IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 19, 2018, accepted August 27, 2018, date of publication September 17, 2018, date of current version October 12, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2870118

Directory-Based Dependency Processing for Software Architecture Recovery

XIANGLONG KONG[1], BIXIN LI[1], LULU WANG[1], AND WENSHENG WU[2]
[1]School of Computer Science and Engineering, Southeast University, Nanjing 211189, China
[2]Huawei Technologies Co., Ltd., Shenzhen 518129, China

Corresponding author: Bixin Li (bx.li@seu.edu.cn)

This work was supported in part by the National Key R&D Program of China under Grant 2018YFB1003901, in part by the National Natural Science Foundation of China under Grant 61572126, Grant 61872078, and Grant 61402103, and in part by the Cooperation Project with Huawei Technologies Co., Ltd., under Grant YBN2016020009.

ABSTRACT Directory structure contains a wealth of software design information; it is used to transfer thoughts of architects to developers. Information extracted from directory paths should play an important role in architecture recovery techniques, but it has been proved that modules or components directly represented by directories are not accurate due to the inconsistency between stages of development and design. To make better use of information extracted from directories, we propose a directory-based dependency processing technique to utilize the information of directories in the process of structure-based architecture recovery. The technique groups the selected inter-coupling files and intra-coupling files in the same directory into a submodule and generates submodule-level dependency graph based on file-level dependency graph. We apply both manual and automatic architecture recovery techniques on submodule-level dependency graph, and the results show that our technique can greatly improve the efficiency and effectiveness of manual and automatic architecture recovery techniques; the technique can also make other structure-based recovery techniques easily scalable to large-sized projects.

INDEX TERMS Software architecture recovery, dependency graph, directory path.

I. INTRODUCTION

Software architecture plays an extremely important role in modern software development. A well-documented software architecture can greatly improve the effectiveness and efficiency of program comprehension and software maintenance [33]. However, it is pretty hard for the developers to keep architecture documented out-to-date during software life cycle, and architecture recovery is also a tedious and costly task in both academia and industry. For example, recovery of the ground-truth architecture of Google Chromium takes the researchers two years of efforts with the cooperation of related developers [10].

For this reason, a huge body of research efforts have been dedicated to software architecture recovery, which aim to recover architecture based on some related information (e.g., code dependencies and functionality of modules). According to the source of input information, these techniques can be divided into two categories, structure-based techniques and knowledge-based techniques [40]. Structure-based techniques reply on software codes to extract data-flow-based or control-flow-based dependencies between them, and then identify components by utilizing some clustering techniques. Knowledge-based techniques recover architecture by clustering software entities which implement similar or related functionalities. The identification of functionalities replies on some textual information (e.g., knowledge of domain experts, design documents and comments). Although knowledge-based techniques have been shown to perform good on some subjects [6], [11], the majority of the existing recovery approaches are still structure-based techniques due to the difficulty in extraction of textual information for knowledge-based techniques [1], [23]–[25], [40], [41].

There are also some empirical studies on current structure-based recovery techniques [2], [9], [16], [20], [26], [44], but the results from these studies are commonly different from each other, none of the current recovery techniques always perform better than others. The major reason of this inconsistency is that these studies used different subjects, metrics and implementations of techniques in the comparisons [22]. The deep-seated contradiction behind this reason is that most of the studied techniques obtained information only from dependencies of different-level code

entities, which belongs to implementation view of programs [18]. However, software architecture belongs to logical view, there may be losing important logical knowledge while translating information from implementation view to logical view. So, can we put some available logical knowledge into the process of structure-based techniques? The most widely-recognized logical information in software is the design of directories, directories are used to keep the initial design by giving a framework to developers and maintainers, they are already considered as feathers of clustering techniques in several studies [3], [5], [14], [44], [45]. However, Lutellier *et al.* [21], [22] proved that directory-based clustering techniques performed worse than file-based clustering techniques in terms of accuracy, and directories generally could not represent the components in software systems [10]. It is a pity that directories contain a wealth of design information of architecture, but architecture recovered directly from directories is not accurate since developer may not always code in compliance with the design of directories.

In pursuit of a more accurate and efficient architecture recovery approach, we utilize some collected information of directories in the process of the current structure-based recovery techniques. The directory-based dependency processing technique is designed to generate a submodule-level dependency graph based on code dependencies and directories, the graph can be used as input of structure-based recovery techniques. We define submodule as a set of coherent files within a module. A module is usually composed of one or more submodules. Although directories could not represent components or modules correctly, we still expect that parts of files inside the directories could represent submodules in a software system. We try to split one directory into several submodules based on dependency relationships between files inside and outside the directory. If a source code file has strong dependency relationships with files from other directories, we think that this file has a high probability to indicate some specific feature of the software and mark this kind of files as inter-coupling files. For each inter-coupling file, we search for the files that have dependency relationships with it in the same directory, and mark this kind of related files as intra-coupling files. Submodules are generated by grouping each inter-coupling file with its related intra-coupling files. After the selection, the remaining files in each directory are also grouped as a submodule. The technique is built on an assumption that a set of files with high cohesion in a directory should usually be used to implement similar or related functionalities. The submodule-level dependency graph is generated by clustering each inter-coupling file and related intra-coupling files in the file-level dependency graph. And the submodule-level dependency graph is designed to be used as the input of structure-based recovery techniques, it can take the place of file-level dependency graph. The inter-coupling files are determined according to threshold values of dependencies (i.e., inter-indegree and inter-outdegree of the nodes in file-level dependency graph). When selecting the inter-coupling files, we only consider the dependencies between different directories. And the intra-coupling files are determined only according to the dependencies within the same directory. Considering extreme cases, when threshold values are over-top, there will be no inter-coupling files in file-level dependency graph, so each directory represents a submodule. And if threshold values are 0, all the source code files are marked as inter-coupling files, each submodule contain only one file in this situation. Our approach obtains file-level dependencies based on the extracted symbol dependencies, and further groups the files into submodules based on information of directories. The objective of our approach is to find a proper way to make architecture recovery techniques more accurate and efficient by utilizing the information from both directories and source code files.

To evaluate the directory-based dependency processing technique, We conduct both automatic and manual structure-based recovery attempts by utilizing submodule-level dependency graph. The evaluation results show that our technique can greatly improve the efficiency of manual recovery; submodule-based automatic recovery techniques perform better than file-based automatic recovery techniques in terms of accuracy; our approach can greatly improve the efficiency of recovery attempts when applying on large-sized subjects, make other structure-based recovery techniques easy to scale to large-sized projects.

In summary, our paper makes the following novel contributions:

- We propose the directory-based dependency processing technique, which brings the gap of the performance of files and directories used in architecture recovery techniques.
- We evaluate the directory-based dependency processing technique by conducting the architecture recovery attempts on submodule-level and file-level dependency graphs. The results show that our approach can generally improve accuracy and efficiency of the used recovery techniques.
- We propose a method to make existing structure-based techniques scalable to large-sized projects by utilizing submodule-level dependency graph instead of file-level dependency graph.
- We conduct a study on the impacts of threshold values on the submodule partitions to find out a suitable setup of threshold values used in the experiments.

The rest of this paper is organized as follows: Section II provide a case study to further demonstrate our motivations and insights. Section III describes the detailed steps of our technique. Section IV describes the experimental setup and the results analysis to present several meaningful observations. Section V presents an overview of related research in the area of software architecture recovery and Section VI concludes this paper.

