Measuring the Impact of Code Dependencies on Software Architecture Recovery Techniques

by

Thibaud Lutellier

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2015

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

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

First, we study the impact of leveraging more accurate symbol dependencies on the accuracy of architecture recovery techniques. In addition, we evaluate other factors of the input dependencies such as the level of granularity, the impact of virtual call resolution, global variable usage and whether using direct dependencies provides better results than using transitive dependencies. Previous studies have not extensively studied how the quality of the input might affect the quality of the output for architecture recovery techniques. Second, we study a system (Chromium) that is substantially larger (10 million lines of code) than those included in previous studies. Obtaining the ground-truth architecture of Chromium involved two years of collaboration with its developers. As part of this work we developed a new submodule-based technique to recover preliminary versions of ground-truth architectures. The other systems that we study have been examined previously. In some cases, we have updated the ground-truth architectures to newer versions, and in other cases we have corrected newly discovered inconsistencies.

Our evaluation of nine variants of six state-of-the-art architecture recovery techniques on 8 types of dependencies shows that symbol dependencies generally produce architectures with higher accuracies than include dependencies. We also observed that using a higher level of granularity (i.e., module level) and direct dependencies helps generating better architectures.

Despite this improvement, the overall accuracy is low for all recovery techniques. The results suggest that (1) in addition to architecture recovery techniques, the type of dependencies used as their inputs is another factor to consider for high recovery accuracy, and (2) more accurate recovery techniques are needed. Our results show that some of the studied architecture recovery techniques (ACDC, Bunch-SAHC, WCA and ARC) scale to the 10M lines-of-code range (the size of Chromium), whereas others do not.

Acknowledgements

I would like to take this opportunity to thank my supervisor, Professor Lin Tan, for investing her time and providing me guidance. Without her, this thesis would not have been possible. In addition, I would like to thank the readers, Professor Derek Rayside, for suggesting the idea of studying the accuracy of dependencies and Professor Michael Godfrey, for his valuable time and comments. I would also like to extend my appreciation to my project colleague, Devin Chollak, for his help concerning evaluating Java projects. Joshua Garcia, Nenad Medvidovic also deserve my gratefulness for providing architecture recovery techniques and giving me valuable feedbacks. Finally, I would like to thank Robert Kroeger for his help in obtaining Chromium and Pei Wang for his feedback and help with extracting dependencies for C++ projects.

Table of Contents

List of Tables

List of Figures

Chapter 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 [26]. 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 [4, 16, 27, 32, 42, 59]. 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 [4, 43, 66], the latest study [25] 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 relatively 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 to create virtual functions. Several techniques exist to build virtual call graphs [6, 18, 56] and, despite the existence of an early study [55] of the impact of call graph construction algorithms on basic architecture recovery, no work has been done to study the effect of virtual calls resolution on recent architecture recovery techniques.

The last question we study pertains to the scalability of existing architecture recovery techniques. The largest software system used in the published evaluations of architecture recovery techniques comprises 4MSLOC, and it revealed the scalability limits of several recovery techniques [25]. The size of software is increasing, and many software projects

are significantly bigger than 4MSLOC. For example, the Chromium open-source browser contains nearly 10MSLOC. In this paper, we test whether existing architecture recovery techniques can scale to software of such size.

To this end, this paper compares the *same* nine variants of six architecture recovery techniques from the previous study [25] 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 virtual call graph construction algorithms, on existing architecture recovery techniques?

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

This paper makes the following contributions:

- We compared nine variants of six architecture recovery techniques using eight types of dependencies at different levels of granularity to assess their affects on accuracy. This is the first substantial study to compare 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 virtual call graph resolution algorithm, the granularity of the dependencies, and whether such dependencies are direct or transitive (**RQ2**).

- Our results show that the accuracy is low for all studied techniques. This corroborates past results [25] 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 architectures of one open-source project, Chromium (svn revision 171054). At 10M, to the best of our knowledge, Chromium is the largest project used for evaluating architecture recovery techniques to date. The ground-truth architecture of Chromium was not available previously. We obtained it through *two years* of regular discussions and meetings with Chromium developers.

We also updated the architectures of Bash and ArchStudio that were reported in [26]. 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 [26], 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**).

Chapter 2

Related Work

2.1 Comparison of Software Architecture Recovery Techniques

This thesis builds on work that was previously reported in [39]. Novelty with respect to this previous work includes a study of the impact of virtual call 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 using transitive dependencies improves the accuracy of recovery techniques.

Many architecture recovery techniques have been proposed [4, 16, 27, 32, 42, 51, 59]. The most recent study [25] 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) [27] and Algorithm for Comprehension-Driven Clustering (ACDC) [59]—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 [4, 25, 32, 37, 43, 66].

The results of the different studies are not always consistent. scaLable InforMation BOttleneck (LIMBO) [3], a recovery technique leveraging an information loss measure, and ACDC performed similarly in one study [4]; however, in a different study, Weighted Combined Algorithm (WCA) [44], a recovery technique based on hierarchical clustering, outperformed Complete Linkage (CL) [44]. In yet another study, CL is shown to be

generally better than ACDC [66]. In the most recent study, ARC and ACDC surpass LIMBO and WCA [25]. Wu et al. [66] 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 [22,23,58]; other techniques use static and dynamic dependency graphs [4]. Previous work [55] examined the effect of different polymorphic call graph construction algorithms on automatic clustering. In their work [42], Mancoridis et al. tried to improve the architectures recovered by Bunch by removing omnipresent dependencies.

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.

Garcia et al. [26] describe a method to recover the ground-truth architectures of four open-source systems. The method involves extensive manual work, and the mean cost of recovering the ground-truth architecture of seven systems ranged from 70KSLOC to 280KSLOC was 107 hours.

Bowman et al. [10] and Xiao et al. [67] recovered the ground-truth architectures of the Linux kernel 2.0 and Mozilla 1.3 respectively. The Linux kernel and Mozilla are large systems, but the evaluated versions are more than a decade old. The version of the Linux kernel recovered was from 1996 and at that time, it contained only 750KSLOC. Mozilla 1.3 is from 2003 with 4MSLOC.

