-NAHC	40	26	53	61	17	52	48
Bunch-SAHC	40	31	53	61	18	54	56
WCA-UE	1	5	8	34	7	6	8
WCA-UENM	1	5	8	33	7	6	8
LIMBO	2	7	19	25	2	17	17
K-means	11	13	29	34	9	26	27
ARC	6	13	NA				
ZBR-tok	5	10	NA				
ZBR-uni	7	13	NA				
Dir. Struc.	1	20	NA				

TABLE 16: Normalized TurboMQ results for Hadoop.

Tables 12 to 16 presents the normalized TurboMQ results, which measure the organization and cohesion of clusters independent of ground-truth architectures. Both types of symbol dependencies generally obtain higher normalized TurboMQ scores than include dependencies, ACDC and Bunch with CHA resolution being exceptions for ArchStudio and Hadoop. In other words, symbol dependencies help recovery techniques produce architectures with better organization and internal component cohesion than include dependencies. TurboMQ results of the 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.

6.1.1 Statistical Significance Test

We conduct statistical significance tests to verify whether using accurate dependencies improves the quality of recovery techniques. We do not use paired t-tests because our data does not follow a normal distribution. Instead, we use the Wilcoxon signed-rank test, which is a non-parametric test.

We also measure the Cliff's δ, a non-parametric effect size metric, to quantify the difference among the different types of dependencies. Cliff's δ ranges from -1 to 1. The sign of the δ indicates whether the first or second sample contains higher values. For example, when looking at results using include versus results obtained using symbol dependencies with CHA resolution for MoJoFM in Table 23, the δ is positive, indicating that results obtained with symbol dependencies and CHA resolution are generally better than the ones obtained using include dependencies results. In contrast, the δ for direct versus transitive dependencies is negative, indicating that using direct dependencies generally produces higher results. To interpret the effect size, we use the following magnitude: negligible ($|\delta| < 0.147$), small ($|\delta| < 0.33$), medium ($|\delta| < 0.474$), and large ($0.474 \leq |\delta|$) [66]. To compute these tests, we use average measurements obtained for each type of dependencies and metrics

Algo.	Bash														
	Inc.			Sym.			Trans.			Funct.			F-GV		
ACDC	20	47	71	27	77	93	13	40	94	0	7	50	0	7	21
Bunch-NAHC	6	20	53	6	20	71	1	10	46	1	10	36	1	9	37
Bunch-SAHC	7	23	43	16	37	81	3	13	46	0	6	41	0	11	37
WCA-UE	0	6	63	0	4	61	0	1	58	0	0	26	0	1	23
WCA-UENM	0	6	63	0	4	61	0	1	58	0	2	24	0	6	45
LIMBO	0	0	57	0	0	60	0	0	63	0	0	32	0	0	32
K-means	10	19	49	9	28	69	9	26	57	0	5	41	0	2	46
ARC		4						20					54		
ZBR-tok		7						7					71		
ZBR-uni		0						0					50		
Dir. Struct.		14						36					64		

TABLE 17: $c2c_{cvg}$ results for majority(50%), moderate(33%) and weak(10%) matches for Bash.

Algo.	ITK																				
	Inc.			S-CHA			S-Int			No DyB			Mod.			Funct.			F-GV		
ACDC	0	0	57	0	0	48	0	0	31	0	0	31	8	8	54	0	8	38	0	8	38
B.-NAHC	0	0	31	0	0	38	0	0	34	0	0	34	5	23	60	0	0	32	0	0	31
B.-SAHC	0	0	0	0	2	54	0	0	29	0	0	23	6	29	57	0	0	34	0	0	32
WCA-UE	0	0	23	0	0	24	0	8	23	0	0	8	18	40	78	0	0	38	0	0	38
WCA-UENM	0	0	23	0	0	23	0	8	23	0	0	8	18	32	80	0	0	38	0	0	38
LIMBO	0	0	8	0	0	12	0	0	5	0	0	0	26	40	86	0	0	9	0	0	9
K-means	0	0	44	0	8	51	0	3	53	0	9	54	29	46	87	0	6	45	0	8	53
ARC				0							0							23			
ZBR-tok											MEM										
ZBR-uni											MEM										
Dir. Struct.				0							0							8			

TABLE 18: $c2c_{cvg}$ results for majority(50%), moderate(33%) and weak(10%) matches for ITK.

Algo.	Chromium																				
	Inc.			S-CHA			S-Int			No DyB			Mod.			Funct.			F-GV		
ACDC	16	30	80	17	45	92	17	37	87	10	23	83	13	17	28	9	17	80	10	17	80
B.-NAHC	0	0	7	0	0	26	0	0	3	0	0	9	7	14	33	0	0	4	0	3	25
B.-SAHC	0	6	19	14	33	80	7	12	36	4	10	54	8	16	39	0	1	38	0	0	4
WCA-UE	0	0	3	0	0	3	0	0	3	0	0	3	17	37	79	0	0	3	0	0	4
WCA-NM	0	0	3	0	0	3	0	0	3	0	0	3	12	35	76	0	0	3	0	0	4
LIMBO		TO		0	0	0	0	0	0	0	0	0	25	45	69	0	0	0	0	0	0
K-means	3	7	31	1	6	25	2	8	24	1	5	20	24	43	78	1	5	21	1	4	20
ARC				1							3							25			
ZBR-tok											MEM										
ZBR-uni											MEM										
Dir. Struct.				4							7							23			

TABLE 19: $c2c_{cvg}$ results for majority(50%), moderate(33%) and weak(10%) matches for Chromium. † Scores denote results for intermediate architectures obtained after the technique timed out.