II. CASE STUDY

In this section, we discuss a case study involving generation of submodule-level dependency graph by using

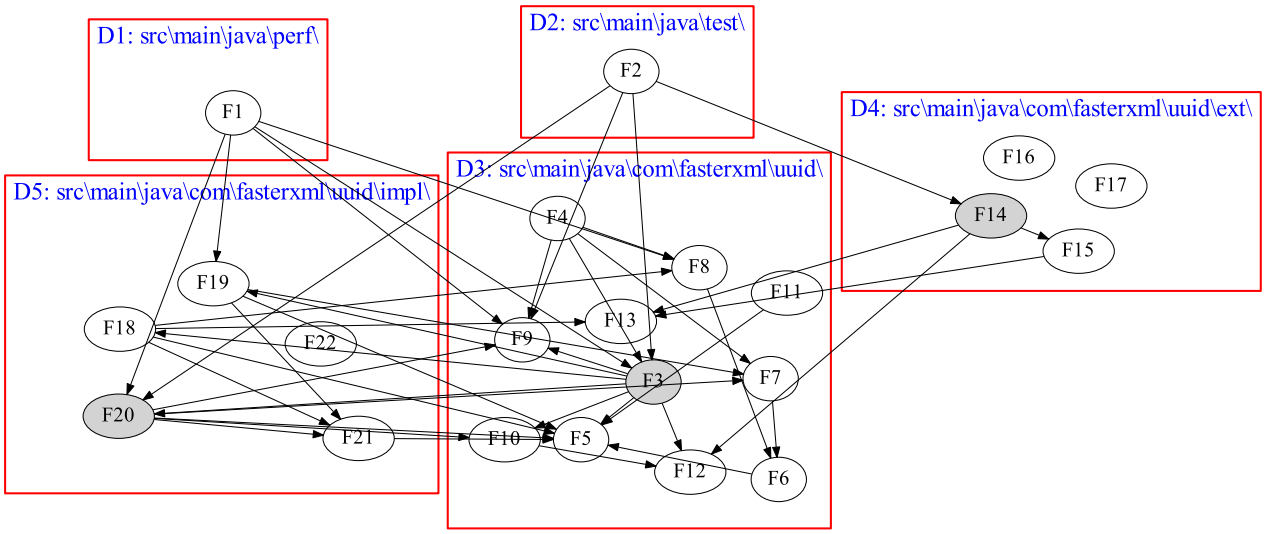


FIGURE 1. File-level dependency graph of project JUG.

directory-based dependency processing technique. We select a small-sized project, i.e., Java UUID Generator (JUG)[1] in our case study. JUG is a set of Java classes for working with UUIDs: generating UUIDs using any of standard methods, outputting efficiently, sorting and so on. The JUG project contains 22 Java source code files, and its file-level dependency graph is presented in Figure 1. The file-level dependency graph is generated based on Eclipse JDT,[2] we will show the details of extraction in section III. In Figure 1, each oval node represents a Java source code file, the number of the node refer to the number of the source code files, which is listed in Table 1. For example, F1 refers to file 1 and D1 means directory 1. The edge between nodes represents the set of symbol (functions, global variables, etc.) dependencies between two files. The number of symbol dependencies is used to check inter-coupling files. To make the figure clear, we have not shown the number beside the edges. The five red boxes in the figure represent the five directories in the project, each directory contains one or several files. The three gray nodes are the selected inter-coupling files. From the figure, we have the following observations.

First, although there are only 22 files in the project, the file-level dependency graph is still too complex to understand easily. Second, if we only consider the file-level dependencies, regardless of the directory structure, the recovery results may not be correct. For example, F3 and F5 in D3 have the largest number of dependencies in the graph. The four files in D5 all depend on F5, and they also have strong dependency relationships with F3. However, some files in D3 do not have direct dependency relationships with F3 and F5, such as F7 and F3. These conditions will make structure-based recovery techniques split the files in D3, and group parts of them with the files in D5. In fact, most of the files in D3 are

[1]JUG Project, https://github.com/cowtowncoder/java-uuid-generator
[2]Java Development Tools, http://www.eclipse.org/jdt

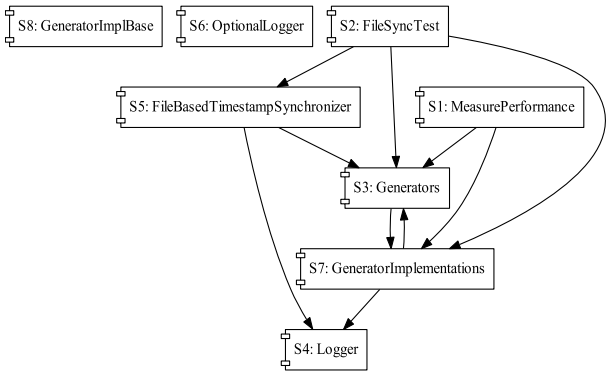


FIGURE 2. Submodule-level dependency graph of JUG.

JUG API classes, and the files in D5 are various UUID generator implementations. So they should not be grouped together in the architecture. Third, if we use directories to represent components, the recovery results are also incorrect. For example, there are some files which do not have dependency relationships with other files in the same directory, such as F16 and F17 in D4, F22 in D5. And F13 in D3 has strong dependency relationships with the files in D4. Actually, F22 is a meaningless file, it contains no variables or functions. F16 and F17 implement similar functionality with F13, they should be grouped together. In summary, we find that we could not obtain good enough information for architecture recovery regardless of directory structure or directly using directory to represent components.

To make better use of information extracted from directories, we apply directory-based dependency processing technique on the file-level dependency graph in Figure 1, the obtained submodule-level dependency graph is presented in Figure 2. We also present the detailed classification and affiliation relationships of the studied files in Table 1.

TABLE 1. List of source code files in project JUG.

Number	File Name	Classification	Belonging to Directory	Belonging to Submodule
F1	MeasurePerformance.java	Remaining File	D1	S1
F2	FileSyncTest.java	Remaining File	D2	S2
F3	Generators.java	Inter-coupling File	D3	S3
F4	Jug.java	Intra-coupling to File 3	D3	S3
F5	UUIDType.java	Intra-coupling to File 3	D3	S3
F6	UUIDGenerator.java	Intra-coupling to File 3	D3	S3
F7	NoArgGenerator.java	Intra-coupling to File 3	D3	S3
F8	StringArgGenerator.java	Intra-coupling to File 3	D3	S3
F9	EthernetAddress.java	Intra-coupling to File 3	D3	S3
F10	UUIDTimer.java	Intra-coupling to File 3	D3	S3
F11	UUIDComparator.java	Intra-coupling to File 3	D3	S3
F12	TimestampSynchronizer.java	Intra-coupling to File 3	D3	S3
F13	Logger.java	Remaining File	D3	S4
F14	FileBasedTimestampSynchronizer.java	Inter-coupling File	D4	S5
F15	LockedFile.java	Intra-coupling to File 14	D4	S5
F16	JavaUtilLogger.java	Remaining File	D4	S6
F17	Log4jLogger.java	Remaining File	D4	S6
F18	NameBasedGenerator.java	Intra-coupling to File 20	D5	S7
F19	RandomBasedGenerator.java	Intra-coupling to File 20	D5	S7
F20	TimeBasedGenerator.java	Inter-coupling File	D5	S7
F21	UUIDUtil.java	Intra-coupling to File 20	D5	S7
F22	GeneratorImplBase.java	Remaining File	D5	S8

In the table, column ''Number'' refers to the number of files in Figure 1, column ''File Name'' presents the name of studied files. Column ''Classification'' presents the role of the files in the process of our approach. Columns ''Belonging to Directory'' and ''Belonging to Submodule'' present the directories and submodules that contain the files. The first step of our technique is selection of inter-coupling files, i.e., source code files that have strong dependency relationships with files from other directories. While selecting inter-coupling files, we only consider the number of symbol dependencies between different directories. In this study, the threshold values of inter-indegree and inter-outdegree of inter-coupling files are set as (3, 1). The selected inter-coupling files are shown as gray node in Figure 1. For example, F14 in D4 has three symbol dependencies with F2 in D2 (the edge between nodes in Figure 1 just represents the whole set of symbol dependencies), and it also has four symbol dependencies with F13 and F12. So its inter-outdegree is 3 and inter-outdegree is 4, F14 is marked as inter-coupling file. After selecting inter-coupling files, we group them with their related intra-coupling as a submodule and group the other remaining files in the directory also as a submodule. Then the submodule-level dependency graph in Figure 2 is generated.

Between Figure 2 and Figure 1, we can find that S1 and S2 are actually equivalent to D1 and D2 because they only contain one file. S3 is grouped by inter-coupling file F3 and its related intra-coupling files, S4 only contains F13. Submodules S3 and S4, S5 and S6, S7 and S8 are composed of files in directory D3, D4, D5, respectively. For directory D3, most of the APIs are grouped as S3 and the logging API is separated as S4 because it has much stronger dependency relationships with files outside the directory. For directory D4, file-based UUID generator API is grouped as S5 and optional logging APIs are grouped as S6. Submodule S6 has none dependency relationships in the graph, but it should be grouped with S4 since they implement similar functionalities. For directory D5, the various UUID generator implementations are grouped as S7 and the meaningless file F22 is separated as S8.