Grosskurth et al. [28] studied the architecture and evolution of web browsers and provide guidance for obtaining a reference architecture for web browsers. Their work does not address the challenges of recovering an accurate ground-truth architecture in general. In addition, it is not clear if their approach is accurate for modern web browsers such as

Chromium, which use new design principles such as a modern threading model for tabbed browsing.

Several commercial tools such as Lattix [61] and Structure101 [14] are used to ensure the quality of a given architecture and monitor its evolution. As far as we know, none of those tools claim to generate automatically the ground-truth architecture of a project.

Chapter 3

Approach

Our approach consists of two parts: the extraction of dependencies and obtaining a ground-truth architecture. In the rest of this chapter, we describe the manner in which we extract the dependencies we study for C/C++ (Section 3.1) and Java (Section 3.2), discuss why symbol dependencies are more accurate than include dependencies (Section 3.3), and elaborate on our approach for obtaining ground-truth architectures (Section 3.4).

3.1 Obtaining Dependencies for 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 [62]. 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. To group code-level entities at the module level, we extract module information from the build files of the project provided by the developers. Transitive dependencies are obtained for all projects using the Floyd-Warshall [24] algorithm. Because the Floyd-Warshall algorithm did not scale for Chromium, we also tried to use Crocopat [8] 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 [25].

3.2 Obtaining Dependencies for 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 [54]. 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. 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 projects without well-defined modules, using C++ namespaces or Java packages could be a good alternative to modules defined in the configuration files.

3.3 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 Chapter 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 (e.g., `java.util.*`) are overly inclusive. Include dependencies are therefore significantly less accurate than symbol dependencies.

3.4 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

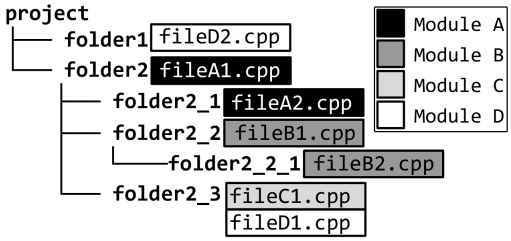

Figure 3.1: Example Project Layout

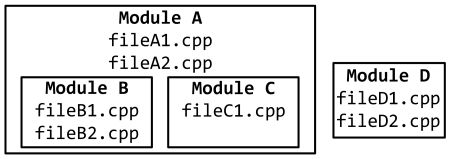

Figure 3.2: Example Project Submodules

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. We look at the common parent directories because some modules may have files in many different directories. 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 3.1, while the module structure that we generate is shown in Figure 3.2.

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

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

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 it to the developers for certification. Multiples rounds of verifications, based on developers' feedback, are necessary to obtain an accurate ground-truth architecture. For Chromium, it took *two years* of meetings and email exchanges with Chromium developers to obtain the ground truth.

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 [19, 26] 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 the submodule-based approach as a starting point for recovering several ground truths, by having different recoverers and receiving feedback from different developers.

Prior work [26] 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 [26]).

Chapter 4

Selected Recovery Techniques

We select the same nine variants of six architecture recovery techniques as in previous work [25] for our evaluation. Four of the selected techniques (ACDC, LIMBO, WCA, and Bunch [42]) use dependencies to determine clusters, while the remaining two techniques (ARC and ZBR [16]) use textual information from source code. We include techniques that do not use dependencies to (1) assess the accuracy of finer-grained, accurate dependencies against these information retrieval-based techniques and to (2) determine their scalability.

Algorithm for Comprehension-Driven Clustering (ACDC) [59] is a clustering technique for architecture recovery. We included ACDC because it performed well in several previous studies [4, 25, 43, 66]. 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 [41, 42] 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 [25], we evaluate two versions of the Bunch hill-climbing

Table 4.1: 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

algorithms—Nearest and Steepest Ascent Hill Climbing (NAHC and SAHC).

Weighted Combined Algorithm (WCA) [44] 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 a recent study [25], UE and UENM performed differently depending on the systems tested, therefore, we evaluate both.

LIMBO [3] 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 [4, 43], except in one recent study [25] where LIMBO achieved surprisingly poor results.

Architecture Recovery using Concerns (ARC) [27] 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) [9], 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 the previous evaluation [25] and thus is important to compare against when evaluating for accuracy.

Similar to ARC, **Zone Based Recovery (ZBR)** [16] 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 [25]. 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 [16] but struggled with memory issues when dealing with larger systems [25].

Chapter 5

Experimental Method

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 4.1.

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 Chapter 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 include dependencies for Chromium. This is the result of both transitive and over-approximation of dependencies, detailed in Section 3.3.

The number of transitive dependencies shown in Table 4.1 for ITK is strikingly high. We leverage Class Hierarchy Analysis (CHA) [18] to build the virtual call dependency graph of symbol dependencies which, in turn, is used for extracting dependencies. For ITK, more than 75% of the dependencies extracted are virtual function calls, as opposed to just 11% for Chromium for example. This high proportion of virtual calls 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 [25], 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 [26], we were unable to use it for our study.

For Chromium, the ground-truth architecture was extracted using the submodule approach outlined in Section 3.4. ITK was refactored in 2013 and its ground-truth architecture, extracted by ITK's developers, is available. ITK developers involved in the ITKv4 project confirmed that this architecture was still valid for ITK 4.5.2.

The version of Bash used in a recent architecture-recovery study [25] was from 1995. Bash has been changed significantly since then (e.g., from 70KSLOC to 115KSLOC). 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 [11].