Algo.	ArchStudio																				
	Inc.			S-CHA			S-Int			No DyB			Trans.			Funct.			F-GV		
ACDC	9	21	47	21	54	77	56	77	93	52	75	89	52	72	85	42	64	77	42	62	77
B.-NAHC	2	5	21	2	6	28	3	11	41	3	10	34	2	5	26	5	13	44	3	9	41
B.-SAHC	3	10	35	2	7	33	5	13	46	5	8	20	4	7	27	3	11	48	5	19	46
WCA-UE	0	5	37	0	5	29	2	14	38	19	35	53	2	13	39	0	7	37	0	8	47
WCA-NM	0	5	37	0	5	29	2	14	38	19	35	53	2	13	39	0	7	37	0	8	47
LIMBO	0	0	69	0	0	65	0	0	68	0	0	74	0	0	63	0	0	66	0	0	68
K-means	6	25	58	1	18	57	5	24	53	8	23	53	8	33	69	4	22	53	5	24	57
ARC				9							29							59			
ZBR-tok				4							16							65			
ZBR-uni				4							23							47			
Dir. Struct.				74							91							100			

TABLE 20: $c2c_{cvg}$ results for majority(50%), moderate(33%) and weak(10%) matches for ArchStudio.

Algo.	Hadoop																				
	Inc.			S-CHA			S-Int			No DyB			Trans.			Funct.			F-GV		
ACDC	0	3	43	4	13	39	7	18	49	7	18	45	4	10	36	7	16	52	9	16	52
B.-NAHC	1	3	35	1	1	28	1	4	36	1	4	33	1	3	24	2	5	35	2	5	38
B.-SAHC	1	3	32	1	7	40	2	6	35	2	6	34	1	3	20	2	8	38	1	5	36
WCA-UE	0	7	37	0	12	29	1	12	33	2	9	42	1	12	33	2	15	38	1	12	35
WCA-NM	0	7	37	0	12	29	1	12	33	2	9	41	1	12	33	2	15	38	1	12	35
LIMBO	0	0	64	0	0	54	0	0	55	0	0	64	0	0	58	0	0	57	0	0	55
K-means	2	22	71	1	15	60	3	19	64	2	16	60	1	14	53	3	18	65	3	19	64
ARC				6							24							63			
ZBR-tok				4							16							65			
ZBR-uni				4							23							47			
Dir. Struct.				28							45							75			

TABLE 21: $c2c_{cvg}$ results for majority(50%), moderate(33%) and weak(10%) matches for Hadoop.

across all projects. This represents 35 paired samples per metric. We report these results in Tables 22 and 23.

When comparing include dependencies and symbol dependencies with interface-only dynamic-bindings resolution, for 4 out of 6 metrics—a2a, TurboMQ, and two $c2c_{cvg}$ metrics (majority and moderate matches)— we found that the p-values are inferior to 0.001, suggesting our results are statistically significant. For these four metrics, we observed an effect size varying from small ($c2c_{cvg}$ metrics) to large (a2a).

When comparing include dependencies and symbol dependencies with CHA, we found that the p-values are inferior to 0.02, for 3 out of 6 metrics. The effect size for these 3 metrics vary from negligible ($c2c_{cvg}$ metrics) to large (a2a).

$c2c_{cvg}$ with weak coverage not being significant indicates that the type of dependency does not matter for producing a "weak" approximation of the ground-truth architecture. However, when attempting to obtain a better architecture (i.e. with a moderate coverage), working with symbol dependencies with interface-only dynamic-bindings resolution is preferable.

6.1.2 Summary of RQ1

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. Especially, using a2a metrics, we observed a statistically significant improvement coupled with a large effect size in favor of symbol dependencies with interface-only dynamic-bindings.

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.

6.2 RQ2: What is the impact of different input factors, such as the granularity level, the use of transitive dependencies, the use of different symbol dependencies and dynamic-bindings graph construction algorithms, on existing architecture recovery techniques?

There are several types of dependencies that can be used as input to recovery techniques. First, we can break symbol dependencies based on the type of symbol (functions, global variables, etc.). Another important factor to take into consideration concerns the way we resolve dynamic bindings. Finally, we also look at the transitivity and the granularity level of the dependencies. Those different factors are considered as important for other software analyzes and have a significant impact on the quality of the recovered architectures.

6.2.1 Impact of Function Calls and Global Variables

Function calls are the most common type of dependencies in a system. The question we study pertains to whether the most common dependencies have the most significant impact on the quality of the recovery techniques.