Overall, the submodule-level partition of the project in Figure 2 is better for architecture recovery than the file-level partition and directory-level partition in Figure 1. First, the submodule-level partition is much easier to understand than file-level partition since the dependency relationships in submodule-level partition are much simpler and we can easily find out the functionality of each submodule. Files contained in submodules are usually tight-coupled, and the irrelevant files are separated. Second, the submodule-level partition can help to recover more accurate architecture than the file-level partition and directory-level partition. The directory-level partition is too rough to represent modules or components of architecture. For example, the directory-level partition cannot identify the meaningless file F22 in directory D5. Submodule S5 and S6 in directory D4 should not be grouped together, because they are designed to implement irrelevant functionalities and they do not have dependency relationships. For the file-level partition, the files in submodule S5 (APIs) and S7 (Implementations) are tight-coupled, the structure-based

architecture recovery techniques are hard to distinguish the Implementation Class (S7) and API Class (S5). The most likely result is that the recovery technique groups parts of files in D3 with files in D5 as a component and group remaining files in D3 into several components.

In this case study, we want to prove that submodule-level dependency graph can perform better than file-level dependency graph and directory-level dependency graph in software architecture recovery. And we will further present how the directory-based dependency processing technique impacts the recovered architecture in Section IV.

III. APPROACH

In this section, we explain details of the directory-based dependency processing technique. The technique comprises two main steps, obtaining dependencies from source code files and dependency processing by applying information of directory .

A. OBTAINING DEPENDENCIES

Include dependencies and symbol dependencies are frequently used in software architecture recovery techniques. They can both represent relationships between code files. Although extraction of include dependencies is much more efficient, we extract symbol dependencies instead of include dependencies from source code files in our study because include dependencies are not always accurate, e.g., *a.c* may declare it includes *b.h*, but may not use any variables or functions of *b.h*. Lutellier *et al.* [22] also approved that using symbol dependencies can recover more accurate software architectures than include dependencies.

The step of obtaining dependencies in our study is implemented based on Eclipse CDT[3] and Eclipse JDT.[4] We extract symbol dependencies by analyzing the AST (i.e., Abstract Syntax Tree), which is generated by program parser in Eclipse CDT/JDT. The types of dependency relationships which we consider in our technique are listed in Table 2. In the table, column ''Dependency Type'' presents the type of dependency relationship considered in the technique, and ''√'' means we extract that type of dependency for the project coding in C/C++ or Java language. The selection of dependency types is according to the specifications of Eclipse CDT/JDT ASTParser. We extract all the dependency relationships from the AST based on the parser of Eclipse CDT/JDT. For example, the ''Inheritance'' dependency can be extracted by analyzing the attribute *TypeDeclaration.getSuperclass* of the nodes in AST, and the ''Call'' dependency can be extracted by analyzing the attribute *MethodInvocation*.

We only consider direct dependencies between symbols, because direct dependencies are proved to perform better than transitive dependencies on the ability of architecture recovery [22]. We present symbol dependencies on file level, once variables or functions in different files have dependency relationship, an directed edge would be added between these two files. For each pair of linked files, we merge the edges with same direction into one edge, and the number of symbol dependencies in this edge would be the weight of the edge. Finally, we get a file-level dependency graph, each node in the graph represents a source code file and each edge could represent one or several symbol dependencies.

[3]C/C++ Development Tools, https://www.eclipse.org/cdt

[4]Java Development Tools, http://www.eclipse.org/jdt

TABLE 2. Extracted dependency relationships.

Dependency Type	C	C++	Java
Inheritance		√	√
Implementation			√
Composition	√	√	√
Call	√	√	√
Use	√	√	√
Reference	√	√	
Write	√	√	√
Read	√	√	√
Instantiation		√	√
Import			√
Definition	√	√	√
Declaration	√	√	√

B. DIRECTORY-BASED DEPENDENCY PROCESSING

Directories contain a wealth of design information of architecture, but current clustering techniques do not have an appropriate way to utilize the information of directory. In order to recover accurate architecture, we apply directory information on file-level dependency graph. To achieve this, we perform the following steps:

- We extract all the directory paths of a software system, mark each source code file with the tag of directory contains it. We treat all the directories fairly in the dependency graph, even though there may be parents-directories and sub-directories. The tags of directories could divide the whole software system into the initial partitions. In this way, each node in the dependency graph represents a code file, the node has a tag of the directory contains it, and each edge has a weight as the number of symbol directories between the nodes.
- For each directory, we check whether there exist inter-coupling files, i.e., the file with a big number of symbol dependencies with the files outside its directory. The inter-coupling files are selected according to the threshold values of dependencies. After the selection of inter-coupling files, we start to search for the intra-coupling files, i.e., the files which have dependency relationships with inter-coupling file in the same directory.
- For each inter-coupling file, we group it with related intra-coupling files into a new submodule. When an intra-coupling file have dependency relationships with two or more inter-coupling files, the weights between the intra-coupling file and the inter-coupling files are used to determine which one it should belong to.

- After grouping inter-coupling files, the other files remained in the original directory are grouped to be a submodule. So all the files are grouped into submodules, and the dependencies between files are still retained by the submodules. The submodule-level dependency graph is generated based on file-level dependency graph and the information of directories.

Actually, there are two different ways to generate submodules based on the inter-coupling files. The first one method is grouping the inter-coupling file with related intra-coupling files separately, i.e., a submodule contains only one inter-coupling file. The other grouping method is grouping all the connected inter-coupling files and intra-coupling files. We select the first grouping method in our technique because it makes our approach partial to file-level entities, and the other method makes the submodules close to directory-level entities. Lutellier *et al.* [22] have proved that directory-based recovery techniques perform worse than file-based recovery techniques in terms of accuracy. To improve the understandability of submodule-level dependency graph, we name the submodule as the name of the largest file it contains. When there is no inter-coupling file in submodule, i.e., the submodule is grouped by the remaining files after the selection of inter-coupling file, the submodule is named as the name of the directory contains it.

An example for directory-based dependency processing technique is shown in Figure 3. In the above part of the figure, we lay out a file-level dependency graph which contains 16 source code files. These files are arranged in two directories, each directory has 8 files. We simply setup the threshold values of dependencies in this example, i.e., both inter-indegree and inter-outdegree are set as 1. When selecting the inter-coupling files, we only consider the dependency relationships between different directories. So we can easily find out that *a.c*, *c.c* and *z.c* are inter-coupling files which are marked with * in the figure.

For each inter-coupling file in Figure 3, we start to select related intra-coupling files according to our approach. Among the files, *b.h* is the intra-coupling file of both *a.c* and *c.c*. To determine the affiliation, we check the weights, the weight of edge between *a.c* and *b.h* is 3 and the weight of edge between *c.c* and *b.h* is 1. So *b.h* should be grouped with *a.c*. And we can split Directory A into two submodules as shown in part (b) of Figure 3. For Directory B, *z.c* is the only inter-coupling file and *z.h* is the related intra-coupling file, we group them as a new submodule. The remaining 6 files are grouped as a submodule. Finally, we get the submodule-level dependency graph which contains 4 submodules. This kind of dependency graph can be used as input of structure-based architecture recovery techniques.

C. THRESHOLD VALUES

Our approach used two threshold values to determine inter-coupling files, i.e., inter-indegree and inter-outdegree. The inter-indegree refers to the number of symbols that are outside the directory and depend this file, inter-outdegree means the number of symbols that are outside this directory and depended by this file. The threshold values are presented as (inter-indegree, inter-outdegree) in the paper. The assumption behind the threshold values is that the selected inter-coupling files have a high probability to indicate some specific feature, they may be interfaces or used to implement some specific functionality. These inter-coupling files could offer the indications of a submodule, so we tried to identify them. The setup of the threshold values directly determine the level of abstraction of the submodule-level dependency graph. Considering extreme cases, when threshold values are over-top, each directory represents a submodule, and if threshold values are 0, each code file represents a submodule. We have conducted a study on the impacts of threshold values in the Section IV, and the results have shown that (3, 1) is the suitable values for the studied subjects.