The scope of our results focuses only on the core of ArchStudio. The ground-truth architecture for ArchStudio was updated, from prior work [26], 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 [25], 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 different types of dependencies. We obtained ACDC and Bunch from their authors' websites. For the other techniques, we used our implementation from the previous study [25]. Each of those implementations was shared with the original authors of the recovery techniques and confirmed as correct [25]. Due to the non-determinism of the clustering algorithms used by ACDC and Bunch, we ran each algorithm five times and reported only the best results. WCA, LIMBO, and ARC can take varying numbers of clusters as input. Based on the number of clusters in the ground-truth architectures, we experimented with 5 to 75 clusters as inputs for Bash and ITK, 25 to 75 for Chromium and 30 to 80 for Hadoop and ArchStudio with an increment of 5 for all cases. ARC also takes a varying number of concerns as input. We experimented with 10 to 150 concerns in increments of 10. We report only the best results for each technique.

5.5 Accuracy Measures

There might be multiple ground-truth architectures for a system [10, 26]; that is, experts might disagree. Therefore, a recovered architecture may be different from a ground-truth architecture used in this paper, but close to another ground-truth architecture of the same project. To mitigate this threat, we selected four different metrics to evaluate recovery techniques. One of the metrics—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.

MoJoFM [64] is defined by the following formula,

$$MoJoFM(M) = (1 - \frac{mno(A, B)}{max(mno(\forall A, B))}) \times 100\% \tag{5.1}$$

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 [25, 32].

Architecture-to-architecture [36] ($a2a$) 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 [36]. 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 [46, 53]. $a2a$ is a distance measure between two architectures:

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

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

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

mto and *aco* are used to calculate the total numbers of the five operations used to transform one architecture into another: additions ($addE$), removals ($remE$), and moves ($movE$) of implementation-level entities from one cluster (i.e., component) to another; as well as additions ($addC$) and removals ($remC$) of clusters themselves.

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

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

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

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 [5, 40, 49]. We implemented the $TurboMQ$ version because it has better performance than $BasicMQ$ [47].

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.

Chapter 6

Results

This chapter 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 6.1-6.20 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 virtual calls (No Vir) or using a class hierarchy analysis of virtual calls (*S-CHA*) or with interface-only resolution of virtual calls (*S-Int*). Bash does not contain virtual calls 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.

Table 6.1: MoJoFM results for Bash.

Algo.	Bash				
	Inc.	**Sym.**	**Trans.**	**Funct.**	**F. GV.**
ACDC	41	59	42	49	50
Bunch-NAHC	44	47	40	52	51
Bunch-SAHC	45	58	41	45	53
WCA-UE	28	37	45	43	43
WCA-UENM	28	34	36	39	38
LIMBO	27	28	30	26	26
ARC	40				
ZBR-tok	35				
ZBR-uni	39				

Table 6.2: *a2a* results for Bash.

Algo.	Bash				
	Inc.	**Sym.**	**Trans.**	**Funct.**	**F. GV.**
ACDC	64	81	81	41	41
Bunch-NAHC	66	86	85	41	41
Bunch-SAHC	68	87	85	41	42
WCA-UE	63	81	81	41	41
WCA-UENM	63	81	81	41	41
LIMBO	63	80	80	39	39
ARC	67				
ZBR-tok	31				
ZBR-uni	32				

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

As explained in section 3.3, 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 6.1-6.20.

Table 6.3: Normalized *TurboMQ* results for Bash.

Algo.	**Bash**				
	Inc.	**Sym.**	**Trans.**	**Funct.**	**F. GV.**
ACDC	10	23	6	29	29
Bunch-NAHC	34	42	28	41	35
Bunch-SAHC	50	40	28	40	37
WCA-UE	1	16	18	28	28
WCA-UENM	23	16	18	28	28
LIMBO	22	24	20	19	20
ARC	16	25	NA		
ZBR-tok	1	3	NA		
ZBR-uni	2	2	NA		

Table 6.4: $c2c_{cvg}$ results for majority(50%), moderate(33%) and weak(10%) matches for Bash.

Algo.	**Bash**											
	Inc.			**Sym.**			**Trans.**			**Funct.**			**F. GV.**		
ACDC	21	50	71	36	79	92	14	42	100	0	7	50	0	7	21
Bunch-NAHC	14	36	71	7	28	85	7	21	50	7	14	50	7	14	57
Bunch-SAHC	14	36	71	21	57	92	7	21	57	0	21	50	0	21	50
WCA-UE	14	36	64	0	21	92	14	28	92	0	14	50	0	14	64
WCA-UENM	0	14	64	0	21	92	0	21	71	0	21	64	0	7	71
LIMBO	0	14	64	0	21	78	0	14	86	0	7	50	0	7	50
ARC	28			57			86								
ZBR-tok	0			0			21								
ZBR-uni	0			0			7								

Table 6.5: MoJoFM results for ITK.

Algo.	ITK						
	Inc.	**S-CHA**	**S-Int**	**No Vir.**	**Mod.**	**Funct.**	**F. GV.**
ACDC	59	56	52	48	84	60	60
B.-NAHC	37	41	39	43	87	48	51
B.-SAHC	33	62	60	53	86	61	61
WCA-UE	31	32	45	45	90	36	36
WCA-UENM	31	32	45	45	89	36	36
LIMBO	31	31	45	38	87	36	36
ARC	59						
ZBR-tok	MEM						
ZBR-uni	MEM						

Table 6.6: $a2a$ results for ITK.

Algo.	ITK						
	Inc.	**S-CHA**	**S-Int**	**No Vir.**	**Mod.**	**Funct.**	**F. GV.**
ACDC	67	74	63	58	84	48	48
Bunch-NAHC	71	80	69	59	87	48	48
Bunch-SAHC	69	80	67	59	86	49	48
WCA-UE	74	82	48	39	90	48	48
WCA-UENM	74	82	48	39	89	48	48
LIMBO	71	80	45	36	87	47	47
ARC	60						
ZBR-tok	MEM						
ZBR-uni	MEM						

Table 6.7: Normalized *TurboMQ* results for ITK.