Global variables typically represent a small, but important part of a program. Global variables are a convenient way to share information between different functions of the same program. If two functions use the same global variables, they might be similar and the files they belong to could be clustered together. Two functions may, in fact, be dependent on each other if they both utilize the same global variable. Therefore, adding a global-variable-usage graph to the function-call graph could help connect similar elements, that do not directly interact with one another via function calls.

We do not use global variable dependencies alone, because for most of the systems, only a minority of elements accesses global variables. Therefore, by considering global variable dependencies alone, we would miss a large number of elements of the system, making several metrics inaccurate. Instead, we measure the improvement on the quality of the recovered architectures obtained by adding global variable dependencies to functions calls compared to using function calls alone. Static analysis using LLVM can detect global variables for C and C++ projects. For Java projects, we consider variables containing the **static** keyword as an approximation of C/C++ global variables.

Overall, MoJoFM and normalized TurboMQ values for function calls alone and symbol dependencies are highly similar. However, for $c2c_{cvg}$ and a2a, results from all symbol dependencies are significantly better than results from function calls alone. For example, a2a results are, on average 16.8 pp better when using all symbol dependencies available (S-CHA) than when using function calls alone. Despite the fact that function calls have a major impact on the accuracy of architecture recovery techniques, using function calls alone is not sufficient for obtaining accurate recovered architectures.

The impact of global variable usage is minor. For example, on average, adding global variable usage to function call dependencies improves the results by 0.1 pp according to a2a. The impact of global variables is reduced because of the small number of global variable accesses in the projects used in our study. For example, Chromium's C and C++ Style Guide [3] discourages the use of global variables. We acknowledge that our results would likely be different for a system relying heavily on global variables, such as the Toyota ETCS system, which contains about 11,000 global variables [67], [68].

We performed statistical tests and confirmed that symbol dependencies are better than functions alone for architecture recovery for 3 out of 6 metrics (p-value <0.05), with an effect size varying from small to medium in favor of symbol dependencies with interface-only resolution. In addition, we did not observe a statistical difference among architectures recovered from dependencies involving functions alone, and dependencies involving both functions and global variables. It confirms that global variable usage is not a key dependency for accurate architecture recovery for the projects we studied.

3. https://www.chromium.org/developers/coding-style

TABLE 22: Wilcoxon Signed Rank when comparing different types of dependencies.

Metrics	p-values				
	Inc. vs S-CHA	**Inc. vs S-Int**	**S-Int vs Trans.**	**S-Int vs Funct.**	**Funct. vs F-GV**
MoJoFM	.87	.06	.004	.60	.59
a2a	<.001	<.001	.61	<.001	.32
TurboMQ	.03	<.001	<.001	.28	.28
$c2c_{cvg}$ _weak	.20	.10	.02	.007	.77
$c2c_{cvg}$ _mod	.02	<.001	.02	.006	.82
$c2c_{cvg}$ _maj	.22	.003	.01	.24	.64

TABLE 23: Cliff's δ Effect Size Tests. A negative value indicates that the effect size favor of the first dependency type listed. The interpretation of the effect size is indicated in parenthesis. neg. stands for negligible and med. for medium.

Metrics	Effect Size				
	Inc. vs S-CHA	**Inc. vs S-Int**	**S-Int vs Trans.**	**S-Int vs Funct.**	**Funct. vs F-GV**
MoJoFM	.03 (neg.)	.11 (neg.)	-.19 (small)	.05 (neg.)	-.01 (neg.)
a2a	.96 (large)	.64 (large)	-.10 (neg.)	-.37 (med.)	.02 (neg.)
TurboMQ	.36 (med.)	.39 (med.)	-.48 (large)	.08 (neg.)	.01 (neg.)
$c2c_{cvg}$ _weak	.12 (neg.)	.10 (neg.)	-.14 (neg.)	-.20 (small)	-.01 (neg.)
$c2c_{cvg}$ _mod	.09 (neg.)	.17 (small)	-.14 (neg.)	-.17 (small)	.03 (neg.)
$c2c_{cvg}$ _maj	.05 (neg.)	.17 (small)	-.23 (small)	-.11 (neg.)	-.006 (neg.)

6.2.2 *Impact of Dynamic-bindings Resolution*

Dynamic-bindings resolution is a known problem in software engineering, and several possible strategies for addressing it have been proposed [11]–[13]. Due to the high number of dynamic bindings in C++ and Java projects, the type of resolution chosen when extracting symbol dependencies can significantly impact the accuracy of recovery techniques. However, it is unclear which type of dynamic-bindings resolution has the greatest impact on the architecture of a system. To determine which dynamic-bindings resolution to utilize for recovery techniques, we evaluate three different resolution strategies: (1) ignoring dynamic bindings, (2) interface-only resolution, and (3) CHA-based resolution.

Ignoring dynamic bindings is the easiest solution to follow. More importantly, including it as a possible resolution strategy allows us to determine whether doing any dynamic bindings analysis improves recovery results.

For interface-only resolution, we only consider the interface of the virtual functions as being invoked and discard potential calls to derived functions. This is the simplest resolution that can be performed that does not ignore dynamic bindings.