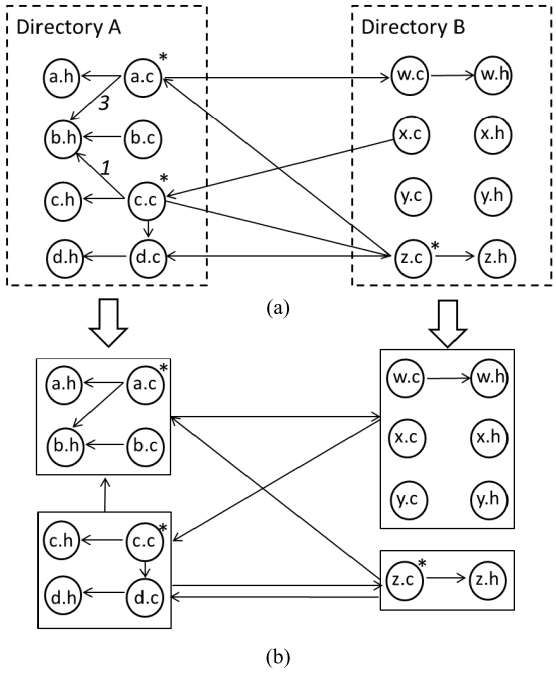


FIGURE 3. An example for Directory-based dependency processing. (a) File-leval dependency graph. (b) Submodel-leval dependency graph.

IV. EXPERIMENTS AND RESULTS

The directory-based dependency processing technique can generate submodule-level dependency graph, which can be directly used in automatic structure-based architecture recovery techniques and manual recovery attempts. In this work, we aim to answer the following research questions:

- **RQ1:** How does the directory-based dependency processing technique perform in tool-assisted manual architecture recovery?
- **RQ2:** How does the directory-based dependency processing technique perform in automatic architecture recovery?

TABLE 3. Subject systems statistics.

Subjects	Language	Version	NLOC	Description
Bash	C	1.14.4	45K	UNIX Shell
Httpd	C	2.4.33	195K	Apache HTTP Server
OODT	Java	0.2	81K	Data Middle-ware
Joda-time	Java	2.9.9	85K	Date and Time Library
Hadoop	Java	0.19.0	224K	Data Processing
Archstudio	Java	4	236K	Architecture IDE

- **RQ3:** How does the directory-based dependency processing technique perform with different setup of threshold values?

A. EXPERIMENTAL SETUP

1) SUBJECT PROJECTS

To investigate the three research questions, we select all the four subject systems from the existing study [10] (i.e., Bash-1.14.4, OODT-0.2, Hadoop-0.19.0 and Archstudio-4), and another two projects from Github repository [12], [13] (i.e., Httpd-2.4.33 and Joda-time-2.9.9). Garcia *et al.* [10] recovered ground-truth architectures of the four projects in their work, which could help to measure quality of the recovered architectures in our experiments. The other two subjects are selected randomly, we are not familiar with the related source code and documents before the selection. We plan to recover ground-truth architectures of the two subjects manually based on submodule-level dependency graphs.

Table 3 presents the detailed subject systems statistics. In the table, column "Subjects" lists all the subject systems that we used. Column "Language" presents the major programming language coding in the subjects, column "Version" lists the selected release version of the subjects. For Bash, OODT, Hadoop and Archstudio, we select the same version as used in the work [10]. For Httpd and Joda-time, we select the latest version in their github repositories [12], [13]. Column "NLOC" presents the net lines of code, i.e., lines of code that are non-blank and non-comment, which are calculated by tool CLOC [35] in the experiments. Finally, column "Description" presents the basic functionality of each project.

2) SELECTED RECOVERY TECHNIQUES

The submodule-level dependency graph can be used in both tool-assisted manual architecture recovery and structure-based automatic architecture recovery. For tool-assisted manual architecture recovery, we implement a similar approach to the existing work [10]. Our approach contains four steps. First, we obtain the submodule-level dependency graph of the selected subjects based on the directory-based dependency processing technique. Second, we check the partition of submodules and identify the functionality of each submodule. While checking the partition of submodules, we mainly focus on the submodule that contains more than 10 files and check whether the files are used to implement similar functionalities. Third, we iteratively group the submodules according to their functionalities and dependency relationships. Fourth, we invite architect from Huawei Technologies Co., Ltd (i.e., the fourth author of the paper) to refine the architecture and confirm the correctness of recovered architecture with us.

For structure-based automatic architecture recovery, We select Bunch [24] in our experiments because Bunch is the most widely used technique in comparison of architecture recovery studies [3], [9], [16], [29], [30]. Bunch groups code entities according to an optimization function called Modularization Quality (MQ). Bunch use two kinds of hill-climbing algorithms to resolve the optimization problem, i.e., nearest and steepest ascent hill climbing (NAHC and SAHC). We obtain executable Bunch-3.5 tool from the website of the authors [34] and applied both NAHC and SAHC on our selected subjects. For the setup of Bunch, we use the Bunch.TurboMQ clustering algorithm and collect the median level of graph as our results.

3) MEASUREMENTS

There exit two kinds of measurements for recovery techniques. One is designed to measure the similarity between recovered architecture and ground-truth architecture, and the other one is designed to measure self-quality independent of the recovered architecture. We use the most widely used measurement for each kind [22], i.e., MoJoFM [42] and Turbo MQ [27] in our experiments.

MoJoFM can be used to compare the different recovered architecture according to their similarity with the ground-truth architecture. It is defined by the following formula,

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

where A indicates recovery architecture, B indicates the ground-truth architecture, $mno(A, B)$ is the minimum number of *Move* and *Join* operations needed to transform A into B.

Turbo MQ [27], [28], extension of Basic Modularization Quality (Basic MQ) [24], supports dependency graph with edge weights. The assumption behind Turbo MQ is that architecture with high cohesion and low coupling is more acceptable. To calculate Turbo MQ, we need to calculate a Cluster Factor first. Cluster Factor of module i is defined by the following formula,

$$CF_i = \frac{\mu_i}{\mu_i + 0.5 \times \sum_j (\epsilon_{ij} + \epsilon_{ji})} \quad (2)$$

where i and j indicate the cluster in dependency graph, μ_i indicates the number of intra-relationships, and $\epsilon_{ij} + \epsilon_{ji}$ indicates the number of inter-relationships between cluster i and cluster j. Turbo MQ is defined as following,

$$Turbo\ MQ = \sum_{i=1}^{N} CF_i \quad (3)$$

where N is the number of clusters in dependency graph, CF_i is the Cluster Factor of module i. In this way, Turbo

TABLE 4. Statistics of software entities in manual recovery.

	File-level statistics		Directory-level statistics			Submodule-level statistics		
Subjects	#Files	#Dependencies	#Net-Dir	#Dependencies	Avg size	#Submodules	#Dependencies	Avg size
Httpd	462	2190	46	53	10	90	173	5
Joda-time	329	2056	15	80	21	179	1453	2

MQ can measure the quality of recovered architecture without similarity to the ground-truth architecture.

In the experiments, we first recover architecture of Httpd-2.4.33 and Joda-time-2.9.9 manually, with the assistance of submodule-level dependency graph. Then we apply Bunch (NAHC and SAHC) on file-level dependency graph and submodule-level dependency graph, and calculate MoJoFM and Turbo MQ scores of the 6 studied subjects. The file-level dependency graph is obtained through Eclipse CDT/JDT, and the submodule-level dependency graph is generated by our approach. The recovered architecture is generate by Bunch, the ground-truth architectures of Bash, OODT, Hadoop and Archstudio are obtained from existing study [10], and the other two ground-truth architectures are obtained by ourselves. We run Bunch on a Ryzen 1700 server with 16GB memory.

B. RESULTS ANALYSIS

1) RQ1: IMPROVEMENTS ON TOOL-ASSISTED MANUAL ARCHITECTURE RECOVERY

Tool-assisted manual architecture recovery techniques are commonly used to obtain ground-truth architecture [5], [10]. Since they usually involve extensive manual work, the biggest limitation of manual recovery is the huge time cost. So our approach focuses on improvement on efficiency of manual recovery. There are already many existing tools which can assist manual recovery by extracting file-level, directory/package-level dependencies automatically, e.g., Dependency Finder [36], Doxygen [37] and SonarQube [38]. But according to the results in our case study and recent work [22], the file-level dependencies are usually too complex to understand, and the directories cannot represent the modules correctly. The directory-based dependency processing technique in our experiments can generate submodule-level dependency graph which aims to represent the architecture on an appropriate level of abstraction between files and directories. There are two major factors which can impact effectiveness and efficiency of manual recovery, i.e., structure of software entities and functionality of each entity. We will explain the improvements on manual recovery in two perspectives, i.e., reduction of the number and size of entities, and the initial functional partition. These two features contribute a lot during the process of manual recovery.