Algo.	ITK						
	Inc.	**S-CHA**	**S-Int**	**No Vir.**	**Mod.**	**Funct.**	**F. GV.**
ACDC	33	24	18	32	47	40	40
Bunch-NAHC	15	24	28	25	58	43	41
Bunch-SAHC	10	46	32	31	65	54	49
WCA-UE	5	20	7	2	74	22	19
WCA-UENM	15	20	7	2	66	22	19
LIMBO	11	21	9	1	47	20	20
ARC	9	33	NA				
ZBR-tok	MEM						
ZBR-uni	MEM						

Table 6.8: $c2c_{cvg}$ results for majority(50%), moderate(33%) and weak(10%) matches for ITK.

Algo.	ITK						
	Inc.	**S-CHA**	**S-Int**	**No Vir.**	**Mod.**	**Funct.**	**F. GV.**
ACDC	0 0 62	0 0 53	0 0 31	0 0 31	8 8 54	0 8 38	0 8 38
B.-NAHC	0 0 30	0 0 61	0 0 46	0 0 46	15 46 76	0 0 38	0 0 38
B.-SAHC	0 0 0	0 7 92	0 0 46	0 0 38	15 62 92	0 0 38	0 0 38
WCA-UE	0 0 23	0 0 30	0 0 0	0 0 23	38 69 100	0 0 38	0 0 38
WCA-UENM	0 0 23	0 0 23	0 0 0	0 0 23	46 62 100	0 0 38	0 0 38
LIMBO	0 0 23	0 0 23	0 0 0	0 0 0	38 69 100	0 0 30	0 0 30
ARC	7		7		54		
ZBR-tok	MEM						
ZBR-uni	MEM						

Table 6.9: MoJoFM results for Chromium. † Scores denote results for intermediate architectures obtained after the technique timed out.

	Chromium						
Algo.	**Inc.**	**S-CHA**	**S-Int**	**No Vir.**	**Mod.**	**Funct.**	**F. GV.**
ACDC	63	70	73	71	62	71	71
Bunch-NAHC	28	31	24	66	70	29	35
Bunch-SAHC	13†	70†	43†	66	63	39	29
WCA-UE	23	23	23	27	90	29	29
WCA-UENM	23	23	23	27	89	29	29
LIMBO	TO	23	23	26	85	27	27
ARC	45						
ZBR-tok	MEM						
ZBR-uni	MEM						

Table 6.10: *a2a* results for Chromium. † Scores denote results for intermediate architectures obtained after the technique timed out.

	Chromium						
Algo.	**Inc.**	**S-CHA**	**S-Int**	**No Vir.**	**Mod.**	**Funct.**	**F. GV.**
ACDC	71	73	74	64	82	62	62
Bunch-NAHC	69	73	76	66	82	63	63
Bunch-SAHC	60†	71†	66†	66	84	64	62
WCA-UE	70	75	78	68	85	66	66
WCA-UENM	70	75	78	68	84	67	66
LIMBO	TO	71	74	64	84	61	62
ARC	56						
ZBR-tok	MEM						
ZBR-uni	MEM						

Table 6.11: Normalized *TurboMQ* results for Chromium. † Scores denote results for intermediate architectures obtained after the technique timed out.

Algo.	Chromium						
	Inc.	**S-CHA**	**S-Int**	**No Vir.**	**Mod.**	**Funct.**	**F. GV.**
ACDC	15	19	18	20	47	24	24
Bunch-NAHC	4	24	9	100	56	16	19
Bunch-SAHC	2†	30†	11†	3	51	29	11
WCA-UE	0	4	4	83	41	4	4
WCA-UENM	0	4	4	98	44	4	4
LIMBO	TO	4	4	70	37	4	4
ARC	4	10	NA				
ZBR-tok	MEM						
ZBR-uni	MEM						

Table 6.12: $c2c_{cvg}$ 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.	Chromium																					
	Inc.			**S-CHA**			**S-Int**			**No Vir.**			**Mod.**			**Funct.**			**F. GV.**		
ACDC	16	30	80	22	48	92	17	38	87	10	23	82	13	17	29	7	17	80	10	17	80
B.-NAHC	0	0	7	0	0	26	0	0	3	0	0	9	12	25	52	0	0	4	0	3	24
B.-SAHC	0	6	19	14	33	80	7	12	36	4	10	53	14	23	54	0	1	38	0	0	4
WCA-UE	0	0	4	0	0	3	0	0	3	0	0	3	35	65	94	0	0	3	0	1	6
WCA-NM	0	0	4	0	0	3	0	0	3	0	0	3	28	59	94	0	0	3	0	1	6
LIMBO	TO			0	0	0	0	0	0	0	0	0	42	65	83	0	0	0	0	0	0
ARC	3						7									80					
ZBR-tok	MEM																				
ZBR-uni	MEM																				

Table 6.13: MoJoFM results for ArchStudio.

Algo.	ArchStudio						
	Inc.	**S-CHA**	**S-Int**	**No Vir.**	**Trans.**	**Funct.**	**F. GV.**
ACDC	60	60	77	79	71	76	74
Bunch-NAHC	52	46	59	56	50	61	53
Bunch-SAHC	62	48	62	50	53	61	64
WCA-UE	32	31	33	47	33	32	32
WCA-UENM	32	31	33	47	33	32	32
LIMBO	26	25	25	26	25	26	25
ARC	62						
ZBR-tok	48						
ZBR-uni	48						

Table 6.14: *a2a* results for ArchStudio.

Algo.	ArchStudio						
	Inc.	**S-CHA**	**S-Int**	**No Vir.**	**Trans.**	**Funct.**	**F. GV.**
ACDC	71	86	88	83	92	87	88
Bunch-NAHC	71	80	83	76	82	82	84
Bunch-SAHC	72	82	85	76	82	82	83
WCA-UE	71	84	84	82	84	82	83
WCA-UENM	71	84	84	82	84	82	83
LIMBO	67	79	79	74	79	78	79
ARC	87						
ZBR-tok	85						
ZBR-uni	86						

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 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 6.2-6.18) relying on both types of symbol dependencies outperforms relying on include dependen-

Table 6.15: Normalized $TurboMQ$ results for ArchStudio.