For the third resolution strategy, we use Class Hierarchy Analysis (CHA) [13], which is a well-known analysis that is computationally costly to perform. For this type of resolution, we consider all the derived functions as potential calls. This resolution also creates a larger dependency graph than interface-only resolution.

The results obtained when ignoring dynamic bindings are shown in column No DyB. The results for symbol dependencies obtained with CHA and Interface-only dynamic-bindings resolution are respectively presented in column S-CHA and S-Int. Bash, written in C, is the only project which does not contain any dynamic bindings.

The results obtained when discarding dynamic bindings (column No DyB) are generally not as good as with other symbol dependencies. According to a2a, using only non-dynamic-bindings dependencies reduces the accuracy of the recovery techniques for all projects and all techniques when compared to using dynamic bindings and the average a2a results without dynamic bindings are 11 pp lower than the results with CHA and 6 pp lower than the results with interface-only resolution.

There are a few exceptions for Chromium, Hadoop, and ArchStudio with the metrics MoJoFM and normalized TurboMQ. The reason for unexpectedly high results with MoJoFM and normalized TurboMQ is that using partial symbol dependencies is not well handled by those two metrics. Using partial symbol dependencies —in our case, we discard symbol dependencies that are dynamic bindings— results in (1) a significant mismatch of files between the ground-truth architecture and the recovered architectures, and (2) a disconnected dependency graph. The file mismatches create artificially high MoJoFM results, and the disconnected dependency graphs can lead to extremely high or even perfect normalized TurboMQ scores, as it is the case for ArchStudio when using Bunch without dynamic-bindings dependencies. $c2c_{cvg}$ results are not conclusive either way.

When looking at interface-only and CHA resolutions, we observe a difference in behavior of the two Java projects and the two C/C++ projects. For Java-based Hadoop and ArchStudio, using an interface-only resolution seems to greatly improve the results over using CHA. Those results are obtained for both projects and for all metrics, with only two exceptions for $c2c_{cvg}$ in Hadoop (Table 20) where using CHA provides slightly better results. On average, according to normalized TurboMQ, using interface-only resolution improves the results by 20 pp for ArchStudio and Hadoop. However, for C++-based ITK and Chromium, the normalized TurboMQ results are improved by 6 pp when using CHA for dynamic-bindings resolution.

There could be several reasons for this difference. First, the two C++ projects are between 10 and 200 times larger than the two Java projects we studied. It is possible that a complete analysis of dynamic bindings only becomes necessary for large projects with many complex virtual objects. Second, in Java, methods are virtual by default, while in C++, methods have to be declared as virtual by using the keyword `virtual`. C and C++ developers also have the possibility to use function pointers instead of dynamic bindings, which are currently not handled properly by our symbol dependency extractor. Those two elements could also be a reason why we observed different affects of dynamic-bindings resolutions for C++ and Java projects.

Our overall results indicate that, to obtain a more accurate recovered architecture, the choice of the dynamic-bindings resolution algorithm depends on the project studied. Specifically, if the project contains a high number of dynamic bindings, CHA is likely to produce better recovery results. Otherwise, interface-only resolution is preferable. Ignoring dynamic bindings is ill-advised in most cases.

6.2.3 Transitive vs. Direct Dependencies

A transitive dependency can be built from direct dependencies. For example, if A depends on B, and B depends on C, then A transitively depends on C. Recovery techniques can use as input (1) direct dependencies only or (2) transitive dependencies. To compare direct dependencies against transitive dependencies, we run a transitive closure algorithm on the symbol dependencies and study the effect of adding transitive dependencies on the accuracy of architecture recovery. We did not use include dependencies for this study because, as explained in Section 3.1.3, include dependencies for C and C++ projects are not direct dependencies. Furthermore, we did not include Chromium because the algorithm generating transitive dependencies does not scale to that size, even when we tried to use advanced computational techniques, such as Crocopat's use of binary decision diagrams [46]. For ITK, although we were able to obtain transitive dependencies, none of the architecture recovery techniques scaled to its size. Therefore, we cannot report those results. Results for Bash, Hadoop, and ArchStudio, are reported in the tables corresponding to the different metrics (column Trans).

When comparing the results obtained with direct (Sym for Bash and S-Int for Hadoop and ArchStudio) and transitive (Trans) symbol dependencies, we observe that using direct dependencies generally provides similar or better results. Results with MoJoFM, normalized TurboMQ, and $c2c_{cvg}$ tend to favor the use of direct dependencies over transitive dependencies (+15 pp on average for normalized TurboMQ when using direct dependencies, +4.9 pp on average for MoJoFM).

According to a2a, using transitive dependencies has a minor impact (-0.33 pp on average) on the results. a2a gives importance to the discrepancy of files between the recovered architecture and the ground truth. As no files are added or removed when obtaining the transitive dependencies from the direct dependencies, this discrepancy is exactly the same between the direct and transitive dependencies. This is why we do not observe a significant difference between direct and transitive dependencies results when using a2a.