First, the directory-based dependency processing technique can greatly reduce the number and size of software entities. In the experiments, there are three different level of software entities, i.e., file, directory and submodule. The statistics of these entities is shown in in Table 4. In the table, column ''Subjects'' lists the programs we selected to recover, column ''#Files'' presents the number of source code files, and column ''Dependencies'' presents the number of dependencies between specific level entities. Column ''#Net-Dir'' presents the number of directories that contain at least one file, column ''Avg size'' presents the average number of files contained by each directory or submodule. From Table 4, we find that (1) the number of submodule-level entities and dependencies is much smaller than that of file-level entities and dependencies; (2) the average size of submodules is much smaller than that of directories. The first finding means that the number of submodule-level entities is greatly reduced in contrast to file-based recovery attempts. To our experience, an entity that contains less files are usually easier to understand than the one contains more files. The second finding illustrates that our technique can help us to comprehend the partition easily. The directory-based dependency processing technique can generate less entities than file-level extractions, and smaller entities than directory-level extractions. The technique can generate an initial partition of architecture on an appropriate level of abstraction between files and directories.

Second, the submodule-level dependency graph is an initial fine-grained partition of software functionalities. Submodules are grouped according to the inter-connectivity and intra-connectivity relationships of each directory. The most important file in submodules is the inter-coupling file, which has a high possibility to be used to implement some specific functionality. The submodule-level dependency graph is expected to make the entities easy to understand in terms of functionality, and improve efficiency of program comprehension. We explain this improvement in a small case. Figure 4 presents the files in directory *modules\generators* of project Httpd-2.4.33. To make the figure clear, we have not shown inter-connectivity relationships of the files in the figure. According to our technique, *mod_status.h* is the only one inter-coupling file. So the 10 files are grouped into two submodules, one contains *mod_status.h* and *mod_status.c*, the other submodule contains the remaining 8 files. The submodules are named as *mod_status* and *mod_generator* according to our approach. We further investigate the functionalities and find that submodule *mod_status* is designed to present the current server statistics. To archive this, submodule *mod_status* needs to communicate with other modules. Submodule *mod_generator* is used to generate several kinds of textual output, e.g., http header files, cgi running environments and so on. This submodule do not need to communicate with other modules. In this way, the directory-based dependency processing technique presents the initial

TABLE 5. Time cost of tool-assisted manual architecture recovery.

Subjects	DBDP	Functionality Identification	Submodule Grouping	Confirmation	Total
Httpd	31s	8h	2h	3h	13h
Joda-time	48s	11h	4h	3h	18h

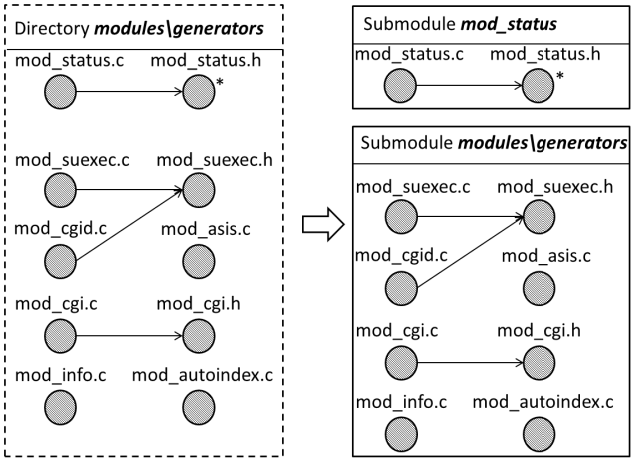


FIGURE 4. A case of submodule partition.

partition according to the functionalities of related files. If we recover the architecture based on file-level or directory-level dependencies, we may split this directory into three or four submodules, or just group these files into a whole submodule. However, we think the current partition is better, because all the files in submodule *mod_generator* are used to generate textual output, they should not be grouped into too many submodules. And considering the fact that submodule *mod_status* has strong dependency relationships with other submodules, we finally group submodule *mod_status* into component *mod_cache* in Figure 5. The initial fine-grained partition of software functionalities can help us to recover architecture correctly and improve the efficiency greatly.

The architectures we recovered manually are shown in Figure 5. The architectures of Httpd and Joda-time contain 13 and 6 components, respectively. The actual time cost of our manual recovery is listed in Table 5. In the table, column "Subjects" lists the programs we selected to recover, column "DBDP" presents the execution time of the directory-based dependency processing. Column "Functionality Identification" presents the time cost of our identification on the functionality of each submodule and the checking on the correctness of each submodule. Column "Submodule Grouping" presents the time cost that we take on grouping submodules with similar functionalities. Column "Confirmation" presents the time cost of our discussion on the confirmation of the recovered architecture. Column "Total" lists the total time cost we take for the manual architecture recovery. From the table, we can find that although the technique can greatly reduce the size and number of software entities compared with directory-based and file-based techniques, the step of functionality identification still take the majority of time. This finding illustrates that the identification of functionality is the hardest work in manual recovery. So our technique focuses on the improvements of effectiveness and efficiency of functionality identification by offering an initial partition of architecture on an appropriate level of abstraction between files and directories.

According to the data in Table 5, we spend 13 hours/18 hours on recovering ground-truth architecture of Httpd and Joda-time. The time cost of our attempts is much smaller than the existing study [10], the average cost of recovering the ground-truth architecture of seven systems by Garcia *et al*. is 107 hours. We think there can be several reasons. First, Garcia *et al*. used file-level dependencies as the major source of architectural information, which contain much more entities than submodule-level dependencies. Second, their recovered architectures is more detailed than ours. They aim to separate the functionalities independently by grouping the source code files which implement specific functionality into a component. In our manual recovery, we group the files which implement similar functionalities together. Third, the largest program in their study has 280K of NLOC, and the NLOC of our two subjects are 85K and 195K. The smaller size of subjects can make the process of recovery faster. Fourth, their recovered architecture is authenticated by certifiers, i.e., the developer or architect of the project. The email communications also take a lot of time. In our experiments, we confirm the recovered architecture through several face-to-face discussions. In summary, the results prove that our technique can greatly improve efficiency of manual recovery by reducing the number of entities and splitting the directories into several submodules with higher cohesion.

However, it cannot be denied that the huge improvement on efficiency may incur the instability of accuracy. Since the larger sized submodules have a higher possibility to contain wrong partition, we examined the submodules which have ten or more files carefully during the recovery. Project Httpd contains 12 out of 90 submodules which have ten or more files, and project Joda-time has 5 out of 179 such submodules. After the examination, 4 submodules in Httpd and 1 submodule in Joda-time are split once again. These submodules contain too many files, and the functionalities vary widely. They are grouped together due to the intra-connectivity dependencies in the same directory.

2) RQ2: IMPROVEMENTS ON AUTOMATIC ARCHITECTURE RECOVERY

We applied Bunch technique (NAHC and SAHC) on both file-level and submodule-level dependency graphs of the

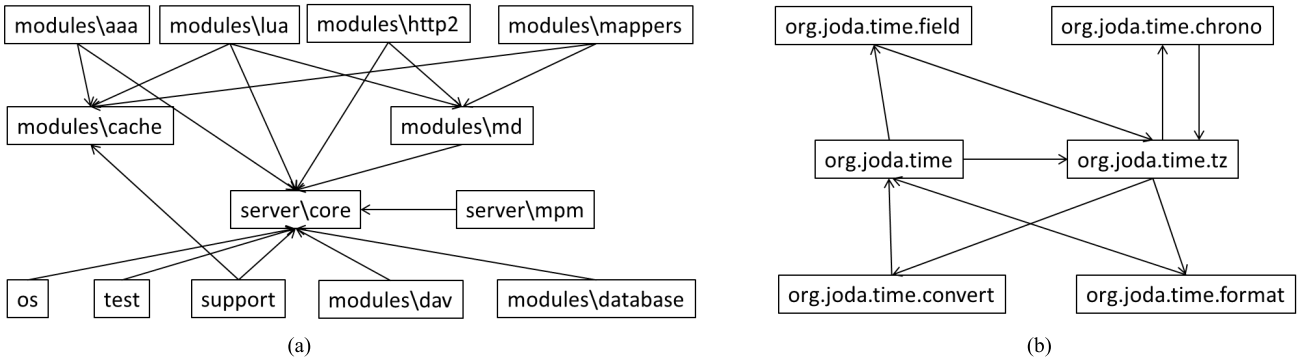


FIGURE 5. Ground-truth architectures of Httpd and Joda-time. (a) Ground-truth architectures of Httpd. (b) Ground-truth architectures of Joda-time.