Algo.	ArchStudio						
	Inc.	**S-CHA**	**S-Int**	**No Vir.**	**Trans.**	**Funct.**	**F. GV.**
ACDC	66	41	76	84	71	73	74
Bunch-NAHC	79	50	79	100	46	79	84
Bunch-SAHC	76	47	80	100	73	84	81
WCA-UE	1	13	24	69	20	11	20
WCA-UENM	1	13	24	69	20	11	20
LIMBO	48	12	32	38	8	25	28
ARC	26	30	NA				
ZBR-tok	6	17	NA				
ZBR-uni	5	15	NA				

Table 6.16: $c2c_{cvg}$ results for majority(50%), moderate(33%) and weak(10%) matches for ArchStudio.

Algo.	ArchStudio						
	Inc.	**S-CHA**	**S-Int**	**No Vir.**	**Trans.**	**Funct.**	**F. GV.**
ACDC	9 21 47	21 54 77	56 77 93	53 75 89	52 72 86	44 65 77	44 63 77
B.-NAHC	4 9 33	9 11 54	11 19 61	5 19 53	5 12 37	9 19 61	7 25 61
B.-SAHC	7 16 49	7 18 54	11 19 54	11 14 37	9 17 67	5 18 70	9 17 67
WCA-UE	0 9 39	0 7 30	7 18 39	30 37 53	7 18 40	0 14 39	7 18 40
WCA-NM	0 9 39	0 7 30	7 18 39	30 37 53	7 18 40	0 14 39	7 18 40
LIMBO	0 0 77	0 0 91	0 0 93	0 0 91	0 0 84	0 0 88	0 0 84
ARC	21		49			88	
ZBR-tok	4		16			65	
ZBR-uni	4		23			47	

cies 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 virtual calls. As ITK contains a large number of virtual call 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 virtual call dependencies of ITK (S-CHA), using symbol dependencies with a class hierarchy analysis of virtual calls outperforms using include dependencies for all techniques. Similar results are observed for $MoJoFM$, despite a few exceptions where include

Table 6.17: MoJoFM results for Hadoop.

Algo.	Hadoop						
	Inc.	**S-CHA**	**S-Int**	**No Vir.**	**Trans.**	**Funct.**	**F. GV.**
ACDC	24	29	41	41	32	41	41
Bunch-NAHC	29	27	29	34	22	37	29
Bunch-SAHC	32	38	40	41	26	37	38
WCA-UE	14	13	17	40	17	17	17
WCA-UENM	14	13	17	37	17	17	17
LIMBO	17	14	15	16	14	15	15
ARC	49						
ZBR-tok	29						
ZBR-uni	38						

Table 6.18: $a2a$ results for Hadoop.

Algo.	Hadoop						
	Inc.	**S-CHA**	**S-Int**	**No Vir.**	**Trans.**	**Funct.**	**F. GV.**
ACDC	68	81	84	79	80	84	84
Bunch-NAHC	68	81	81	78	80	82	81
Bunch-SAHC	69	83	83	80	79	82	83
WCA-UE	68	80	81	79	81	81	82
WCA-UENM	68	80	81	79	81	81	82
LIMBO	68	80	80	76	80	79	80
ARC	84						
ZBR-tok	81						
ZBR-uni	83						

dependencies outperformed symbol dependencies by a few percentage points. On average, using symbol dependencies respectively improves the accuracy by 9 percentage points (pp) and 5 pp according to $a2a$ and $MoJoFM$. For $MoJoFM$ and $a2a$, the technique obtaining the greatest improvement from the use of symbol dependencies, as compared to include dependencies, is Bunch-SAHC, with an average improvement of almost 17 pp for $MoJoFM$ and a 11.5 pp for $a2a$.

Tables 6.4-6.20 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

Table 6.19: Normalized *TurboMQ* results for Hadoop.

Algo.	Hadoop						
	Inc.	**S-CHA**	**S-Int**	**No Vir.**	**Trans.**	**Funct.**	**F. GV.**
ACDC	48	29	59	65	29	57	29
Bunch-NAHC	44	31	61	81	35	59	35
Bunch-SAHC	50	36	56	70	31	57	31
WCA-UE	2	6	8	37	11	8	11
WCA-UENM	2	6	9	36	11	8	11
LIMBO	3	7	19	26	4	18	4
ARC	10	15	NA				
ZBR-tok	5	10	NA				
ZBR-uni	7	13	NA				

Table 6.20: $c2c_{cvg}$ results for majority(50%), moderate(33%) and weak(10%) matches for Hadoop.

Algo.	Hadoop																				
	Inc.			**S-CHA**			**S-Int**			**No Vir.**			**Trans.**			**Funct.**			**F. GV.**		
ACDC	0	3	43	4	13	39	7	18	49	7	18	45	4	10	37	7	16	52	9	16	52
B.-NAHC	1	3	35	1	4	47	0	12	49	3	9	43	1	7	34	4	10	55	4	9	49
B.-SAHC	1	3	32	4	21	81	4	10	49	7	21	85	1	6	25	4	13	58	3	10	46
WCA-UE	0	7	37	0	13	30	1	15	34	3	10	73	1	13	33	3	19	39	1	16	36
WCA-NM	0	7	37	0	13	30	1	15	34	3	12	69	1	13	33	3	19	39	1	16	36
LIMBO	0	0	64	0	0	81	0	0	79	0	1	81	0	0	84	0	0	82	0	0	79
ARC	21						49						88								
ZBR-tok	4						16						65								
ZBR-uni	4						23						47								

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 6.16, 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 6.4 to 6.20 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.

Tables 6.3 to 6.19 presents the normalized $TurboMQ$ results, which measure the organization and cohesion of clusters independent of ground-truth architectures. Both types of symbol dependencies obtain higher normalized $TurboMQ$ scores than include dependencies. In other words, symbol dependencies help recovery techniques produce architectures with better organization and internal component cohesion than include dependencies.