When running statistical tests, we found that results from direct dependencies (S-Int) are statistically different from results obtained from transitive dependencies for all metrics, except a2a, confirming our conclusion that direct dependencies have a positive impact on the quality of the recovered architectures.

With fewer dependencies, using direct dependencies is more scalable than transitive dependencies. In summary, direct dependencies help generate more accurate architectures than transitive dependencies in most cases.

6.2.4 Impact of the Level of Granularity of the Dependencies

Results at the module level are reported for ITK and Chromium under the column Mod. of Tables 3, 4, 8, 9, 13, 14, 18, and 19. Module dependencies are obtained by adding information extracted from the configuration files to group files together. This information is written by the developers and could represent the architecture of the project as it is understood by developers. Given the inherent architectural information in such dependencies, it is expected that they would improve a recovery technique's accuracy. Because we only have module dependencies for ITK and Chromium, we do not have enough data points to measure statistical significance of our results. However, results obtained from module-level dependencies tend to be much better than from file-level dependencies. For example, on average, compared to using the best file-level dependencies, using module-level information improves the results by 7.5 pp according to a2a.

Overall, our results indicate that module information, when available, significantly improves recovery accuracy and scalability of all recovery techniques. As shown in Table 1, the number of module dependencies is almost 70 times lower than the number of file dependencies. Because of this reduction in the number of dependencies, we obtain results from all recovery techniques in a few seconds when working at the module level, as opposed to several hours for each technique when working at the file level.

6.3 RQ3: Can existing architecture recovery techniques scale to large projects comprising 10MSLOC or more?

6.3.1 Scalability

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 are less direct than symbol dependencies.

As mentioned in the discussion of the previous research question, working at the module-level significantly reduces the number of dependencies and, therefore, greatly improves the scalability of all dependency-based techniques for large projects. Indeed, at the module-level we were able to obtain results in a few seconds, even for techniques that did not scale with file-level dependencies.

6.3.2 *Metrics vs. Size*

While some algorithms are scalable for large projects, it does not mean that results obtained for large systems are as relevant as results obtained for smaller systems. We verify if automatic architecture recovery techniques perform equally for software of all sizes by measuring the evolution of the architectures' quality, when the size of the projects increases.

Overall, we can see that results for Chromium (the largest project) are generally less accurate than results for Bash, ArchStudio, or Hadoop (the smallest projects). However, results for ITK are generally worst than for Chromium, despite ITK being ten times smaller. Because we only study 5 different projects, we cannot draw clear conclusions. Nonetheless, the fact that results for ITK are worst than for Chromium seems to indicate that the size of the project under study is not the only factor affecting the quality of recovered architectures. Other factors such as the programming language, the coding style, and the use of dynamic bindings probably also have an impact that we can't measure with only five different projects.

6.4 Comparison with Baseline Algorithms

To see whether software architecture recovery algorithms are effective, we compare their results with two baseline recovered architecture.

For the first baseline, we recovered the architecture based on the directory structure of the project. According to a2a results, all recovery techniques performed better than the baseline for Bash, ITK and Chromium. A similar trend can be observed for TurboMQ and $c2c_{cvg}$, with a few exception (e.g. WCA and Limbo for Chromium for TurboMQ). This seems to indicate that architecture recovery techniques might be helpful to improve the architecture of these projects.

For ArchStudio and Hadoop, the directory structure-based architecture consistently outperforms architectures recovered with other algorithms, except for TurboMQ for which results are less consistent. This seems to indicate that ArchStudio and Hadoop already have a directory relatively similar to their ground truth architecture and that architecture refactoring might not be necessary for these two projects.

Our second baseline consists in comparing the algorithms specifically designed for architecture recovery (ACDC, Bunch, WCA, and Limbo) with the results obtained from a basic machine learning algorithm, k-means, used with default parameters. Tables 24 and 25 show the statistical significance and the effect size of the difference between K-means and other algorithms, independently from the dependencies used. ACDC is the only algorithm that produces equivalent or better results than K-means consistently, for all metrics. WCA-UE and Limbo always produce worst results than K-means. Finally, the two Bunch algorithms produce better results than K-means only for some metrics.

The three techniques that performed consistently worst than the baseline algorithm are all hierarchical clustering algorithms. It is possible that techniques based on hierarchical clustering are not adapted to recover flat architectures. Results could be different for other projects or ground-truth architectures.

6.5 Summary of Results

Overall, we discovered three main findings from our study. First, using accurate symbol dependencies improves the quality of the recovered architectures. Second, using direct dependencies is more scalable and generally improves the quality of the results. Finally, working with high-level dependencies (i.e. module dependencies) is another key factor for scalable and high-quality architecture recovery of large systems.

6.6 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 because of the differences described in Section 5.3. When using the same type of dependencies (Inc) as in our previous study, we observe minor differences for some algorithms. However, on average, the MoJoFM scores only drop by 0.1 pp for all techniques over the scores reported in [10]. 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.

7 Threats to Validity

This section describes several secondary results of our research such as issues encountered with the different metrics, extreme architectures, and guidelines concerning the dependencies, the architecture recovery techniques, and the metrics to use in future work.