TABLE 6. Results of turbo MQ scores.

Subjects	Submodule-NAHC	Submodule-SAHC	File-NAHC	File-SAHC
Bash	2.81	2.81	3.70	3.97
Httpd	9.10	8.90	8.60	8.80
OODT	49	50	40	37
Joda-time	9.72	9.70	4.75	4.70
Hadoop	16.71	19.49	15.56	16.01
Archstudio	26.07	26.63	16.88	12.13
Average	18.90	19.59	14.92	13.77

TABLE 7. Results of MoJoFM scores.

Subjects	Submodule-NAHC	Submodule-SAHC	File-NAHC	File-SAHC
Bash	24%	29%	47%	42%
Httpd	53%	51%	41%	38%
OODT	61%	69%	39%	42%
Joda-time	44%	41%	37%	39%
Hadoop	50%	49%	33%	40%
Archstudio	60%	64%	51%	39%
Average	49%	51%	41%	40%

6 studied projects. Table 6 and Table 7 present the results of Turbo MQ and MoJoFM of our recovery attempts. In the tables, NAHC and SAHC refer to nearest and steepest ascent hill climbing algorithm used in Bunch. Column ''Subjects'' presents the studied projects, the other four columns present the results of recovery attempts applying NAHC and SAHC on submodule-level and file-level dependency graph, respectively.

Turbo MQ measurement in Table 6 represents quality of recovered architecture itself in terms of coupling and cohesion. The value of Turbo MQ measurement is calculated as summation of cluster factors for each component in the architecture. A higher value means organization of the architecture is better, i.e., satisfying the ''High Cohesion and Low Coupling'' design principle. From Table 6, we have the following observations. First, both NAHC and SAHC on submodule-level dependencies obtain higher Turbo MQ scores than those on file-level dependencies in most cases. This finding means our technique generally can help Bunch generate component graph with good organization. The only exception is Bash, the scores of Turbo MQ on submodule-level dependencies are lower than file-level dependencies. The reason is that Bash has quite small number of source code files and file-level dependencies, i.e., 150 C files and 345 dependencies. After applying our approach, there are only 27 submodules. The small number of submodules leads to a small number of component, and further results in low score of Turbo MQ measurement because the score is calculated as summation of individual scores for each component. Second, the effectiveness of SAHC is a little higher than that of NAHC. For the 12 recovery attempts of each technique, NAHC outperforms SAHC on 5 attempts, SAHC performs better on 6 attempts, and their scores are equal on 1 attempt. Third, the average scores of Turbo MQ in our experiments are lower than those in the recent empirical study [22]. In that work, the average scores of Turbo MQ on five subjects are 29.2 (Bunch-NAHC, symbol) and 32.2 (Bunch-SAHC, symbol), which are much higher than the average scores in Table 6. The reason is that when applying the directory-based dependency processing technique, the number of components in our recovered architecture is much smaller than the number in their work. And according to the formula (3), an architecture contains smaller number of components will probably obtain a smaller score of Turbo MQ.

MoJoFM measurement in Table 7 represents similarity of recovered architecture and ground-truth architecture. A score of 100% indicates the recovered architecture is the same as the ground-truth architecture. From Table 7, the observations are similar to those from Table 6. Submodule-based recovery attempts generally perform better than file-based recovery attempts in terms of MoJoFM scores, which can be considered as indicator of accuracy. For MoJoFM measurement, Bash is still an exception due to the small number of submodules. In the table, MoJoFM scores of Bash on submodule-level dependencies are the lowest. When using submodule-level dependencies, NAHC identified

TABLE 8. Time cost of each recovery attempt.

Subjects	#Files	Dependency Obtaining	Submodule Grouping	#Submodules	Submodule-NAHC	Submodule-SAHC	File-NAHC	File-SAHC
Bash	150	3s	6s	27	0.02s	0.03s	2s	7s
Httpd	462	11s	20s	90	0.24s	0.50s	416s	1,551s
OODT	1017	32s	55s	538	211s	4,035s	5,769s	66,347s
Joda-time	329	36s	12s	179	5s	28s	618s	1,060s
Hadoop	1707	166s	84s	897	3,498s	21,757s	39,529s	383,431s
Archstudio	2305	196s	115s	1,094	6,647s	53,710s	246,973s	2,198,060s

6 components and SAHC identified 8 components. The ground-truth architecture of Bash consists of 25 components. So there need a lot of *move* operations to transform the recovered architecture to the ground truth architecture. In the recent empirical study [22], the average scores of MoJoFM on five subjects are 34.2 (Bunch-NAHC, symbol) and 46.8 (Bunch-SAHC, symbol). So we can find that the architecture recovered by Bunch can obtain the highest MoJoFM score when using submodule-level dependency graph, and the score on symbol-level dependency is higher than that on file-level dependency. According to the above analysis, we can conclude that the directory-based dependency processing can generally improve the accuracy of automatic software architecture recovery techniques that we studied. Although our technique performs best on improvements of accuracy, the average rate of accuracy is just around 50%. The studied architecture recovery techniques still have significant room for improvements in terms of accuracy.

We further show the detailed time cost of the recovery attempts in Table 8. Column ''Dependencies Obtaining'' presents the time cost of extracting dependencies based on Eclipse CDT and JDT, column''Submodules Grouping'' presents the execution time of our approach to generate submodule-level dependency graph. Column ''#Submodules'' presents the number of submodules our approach generated. The last four columns present the execution time of each recovery attempt, respectively. From Table 8, we have the following observations. First, the directory-based dependency processing technique cannot improve the efficiency of recovery attempts on the subject with small number of source code files in the experiments, i.e., Bash. The reason is that both NAHC and SAHC in Bunch already perform efficient on project Bash. Second, our approach can greatly reduce the time of recovery executions on larger-sized subjects, e.g., OODT, Hadoop and Archstudio. The execution time of Bunch tool grows rapidly with the increase of the number of entities. The iterative process in automatic architecture recovery makes the techniques cannot scale to large-sized subjects [22]. Third, the step of submodules grouping usually take more time than the step of dependencies obtaining on smaller sized subjects, and becomes more efficient on larger sized subjects. So we can conclude that the directory-based dependency processing technique can make structure-based automatic architecture recovery techniques perform efficient on large-sized subjects.

3) RQ3: IMPACTS OF THRESHOLD VALUES

As discussed in Section III, the inter-coupling file is determined by threshold values of inter-indegree and inter-outdegree, and the submodules are grouped based on the inter-coupling files. So the setup of threshold values can directly determine the number of submodules. According to the design of our approach, when threshold values are 0, each source code file will be treated as a submodule, and each directory will be a submodule if threshold values are extremely large. To investigate the detailed impacts of threshold values on the partitions of submodules, we applied the directory-based dependency processing technique with different setup of threshold values. Because directory-based clustering techniques are proved to perform worse than file-based clustering techniques in terms of accuracy [21], [22], our attempts of generating submodules start from file level to directory level, i.e., the threshold values generated set from smaller number to larger number. The threshold values consist of inter-indegree and inter-outdegree, ''inter-'' indicates that we only consider the symbol dependencies between directories in the selection of inter-coupling file. To avoid the situation that too many library files are selected as inter-coupling file, the value of inter-outdegree should be greater than zero. The threshold values are presented as ''inter-indegree''-''inter-outdegree'' and we have evaluated the impacts of 20 different sets of threshold values in the experiments, ranged in {0-1, 0-2...4-4}.