The overall conclusion from applying these four metrics is that symbol dependencies allow recovery techniques to increase their accuracy for all systems in almost every case, independently of the metric chosen. The different metrics sometimes contradict one another. For example, in the case of the ITK's architecture recovered using ACDC, include dependencies outperform symbol dependencies with CHA, as measured using $MoJoFM$. However, when measured using $a2a$, the opposite result occurs. This difference is likely due to the perspective from which the metrics measure an architecture. Nevertheless, our results tend to be similar for the four metrics, indicating that symbol dependencies increase the accuracy of architecture recovery.

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. Our results also reinforce the observation made in our recent related study [25] that multiple ground-truth architectures for a system may be needed.

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 virtual call 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 virtual calls. 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. If two functions use the same global variables, they might be similar and the files they belong to could be clustered together. Adding global variable dependencies to our dependency 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.

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 16.5 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 0.3 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 [1] 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 [31, 34].

6.2.2 Impact of virtual call resolution

Virtual call resolution is a known problem in software engineering, and several possible strategies for addressing it have been proposed [6,18,56]. Due to the high number of virtual calls 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 virtual call resolution has the greatest impact on the architecture of a system. To determine which virtual call resolution to utilize for recovery techniques, we evaluate three different resolution strategies: (1) ignoring virtual calls, (2) interface-only resolution, and (3) CHA-based resolution.

Ignoring virtual calls is the easiest solution to follow. More importantly, including it as a possible resolution strategy allows us to determine whether doing any virtual call 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 virtual calls.

For the third resolution strategy, we use Class Hierarchy Analysis (CHA) [18], 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 virtual calls are shown in column No Vir. The results for symbol dependencies obtained with CHA and Interface-only virtual call resolution

[1] https://www.chromium.org/developers/coding-style

are respectively presented in column S-CHA and S-Int. Bash, written in C, is the only project which does not contain any virtual calls.

The results obtained when discarding virtual calls (column No Vir.) are generally less good than with other symbol dependencies. According to $a2a$ and $c2c_{cvg}$, using only non-virtual call dependencies reduces the accuracy of the recovery techniques for all projects and all techniques when compared to using virtual calls. According to $a2a$ the average results without virtual calls 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 virtual calls—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 virtual call dependencies.

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 three exceptions for $c2c_{cvg}$ in Hadoop (Table 6.16) 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 7 pp when using CHA for virtual-call 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 virtual calls 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 virtual calls, 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 virtual-call resolutions for C++ and Java projects.

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

6.2.3 Direct vs. transitive 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 3.3, 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 [8]. 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 (+11 pp on average for normalized $TurboMQ$ when using direct dependencies, +4.6 pp on average for $MoJoFM$).

According to $a2a$, using transitive dependencies has a minor impact (-0.7 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 $a2a$.

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 6.9, 6.5, 6.6, 6.10, 6.11, 6.7, 6.8, and 6.12. 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.

Overall, our results indicate that module information, when available, significantly improves recovery accuracy and scalability of all recovery techniques. As shown in Table 4.1, 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. On average, compared to using the best file-level dependencies, using module-level information improves the results by 8 pp according to $a2a$.

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

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

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

improves MQ. Bunch-SAHC's investigation of all neighboring architectures makes it less scalable than Bunch-NAHC.

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

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

The use of symbol dependencies improves the recovery techniques' scalability over include dependencies for large projects (i.e., ITK and Chromium). The main reason for this phenomenon is that include dependencies 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.

6.4 Comparison with the Prior Work

As previously mentioned, three of our subject systems were also used in the previous study [25]. 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 the previous study, we observe that the $MoJoFM$ scores drop by 6 pp on average for all techniques over the scores reported in [25]. It is possible that the newer version of Bash is more complex and its architecture harder to automatically recover. In the cases of Hadoop and ArchStudio, the previous study used a different level of granularity (class level), which makes comparison with current work irrelevant.

Chapter 7

Threats to Validity

There is not necessarily a unique, correct architecture for a system [10, 26]. Recovering ground-truth architectures require heavy manual work from experts. Therefore, it is challenging to obtain different certified architectures for the same system. As we are using only one ground-truth architecture, there is a threat that our study is biased toward that specific architecture. To reduce this threat, we use four different metrics, including one independent of the ground-truth architecture. Two of the metrics used in this study were developed by some authors of this paper, which might have caused a bias in this study. However, all four metrics, including metrics developed independently, follow the same trend—symbol dependencies are better than include dependencies—which mitigates some of the potential bias.

The metrics chosen in this paper measure the similarity and quality of an architecture at different levels—the system level (measured by $MoJoFM$ and $a2a$), the component level (measured by $c2c_{cvg}$) and the dependency-cohesion level (measured by 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.

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

Chapter 8

Discussion

This chapter 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.

8.1 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 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 3.3, 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-virtual call 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-virtual call dependencies for ArchStudio and Chromium in Tables 6.11 and 6.15. 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.

8.2 Architecture Recovery Techniques

The architecture recovery techniques evaluated in this study all recover "flat", i.e., non-hierarchical architectures. This design choice is not necessarily the best. Indeed, 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. This kind of architecture is not well supported by current architecture recovery techniques and metrics. Possible future work may focus on developing or updating architecture recovery techniques to be able to obtain a hierarchical organization of the information which is potentially more useful to developers.

8.3 Dependencies Matter for Evaluating Architecture Recovery Techniques

This paper explores whether the type of dependencies used affects the quality of the architecture recovered, and answers in the affirmative: each recovery technique gets better if more detailed input dependencies are used. This paper has not focused on the question of which architecture recovery technique is best. The results in this paper show, however, that any attempted evaluation of architecture recovery techniques must be careful about dependencies: 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 [33].

In addition, we do not consider the weights on the 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 be a way to improve the quality of the recovered architectures.

8.4 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.

Chapter 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 virtual-call 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 virtual-call 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. [12] 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.

More empirical work is also needed to explore the idea of multiple ground-truth archi-

tectures for a given system. One possible direction is to do ground-truth extraction with different groups of engineers on the same system. Another direction would be to have system engineers develop "ground-truth" architectures starting from automatically recovered architectures. The ground-truth architecture is an important input into this kind of evaluation and deserves greater examination.