7.1 Metrics Limitations

As mentioned in section 5.5, some metrics have limitations and can be biased toward specific architectures. In this section, we explain the limitations we encountered with two

TABLE 24: Wilcoxon Signed Rank for each algorithm when compared to K-means

Metrics	p-values					
	ACDC	**Bunch-NAHC**	**Bunch-SAHC**	**WCA-UE**	**WCA-UENM**	**Limbo**
MoJoFM	<.001	.002	.27	<.001	<.001	<.001
a2a	.42	<.001	<.001	.11	.11	<.001
TurboMQ	<.001	<.001	<.001	<.001	<.001	<.001
$c2c_{cvg}$ _weak	.003	<.001	.004	<.001	<.001	0.004
$c2c_{cvg}$ _mod	.002	<.001	.003	<.001	<.001	<.001
$c2c_{cvg}$ _maj	<.001	<.001	.65	<.001	<.001	<.001

TABLE 25: Cliff's δ Effect Size Tests. A negative value indicates that the effect size is in favor of K-means.

Metrics	Effect Size					
	ACDC	**Bunch-NAHC**	**Bunch-SAHC**	**WCA-UE**	**WCA-UENM**	**Limbo**
MoJoFM	.54	-.25	-.04	-.67	-.64	-.86
a2a	-.005	-.15	-.16	-.08	-.08	-.31
TurboMQ	.52	.49	.51	-.66	-.66	-.53
$c2c_{cvg}$ _weak	.27	-.57	-.45	-.56	-.56	-.11
$c2c_{cvg}$ _mod	.24	.58	-.33	-.57	-.55	-.97
$c2c_{cvg}$ _maj	.42	-.25	-.06	-.55	-.55	-.77

of the metrics we used. Those limitations appeared because, the metrics in question were neither explicitly intended for nor adapted to specific types of dependencies.

The dependencies are often incomplete. For example, include dependencies generally contain fewer files than the ground-truth architecture. The reasons were explained in Section 3.1.3, 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 file name 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.

In addition to the "file mismatch" issue with MoJoFM, we also identified issues with TurboMQ, as discussed in Section 5.5. Replacing TurboMQ by its normalized version yielded an improvement. However, one has to be careful when using normalized TurboMQ. We identified two boundary cases where normalized TurboMQ results are incorrectly high. It is possible to obtain the maximum score for normalized TurboMQ by grouping all the elements of the recovered architecture in a single cluster. As there will be no inter-cluster dependencies, the score will be 100%. We manually checked all recovered architectures to make sure this specific case never happened in our evaluation. The second "extreme case" occurs when the dependency graph used as input is not fully connected. This can happen when using only partial symbol dependencies (i.e., global variable usage, non-dynamic-bindings dependency graph, etc.). In this case, some recovery techniques will create architectures in which clusters are not connected to one another. This also results in a normalized TurboMQ score of 100%. In our evaluation, this issue occurs when using non-dynamic-bindings dependencies for ArchStudio and Chromium in Tables 13 and 15. This is a limit of normalized TurboMQ when using partial dependencies.

Those are specific issues we observed performing our analysis. It is conceivable that biases towards other types of architecture have yet to be discovered. This suggests that a separate, more extensive study on the impact of different architectures on the metrics would be useful in order to obtain a better understanding of those metrics. Such a study has not been performed to date.

Metrics are convenient because they quantify the accuracy of an architecture with a score, allowing comparisons between recovery techniques. Our study has included, developed, adapted, and evaluated a larger number of metrics than prior similar studies. However, the value of this score by itself must be treated judiciously. Obviously, the "best" recovered architecture is the one that is the closest to the ground-truth. At the same time, important questions such as "Is the recovered architecture good enough to be used by developers?", "Can an architecture with an a2a score of 90% be used for maintenance tasks?" cannot be answered by solely using metrics. A natural outgrowth of this work, then would be to involve real engineers in performing maintenance tasks in real settings. Then it would be possible to evaluate the extent to which the metrics are indicative of the impact on completing such tasks. We are currently preparing such a study with the help of several of our industrial collaborators.

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 normalized 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.

7.2 Selecting Metrics and Recovery Techniques

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, $c2c_{cvg}$, and normalized 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.

7.3 Non-uniqueness of Ground-Truth Architectures

There is not necessarily a unique, correct architecture for a system [1], [31]. 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 per project, there is a threat that our study may not reflect other ground-truth architectures. 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. Furthermore, actual developers or architects of the projects aided in the production of our ground-truth architectures, further reducing any bias introduced in those architectures.

7.4 Number of Projects Studied

We have evaluated recovery techniques on only five systems, which limits our study's generalizability to other systems. Adding more projects is challenging. First, manually recovering the ground-truth architecture of a test project is time-consuming and requires the help of an expert with deep knowledge of the project [1]. Second, the projects studied need to be compatible with the tools used to extract dependencies. For example, the C++ projects evaluated need to be compilable with Clang. To mitigate this threat, we selected systems of different sizes, functionalities, architecture paradigms, and languages.