The impacts of threshold values are three-fold: (1) the threshold values can impact the distribution of submodules, improper values may result in oversize submodules. According to the analysis in RQ1, the oversize submodules are likely to contain files that are used to implement irrelevant functionalities. We use CV (i.e., Coefficient of Variation) to measure the quality of distribution. CV is used to measure degree of dispersion for a set of data, it is defined as the ratio of the standard deviation σ to the mean value μ, i.e., $\frac{\sigma}{\mu}$. When the score of CV is low, the sizes of submodules are distributed close to each other. (2) The threshold values can impact the quality of architecture organization which can be measured by Turbo MQ scores. (3) The threshold values can also impact the accuracy of recovered architecture and the

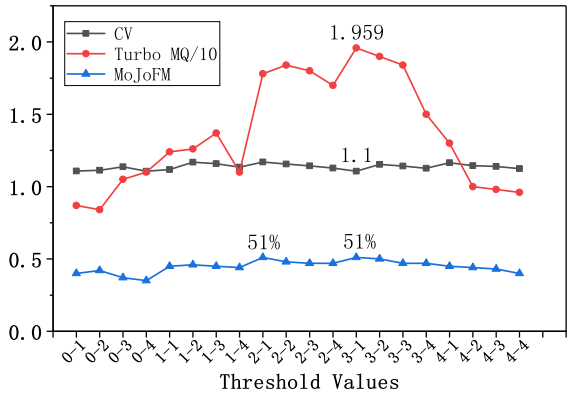


FIGURE 6. The impacts of different threshold values on Bunch-SAHC.

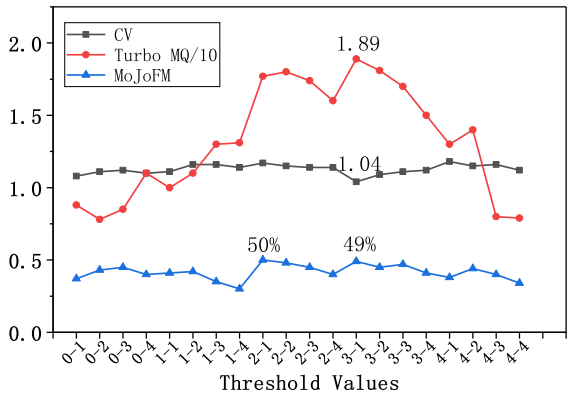


FIGURE 7. The impacts of different threshold values on Bunch-NAHC.

accuracy can be measured by MoJoFM scores. We evaluate the impacts of different sets of threshold values on Bunch-SAHC and Bunch-NAHC in the experiments, and the results of the three kinds of measurements are shown in Figure 6 and Figure 7. In the figures, the x axis represents the different sets of threshold values and the y axis represents the scores of different measurements. The black line of data presents the scores of CV, and the blue line of data presents the scores of MoJoFM . To make the figure clear, we decrease the value to Turbo MQ scores into 10% and present the results in red line of data. From Figure 6 and Figure 7, we have the following observations.

First, for both Bunch-SAHC and Bunch-NAHC, the trend of CV score is stable with the increase of threshold values. This is because the majority of submodules in our experiments only contain 1 or 2 source code files. The number of small-sized submodules is much larger than that of large-sized submodules. When the threshold values increase, the number of large-sized submodules will usually increase, but it is still much less than the number of small-sized submodules. So the scores of CV will not change obviously in this case. Among the 20 sets of threshold values, the set of 3-1 obtains the lowest CV score (i.e., 1.1 and 1.04) for both Bunch-SAHC and Bunch-NAHC, it can generate the best distribution of submodules in our experiments.

Second, for both Bunch-SAHC and Bunch-NAHC, the trend of MoJoFM scores varies slightly with the increase of threshold values. This finding indicates that although our technique can greatly improve the accuracy of recovered architecture, the different sets of threshold values cannot impact the improvements too much. The threshold values 0-1 perform much better than 0-0 for architecture recovery, but the improvements vary slightly with the increase of threshold values. For Bunch-SAHC, threshold values 2-1 and 3-1 can obtain higher MoJoFM scores than other sets, their scores are both 51%. For Bunch-NAHC, threshold values 2-1 performs best in terms of MoJoFM, and threshold values 3-1 obtain a little lower MoJoFM score (i.e., 49%). We have investigated the reason why almost all the sets of threshold values can help to improve the accuracy and perform similar with each other, and find that it is because the improvements on accuracy mainly come from the grouping of remaining files in directories. There are two kinds of remaining files, one kind of files is orphan file which is un-connected with any files on file-level dependency graph, and the other kind of files has dependencies with other files but it is neither inter-coupling file nor intra-coupling file. These two kind of files are usually used to implement different functionalities with inter-coupling files, and the current recovery techniques are hard to cluster them to proper components based on file-level and directory-level dependencies. The orphan files would be separated with any threshold values that are greater than 0, and they are not affected with the increase of threshold values. For the other kind of remaining files, the files that have dependencies with files in same directory may be grouped with other potential inter-coupling files with a high number of threshold values, but the files that only have dependencies with files outside the directory are not affected by the threshold values. So the trend of average MoJoFM scores varies slightly with the increase of threshold values.

Third, in most cases, the score of Turbo MQ increases rapidly when inter-indegree increase from 1 to 2, and decreases rapidly when inter-indegree increase from 3 to 4. The different sets of threshold values can result in different number of submodules, which will further impact the whole organization of the recovered architecture. To our surprise, the threshold values 3-1 still perform best in terms of Turbo MQ score for both Bunch-SAHC and Bunch-NAHC. So we can conclude that the threshold values of symbol dependency (i.e., inter-indegree is 3, inter-outdegree is 1) perform best for the studied projects in our experiments.

C. THREATS TO VALIDITY

1) THREATS TO CONSTRUCT VALIDITY

The main threat to construct validity is the metrics that we used to evaluate the effectiveness of recovered architecture. To reduce this threat, we selected the widely used measurements, i.e., Turbo MQ and MoJoFM. And we will conduct the study with more measurements in the future work. The other threat to construct validity is the threshold values we

used in the experiments. Although we have compared results of the threshold values ranged from 0-1 to 4-4 on all the studied subjects, the data is still not enough to prove that 3-1 is the best suitable threshold values. We will conduct the comparison of more different threshold values on more subjects with more recovery techniques.

2) THREATS TO INTERNAL VALIDITY

The main threat to internal validity is the potential faults in the process of manual recovery. For subject Httpd and Joda-time, we carefully identified the functionality of each submodule and determined the affiliation to component according to the similarity of functionalities and dependencies. To examine the accuracy of architectures we recovered manually, we studied as many as possible related documents, and discussed the conformation of recovered architecture with the professional architect Mr. Wu frequently. For the other four studied subjects, we obtained component-level dependency graphs of these subjects from the existing study [10], and identified the ownership of files to components based on the precise names of components and the conceptual architectures. To reduce this threat, we will try to communicate with the related authors to obtain the detailed document of the architecture. The other threat to internal validity is the potential faults in the configurations of existing tools, or in our data analysis. To reduce this threat, the first author carefully reviewed all the tool configurations, code, and data analysis scripts during the study.

3) THREATS TO EXTERNAL VALIDITY

The subject systems, dependency extracting tools and automatic architecture recovery tools used in our study may all pose threats to external validity. For the subjects, Bash, OODT, Hadoop and Archstudio are already used in several studies [4], [9], [25], [41], and Httpd and Joda-time are also hot projects on github. We obtained the source code of these subjects from several on-line repositories. We used Eclipse CDT and JDT to extract symbol dependencies, and the dependencies may be incomplete due to the limitation of extracting tools. We used only one architecture recovery tool, i.e., Bunch to investigate the impacts of our approach on automatic recovery. Bunch is implemented based on two heuristic methods, genetic algorithm and hill climbing algorithm. The iterative random process makes the recovered architecture is not unique for a specific project. To reduce these threats, we will conduct the study with more dependencies extracting tools, more automatic architecture recovery tools and more subject projects.

V. RELATED WORK

There are already various software architecture recovery techniques in the literature. From a standpoint of input information, current recovery techniques can be roughly divided into two categories, structure-based techniques and knowledge-based techniques [40].

A. STRUCTURE-BASED ARCHITECTURE RECOVERY TECHNIQUES

Bunch [24] is a clustering approach that optimizes the objective features according to an optimization function called Modularization Quality (MQ). Bunch uses two kinds of hill-climbing algorithms to resolve the optimization problem, i.e., nearest and steepest ascent hill climbing (NAHC and SAHC). ACDC [40] is a pattern-based approach that groups code entities according to the way how developers describe the components of a software system. LIMBO [1] is a scalable hierarchical clustering approach that quantifies the relevant information preserved based on the information bottleneck framework. The weighted combined algorithm [25] is a hierarchical clustering approach that groups code entities according to the inter-cluster distance. Structure-based software architecture recovery techniques are usually consist of two parts (i.e., extraction of structural information and grouping method), and are easy to be fully automated. But there are still some structure-based techniques that involves varying degrees of manual work, we categorize them into semi-automatic structure-based software architecture recovery techniques, and the above techniques are typical automatic techniques. Bowman *et al*. obtain conceptual architecture manually and use it to improve concrete architecture [5]. Focus [7] applies manual work to generate idealized architecture evolution and uses it to address affected components. Tool-assisted recovery techniques are frequently-used semi-automatic architecture recovery techniques, they are usually used to obtain ground-truth architectures. The tools can help users to extract and visualize software dependencies, e.g., Rigi [32] and AOVis [17]. Since all these techniques start from a specific level dependency graph, and our approach can produce a submodule-level dependency graph, our approach is potential to improve both automatic and semi-automatic structure-based software architecture recovery techniques.