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

References

[1] *8th International Workshop on Program Comprehension (IWPC 2000), 10-11 June 2000, Limerick, Ireland*. IEEE CS Press, 2000.

[2] TIOBE Index for April 2014. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 2014.

[3] Periklis Andritsos, Panayiotis Tsaparas, Renée J Miller, and Kenneth C Sevcik. LIMBO: Scalable Clustering of Categorical Data. In *Adv. Database Technol. - EDBT 2004*, pages 531–532. 2004.

[4] Periklis Andritsos and Vassilios Tzerpos. Information-Theoretic Software Clustering. *IEEE Trans. on Softw. Eng.*, 31(2), February 2005.

[5] Mahir Arzoky, Stephen Swift, Allan Tucker, and James Cain. Munch: An Efficient Modularisation Strategy to Assess the Degree of Refactoring on Sequential Source Code Checkings. In *Proc. ICST Workshops*, pages 422–429, 2011.

[6] David F Bacon and Peter F Sweeney. Fast static analysis of c++ virtual function calls. *ACM Sigplan Notices*, 31(10):324–341, 1996.

[7] Adam Barth, Collin Jackson, Charles Reis, and The Google Chrome Team. The Security Architecture of the Chromium Browser. Technical report, 2008.

[8] Dirk Beyer. Relational Programming with CrocoPat. In *Proc. ICSE*, pages 807–810, Shanghai, China, 2006. IEEE.

[9] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent Dirichlet Allocation. *J. Mach. Learn. Res.*, 3:993–1022, March 2003.

[10] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux As a Case Study: Its Extracted Software Architecture. In *Proc. ICSE*, pages 555–563, New York, NY, USA, 1999. ACM.

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

[12] Jill Butler, William Lidwell, and Kritina Holden. *Universal Principles of Design*. Rockport Publishers, Gloucester, MA, 2nd edition, 2010.

[13] Yuanfang Cai, Hanfei Wang, Sunny Wong, and Linzhang Wang. Leveraging Design Rules to Improve Software Architecture Recovery. In *Proc. QoSA*, QoSA '13, pages 133–142, New York, NY, USA, 2013. ACM.

[14] Chris Chedgey, Paul Hickey, Paul O'Reilly, and Ross McNamara. *Structure101*.

[15] Alistair Cockburn and Jim Highsmith. Agile Software Development, the People Factor. *Computer*, 34(11):131–133, 2001.

[16] Anna Corazza, Sergio Di Martino, Valerio Maggio, and Giuseppe Scanniello. Investigating the use of lexical information for software system clustering. In *Proc. CSMR*, pages 35–44. IEEE, 2011.

[17] Lakshitha De Silva and Dharini Balasubramaniam. Controlling Software Architecture Erosion: A Survey. *Journal of Systems and Software*, 85(1):132–151, 2012.

[18] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP95Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, pages 77–101. Springer, 1995.

[19] Lei Ding and Nenad Medvidovic. Focus: A Light-Weight, Incremental Approach to Software Architecture Recovery and Evolution. In *Proc. WICSA*, pages 191–200, Washington, DC, USA, 2001. IEEE CS Press.

[20] Stéphane Ducasse and Damien Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Trans. Software Eng.*, 35(4):573–591, 2009.

[21] Stephane Ducasse and Damien Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Trans. Softw. Eng.*, 35(4):573–591, July 2009.

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

[23] Roberto Fiutem, Giulio Antoniol, Paolo Tonella, and Ettore Merlo. ART: an Architectural Reverse Engineering Environment. *Journal of Software Maintenance*, 11(5):339–364, 1999.

[24] Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[25] Joshua Garcia, Igor Ivkovic, and Nenad Medvidovic. A comparative analysis of software architecture recovery techniques. In *Proc. ASE*, pages 486–496. IEEE, 2013.

[26] Joshua Garcia, Ivo Krka, Chris Mattmann, and Nenad Medvidovic. Obtaining Ground-truth Software Architectures. In *Proc. ICSE*, ICSE '13, pages 901–910, Piscataway, NJ, USA, 2013. IEEE Press.

[27] Joshua Garcia, Daniel Popescu, Chris Mattmann, Nenad Medvidovic, and Yuanfang Cai. Enhancing Architectural Recovery Using Concerns. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *ASE*, pages 552–555, 2011.

[28] Alan Grosskurth and Michael W. Godfrey. A Case Study in Architectural Analysis: The Evolution of the Modern Web Browser, 2007.

[29] Halûk Gümüşkaya. Core Issues Affecting Software Architecture in Enterprise Projects. *Proceedings of the Enformatika*, 9:32–37, 2005.

[30] Anton Jansen, Paris Avgeriou, and Jan Salvador van der Ven. Enriching Software Architecture Documentation. *Journal of Systems and Software*, 82(8):1232–1248, 2009.

[31] MT Kirsch, VA Regenie, ML Aguilar, O Gonzalez, M Bay, ML Davis, CH Null, RC Scully, and RA 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.

[32] Kenichi Kobayashi, Manabu Kamimura, Koki Kato, Keisuke Yano, and Akihiko Matsuo. Feature-gathering Dependency-based Software Clustering using Dedication and Modularity. *Proc. ICSM*, 0:462–471, 2012.

[33] Martin Konôpka and Mária Bieliková. Software developer activity as a source for identifying hidden source code dependencies. In *SOFSEM 2015: Theory and Practice of Computer Science*, pages 449–462. Springer, 2015.

[34] Phil Koopman. A case study of toyota unintended acceleration and software safety. *Presentation. Sept*, 2014.

[35] Rainer Koschke. Architecture Reconstruction. In *Proc. ISSSE*, pages 140–173, 2008.

[36] Duc Le, Pooyan Behnamghader, Joshua Garcia, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. An Empirical Study of Architectural Change in Open-Source Software Systems. In *Technical Report USC-CSSE-2014-509, Center for Systems and Software Engineering, University of Southern California*, 2014.