8 Future Work

Dependencies. This paper explores whether the type of dependencies used affects the quality of the architecture recovered, and answers in the affirmative: Each recovery technique improves if more detailed input dependencies are used.

The results in this paper show, however, that any attempted evaluation of architecture recovery techniques must be careful about dependencies: For example, if we look at the best architecture recovery technique to recover Bash, MoJoFM would select a different best technique in four out of five cases with different input dependencies; $c2c_{cvg}$ in 3/5 cases; and normalized TurboMQ in 2/5 cases. a2a is more stable and would select Bunch-SAHC in all the cases, but a2a also shows that most of the techniques perform similarly for Bash when using similar dependencies. If we look at the other projects, we also observe that none of the metrics always pick the same best recovery technique when using different dependencies.

In this paper, we evaluate architecture recovery techniques using source-code dependencies. Other types of dependencies can alternatively be used. For example, one can look at a developers' activity (e.g., files modified together) to obtain code dependencies [69] and further work is necessary to evaluate if completely different types of dependencies such as directory structure, historical information or developer's activity can be use in the context of automatic architecture recovery.

In addition, we do not consider weighting dependencies. For example, consider FileA that uses one symbol from FileB, and FileC that uses 20 symbols from FileB. Intuitively, it seems that FileB and FileC are more connected than FileA and FileB. Unfortunately, the current implementations of the architecture recovery techniques do not consider weighted graphs. Using weighted dependencies could also be a way to improve the quality of the recovered architectures.

Nested architectures. The architecture recovery techniques evaluated in this study all recover "flat", i.e., non-hierarchical architectures. We focus on flat architectures for several reasons. First, for 4/5 systems we only have access to a flat ground-truth architecture. Second, the existing automatic architecture recovery techniques we evaluate only recover flat architectures.

In previous work on obtaining ground-truth architectures [1], results indicate that architects do not necessarily agree as to the importance of having a nested or flat architecture. However, when discussing with Google developers during the recovery of Chromium's ground truth, it appeared that they view their architecture as a nested architecture in which files are clustered into small entities, themselves clustered into larger entities. Some work has been done on improving metrics to compare nested architectures [70], [71], but little work has been done on proposing and evaluating automatic techniques for recovering nested architectures. A proper treatment of nested architectures, while out of scope of this paper, is an important area for future research.

Multiple Ground-Truth Architectures. Our present work relies on several metrics used for evaluation of architecture recovery, some of which require a ground-truth architecture that might not be unique. More empirical work is needed to explore the idea of multiple ground-truth architectures for a given system. One possible direction is to conduct 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. Ground-truth architectures are important for quality architecture-recovery evaluation and deserve further examination.

9 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 also study the effect of different factors on the accuracy and scalability of recovery techniques, such as the type of dynamic bindings resolution, the granularity-level of the dependencies and whether the dependencies are direct or transitive. We studied nine variants of six architecture recovery techniques on five open-source systems. To perform our evaluation, we recovered the ground-truth architecture of Chromium, and updated ArchStudio and Bash architectures. In addition, we proposed a simple but novel submodule-based architecture recovery technique to recover preliminary versions of ground-truth architectures. In general, each recovery technique extracted a better quality architecture when using symbol dependencies instead of the less-detailed include dependencies. Working with direct dependencies at module level also helps with obtaining a more accurate recovered architecture. Finally, it is important to carefully choose the type of dynamic-bindings resolution when working with symbol dependencies, as it can have a significant impact on the quality of the recovered architectures.

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. [72] 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 architecture recovery that are worthy of future exploration, such as: recovering nested architectures; evaluating the usefulness of the recovered architecture to do specific maintenance tasks; and resolving function pointers and dynamic bindings.

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. F. Bacon and P. F. Sweeney, “Fast static analysis of c++ virtual function calls,” *ACM Sigplan Notices*, vol. 31, no. 10, pp. 324–341, 1996.

[12] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin, *Practical virtual method call resolution for Java*. ACM, 2000, vol. 35, no. 10.

[13] J. Dean, D. Grove, and C. Chambers, “Optimization of object-oriented programs using static class hierarchy analysis,” in *ECOOP95Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*. Springer, 1995, pp. 77–101.

[14] 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.

[15] A. E. Hassan, Z. M. Jiang, and R. C. Holt, “Source versus object code extraction for recovering software architecture,” in *Reverse Engineering, 12th Working Conference on*. IEEE, 2005, pp. 10–pp.

[16] J.-M. Favre, “Cacophony: Metamodel-driven software architecture reconstruction,” in *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*. IEEE, 2004, pp. 204–213.

[17] Y. Tsuchitoi and H. Sugiura, “10 mloc in your office copier,” *IEEE software*, vol. 28, no. 6, p. 93, 2011.

[18] S. Brahler, “Analysis of the android architecture,” *Karlsruhe institute for technology*, vol. 7, 2010.

[19] 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.

[20] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger, “Comparing software architecture recovery techniques using accurate dependencies,” *Proc. ICSE (SEIP)*, 2015.

[21] G. C. Murphy, D. Notkin, and K. Sullivan, “Software reflexion models: Bridging the gap between source and high-level models,” *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. 4, pp. 18–28, 1995.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] J.-M. Favre, F. Duclos, J. Estublier, R. Sanlaville, and J.-J. Auffre, "Reverse engineering a large component-based software product," in *Software Maintenance and Reengineering, 2001. Fifth European Conference on*. IEEE, 2001, pp. 95–104.

[29] P. K. Laine, "The role of sw architecture in solving fundamental problems in object-oriented development of large embedded sw systems," in *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*. IEEE, 2001, pp. 14–23.

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

[31] 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.

[32] 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.

[33] L. O'Brien, C. Stoermer, and C. Verhoef, "Software architecture reconstruction: Practice needs and current approaches," DTIC Document, Tech. Rep., 2002.

[34] M.-A. D. Storey, K. Wong, and H. A. Müller, "Rigi: a visualization environment for reverse engineering," in *Proceedings of the 19th international conference on Software engineering*. ACM, 1997, pp. 606–607.

[35] H. M. Kienle and H. A. Müller, "Rigian environment for software reverse engineering, exploration, visualization, and redocumentation," *Science of Computer Programming*, vol. 75, no. 4, pp. 247–263, 2010.

[36] R. Kazman and S. J. Carrière, "Playing detective: Reconstructing software architecture from available evidence," *Automated Software Engineering*, vol. 6, no. 2, pp. 107–138, 1999.

[37] P. J. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. A. Muller, J. Mylopoulos, S. G. Perelgut, M. Stanley, and K. Wong, "The software bookshelf," *IBM systems Journal*, vol. 36, no. 4, pp. 564–593, 1997.

[38] H. Muller, "Integrating information sources for visualizing java programs," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. IEEE Computer Society, 2001, p. 250.

[39] J. Koch and K. Cooper, "Aovis: A model-driven multiple-graph approach to program fact extraction for aspectj/java source code," *Software Engineering: An International Journal*, vol. 1, 2011.

[40] N. Synytskyy, R. C. Holt, and I. Davis, "Browsing software architectures with lsedit," in *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*. IEEE, 2005, pp. 176–178.

[41] Understand, Scitools.com. https://scitools.com.

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

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

[44] 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.

[45] R. W. Floyd, "Algorithm 97: shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

[46] D. Beyer, "Relational Programming with CrocoPat," in *Proc. ICSE*. Shanghai, China: IEEE, 2006, pp. 807–810.

[47] 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.

[48] P. Wang, "Generating accurate dependencies for large software," 2013.

[49] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *SOFTWARE - PRACTICE AND EXPERIENCE*, vol. 30, no. 11, pp. 1203–1233, 2000.

[50] S. Teller, *Data Visualization with D3.Js*. Packt Publishing, 2013.

[51] 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.

[52] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet Allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, Mar. 2003.

[53] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural change in open-source software systems," in *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 2015, pp. 235–245.

[54] A. Brown and G. Wilson, in *The Architecture of Open Source Applications*. Lulu, 2011.

[55] 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*, vol. 12, pp. 2825–2830, 2011.

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

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

[58] 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 *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR 2015)*, 2015.

[59] 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.

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

[61] J. Munkres, "Algorithms for the assignment and transportation problems," *Journal of the Society for Industrial and Applied Mathematics*, vol. 5, no. 1, pp. 32–38, 1957.

[62] 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.

[63] M. Arzoky, S. Swift, A. Tucker, and J. Cain, "Munch: An Efficient Modularisation Strategy to Assess the Degree of Refactoring on Sequential Source Code Checkings," in *Proc. ICST Workshops*, 2011, pp. 422–429.

[64] A. S. Mamaghani and M. R. Meybodi, "Clustering of Software Systems Using New Hybrid Algorithms," in *Proc. CIT*, 2009, pp. 20–25.

[65] B. S. Mitchell and S. Mancoridis, "On the Automatic Modularization of Software Systems Using the Bunch Tool," *IEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 193–208, Mar. 2006.

[66] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's δ for evaluating group differences on the NSSE and other surveys?" in *annual meeting of the Florida Association of Institutional Research, February*, 2006, pp. 1–3.

[67] P. Koopman, "A case study of toyota unintended acceleration and software safety," *Presentation. Sept*, 2014.

[68] M. Kirsch, V. Regenie, M. Aguilar, O. Gonzalez, M. Bay, M. Davis, C. Null, R. Scully, and R. Kichak, "Technical support to the national highway traffic safety administration (nhtsa) on the reported toyota motor corporation (tmc) unintended acceleration (ua) investigation," *NASA Engineering and Safety Center Technical Assessment Report (January 2011)*, 2011.

[69] M. Konôpka and M. Bieliková, "Software developer activity as a source for identifying hidden source code dependencies," in *SOFSEM 2015: Theory and Practice of Computer Science*. Springer, 2015, pp. 449–462.

[70] M. Shtern and V. Tzerpos, "Lossless comparison of nested software decompositions," in *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*. IEEE, 2007, pp. 249–258.

[71] ——, "A framework for the comparison of nested software decompositions," in *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*. IEEE, 2004, pp. 284–292.

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