[37] Timothy Lethbridge and Nicolas Anquetil. Comparative Study of Clustering Algorithms and Abstract Representations for Software Remodularization. *IEE Proceedings - Software*, 150(3):185–201, 2003.

[38] Peng Liang and Paris Avgeriou. Tools and Technologies for Architecture Knowledge Management. In *Software Architecture Knowledge Management*, pages 91–111. Springer, 2009.

[39] Thibaud Lutellier, Devin Chollak, Joshua Garcia, Lin Tan, Derek Rayside, Nenad Medvidovic, and Robert Kroeger. Comparing software architecture recovery techniques using accurate dependencies. *Proc. ICSE (SEIP)*, 2015.

[40] Ali Safari Mamaghani and Mohammad Reza Meybodi. Clustering of Software Systems Using New Hybrid Algorithms. In *Proc. CIT*, pages 20–25, 2009.

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

[42] Spiros Mancoridis, Brian S. Mitchell, Yih-Farn Chen, and Emden R. Gansner. Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In *Proc. ICSM*, 1999.

[43] Onaiza Maqbool and Haroon Babri. Hierarchical Clustering for Software Architecture Recovery. *IEEE Trans. Softw. Eng.*, 33(11):759–780, November 2007.

[44] Onaiza Maqbool and Haroon Atique Babri. The Weighted Combined Algorithm: A Linkage Algorithm for Software Clustering. In *Proc. CSMR*, pages 15–24. IEEE CS Press, 2004.

[45] Matthew McCormick, Xiaoxiao Liu, Julien Jomier, Charles Marion, and Luis Ibanez. ITK: Enabling Reproducible Research and Open Science. *Front Neuroinform*, 8(13), 02 2014.

[46] Nenad Medvidovic. ADLs and Dynamic Architecture Changes. In *Proc. ISAW*, ISAW '96, pages 24–27, New York, NY, USA, 1996. ACM.

[47] Brian S. Mitchell. A Heuristic Approach to Solving the Software Clustering Problem. In *Proc. ICSM*, pages 285–288. IEEE CS Press, 2003.

[48] Brian S. Mitchell and Spiros Mancoridis. Comparing the Decompositions Produced by Software Clustering Algorithms Using Similarity Measurements. In *ICSM*, pages 744–753, 2001.

[49] Brian S. Mitchell and Spiros Mancoridis. On the Automatic Modularization of Software Systems Using the Bunch Tool. *IEEE Trans. Softw. Eng.*, 32(3):193–208, March 2006.

[50] J. David Morgenthaler, Misha Gridnev, Raluca Sauciuc, and Sanjay Bhansali. Searching for Build Debt: Experiences Managing Technical Debt at Google. In *Proc. MTD*, pages 1–6, 2012.

[51] Gail C Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. *ACM SIGSOFT Software Engineering Notes*, 20(4):18–28, 1995.

[52] John Noll, Sarah Beecham, and Ita Richardson. Global Software Development and Collaboration: Barriers and Solutions. *ACM Inroads*, 1(3):66–78, 2010.

[53] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based Runtime Software Evolution. In *Proc. ICSE*, pages 177–186, Washington, DC, USA, 1998. IEEE CS Press.

[54] Derek Rayside and Kostas Kontogiannis. Extracting Java Library Subsets for Deployment on Embedded Systems. *Sci. Comput. Program.*, 45(2-3):245–270, November 2002.

[55] Derek Rayside, Steve Reuss, Erik Hedges, and Kostas Kontogiannis. The Effect of Call Graph Construction Algorithms for Object-Oriented Programs on Automatic Clustering. In *Proc. IWPC* [1], pages 191–200.

[56] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. *Practical virtual method call resolution for Java*, volume 35. ACM, 2000.

[57] Ricardo Terra, Marco Tulio Valente, Krzysztof Czarnecki, and Roberto S Bigonha. Recommending Refactorings to Reverse Software Architecture Erosion. In *Proc. CSMR*, pages 335–340. IEEE, 2012.

[58] Paolo Tonella, Roberto Fiutem, Giuliano Antoniol, and Ettore Merlo. Augmenting Pattern-Based Architectural Recovery with Flow Analysis: Mosaic -A Case Study. In *Proc. WCRE*, pages 198–207, 1996.

[59] Vassilios Tzerpos and R. C. Holt. ACDC : An Algorithm for Comprehension-Driven Clustering. In *Proc. WCRE*, pages 258–267. IEEE, 2000.

[60] Vassilios Tzerpos and Richard C. Holt. MoJo: A Distance Metric for Software Clusterings. In *Proc. WCRE*, pages 187–193, 1999.

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

[62] Pei Wang, Jinqiu Yang, Lin Tan, Robert Kroeger, and David Morgenthaler. Generating Precise Dependencies For Large Software. In *Proc. MTD*, pages 47–50, May 2013.

[63] Zhihua Wen and Vassilios Tzerpos. An Optimal Algorithm for MoJo Distance. volume 0, page 227, Los Alamitos, CA, USA, 2003. IEEE CS Press.

[64] Zhihua Wen and Vassilios Tzerpos. An Effectiveness Measure for Software Clustering Algorithms. In *Proc. IWPC*, pages 194–203, 2004.

[65] Zhihua Wen and Vassilios Tzerpos. Evaluating Similarity Measures for Software Decompositions. In *Proc. ICSM*, pages 368–377. IEEE CS Press, 2004.

[66] Jingwei Wu, Ahmed E. Hassan, and Richard C. Holt. Comparison of Clustering Algorithms in the Context of Software Evolution. In *Proc. ICSM*, pages 525–535, 2005.

[67] Chenchen Xiao and Vassilios Tzerpos. Software Clustering Based on Dynamic Dependencies. In *Proc. CSMR*, CSMR '05, pages 124–133, Washington, DC, USA, 2005. IEEE CS Press.

[68] Lu Xiao, Yuanfang Cai, and Rick Kazman. Design Rule Spaces: a New Form of Architecture Insight. In *Proc. ICSE*, pages 967–977, 2014.