B. KNOWLEDGE-BASED ARCHITECTURE RECOVERY TECHNIQUES

Kuhn *et al.* [19] proposed a Latent Semantic Indexing based clustering approach that groups code files containing similar terms in the comments. Garcia *et al.* [11] proposed a machine learning based technique that can identify components and connectors according to the concerns. It considers a program as a set of textual information and extracts concerns based on Latent Dirichlet Allocation (LDA) model. Corazza *et al.* [6] proposed a nature language processing-based clustering approach that divides code files into different zones. The zones are weighted based on Expectation-Maximization algorithm, and then grouped by Hierarchical Agglomerative Clustering technique.

C. MEASUREMENT OF SOFTWARE ARCHITECTURE RECOVERY TECHNIQUES

To evaluate the similarity between recovered architecture and ground-truth architecture, several measures are proposed.

MoJo [39] measures the distance between two decompositions of the same set of software entities by counting the minimum number of *Move* and *Join* operations needed to transform the recovered architecture into ground-truth architecture. MoJo also has some extended variants, e.g., MoJoSim [39], MoJoPlus [8], MoJoFM [42] and EdgeMoJo [43]. Among them, MoJoFM is the most widely used measure [3], [4], [15], [22], and it is proved to be more accurate than other measures [9], [16]. Due to the difficulty to obtain ground-truth architecture, there also need a measure which is independent of any ground-truth architecture. Basic Modularization Quality (Basic MQ) [24] measures inter-connectivity (coupling) and intra-connectivity (cohesion) of dependency graph. Turbo MQ [27], [28], extension of Basic MQ, supports dependency graph with edge weights. The assumption behind Turbo MQ is that architecture with high cohesion and low coupling is more acceptable. To conduct an effective comparison of recovered architecture and ground-truth architecture, we use both MoJoFM and Turbo MQ in our experiments.

D. COMPARISON OF SOFTWARE ARCHITECTURE RECOVERY TECHNIQUES

There are several evaluation studies of architecture recovery techniques [2], [4], [6], [9], [31], [44]. But the conclusions of these studies are not always consistent. For example, Architecture Recovery using Concerns and ACDC perform better than other techniques in one study [9]. And LIMBO and ACDC outperform other techniques in another study [2]. The main reason is that the performance of architecture recovery techniques is impressionable with the impacts of different subjects, assessment measures and implementations of techniques in the comparisons [22].

VI. CONCLUSION

We have proposed a novel technique of directory-based dependency processing to generate submodule-level dependency graph based on file-level dependency graph. Our technique utilizes the design concepts hidden in the directory structure to improve effectiveness and efficiency of both manual and automatic architecture recovery techniques.

To evaluate the impacts of our technique on architecture recovery, we conduct a comparison study by applying the directory-based dependency processing technique in both manual and automatic architecture recovery on 6 subject systems. The results show that our technique can greatly improve the efficiency of tool-assisted manual recovery; the submodule-level dependency graph can generally help automatic architecture recovery tools to generate more accurate architecture; our technique can also make the automatic architecture recovery tools easy to scale to large-sized projects. Finally, we prove that the current setup of threshold values is suitable for the studied subjects in our experiments by comparing the impacts of 20 different sets of threshold values.

The results also show that our technique cannot improve the performance of recovery techniques on small-sized projects because current recovery techniques already perform good with small number of software entities. And in rare cases, our technique may generate oversize submodules which contain too many files due to the complex coupling relationships in the directory. This kind of oversize submodules should be examined manually. In the future, we will conduct a more extensive study of the impacts of submodule-based dependency graph with more subjects and more recovery techniques to further improve the directory-based dependency processing technique. We will also try to implement a flexible method for setting of threshold values (e.g., learning the threshold values setting method from historical recovery attempts) to fix the problems of oversize submodules.

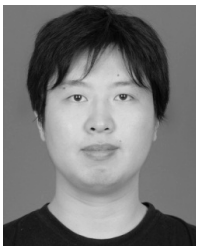
REFERENCES

[1] P. Andritsos, P. Tsaparas, R. J. Miller, and K. C. Sevcik, "LIMBO: Scalable clustering of categorical data," in *Proc. Int. Conf. Extending Database Technol.*, 2004, pp. 123–146.
[2] P. Andritsos and V. Tzerpos, "Information-theoretic software clustering," *IEEE Trans. Softw. Eng.*, vol. 31, no. 2, pp. 150–165, Feb. 2005.
[3] F. Beck and S. Diehl, "Evaluating the impact of software evolution on software clustering," in *Proc. Work. Conf. Reverse Eng.*, Oct. 2010, pp. 99–108.
[4] P. Behnamghader, D. M. Le, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic, "A large-scale study of architectural evolution in open-source software systems," *Empirical Softw. Eng.*, vol. 22, no. 3, pp. 1146–1193, 2017.
[5] I. T. Bowman, R. C. Holt, and N. V. Brewster, "Linux as a case study: Its extracted software architecture," in *Proc. ACM Int. Conf. Softw. Eng.*, May 1999, pp. 555–563.
[6] A. Corazza, S. Di Martino, V. Maggio, and G. Scanniello, "Investigating the use of lexical information for software system clustering," in *Proc. 15th Eur. Conf. Softw. Maintenance Reengineering*, Mar. 2011, pp. 35–44.
[7] L. Ding and N. Medvidovic, "Focus: A light-weight, incremental approach to software architecture recovery and evolution," in *Proc. Work. IEEE/IFIP Conf. Softw. Archit.*, Aug. 2001, pp. 191–200.
[8] M. El-Ramly, P. Iglinski, E. Stroulia, P. Sorenson, and B. Matichuk, "Modeling the system-user dialog using interaction traces," in *Proc. 8th Work. Conf. Reverse Eng.*, Oct. 2001, pp. 208–217.
[9] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Nov. 2013, pp. 486–496.
[10] J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic, "Obtaining ground-truth software architectures," in *Proc. 35th Int. Conf. Softw. Eng.*, May 2013, pp. 901–910.
[11] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns," in *Proc. 26th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Nov. 2011, pp. 552–555.
[12] *Github-Repository. Apache*. [Online]. Available: https://github.com/apache/httpd
[13] *Github-Repository. Joda-Time*. [Online]. Available: https://github.com/JodaOrg/joda-time
[14] M. W. Godfrey and Q. Tu, "Evolution in open source software: A case study," in *Proc. Int. Conf. Softw. Maintenance*, Oct. 2000, pp. 131–142.
[15] Q. Gunqun, Z. Lin, and Z. Li, "Applying complex network method to software clustering," in *Proc. Int. Conf. Comput. Sci. Softw. Eng.*, Dec. 2008, pp. 310–316.
[16] K. Kobayashi, M. Kamimura, K. Kato, K. Yano, and A. Matsuo, "Feature-gathering dependency-based software clustering using dedication and modularity," in *Proc. 28th IEEE Int. Conf. Softw. Maintenance*, Sep. 2012, pp. 462–471.
[17] J. Koch and K. Cooper, "AOVis: A model-driven multiple-graph approach to program fact extraction for AspectJ/Java source code," *Software Eng., Int. J.*, vol. 1, no. 1, pp. 60–71, 2011.
[18] P. Kruchten, "Architectural blueprints—The '4+1' view model of software architecture," *IEEE Softw.*, vol. 12, no. 6, pp. 42–50, Nov. 1995.
[19] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic clustering: Identifying topics in source code," *Inf. Softw. Technol.*, vol. 49, no. 3, pp. 230–243, 2007.

[20] T. C. Lethbridge and N. Anquetil, "Comparative study of clustering algorithms and abstract representations for software remodularisation," *IEE Proc.-Softw.*, vol. 150, no. 3, pp. 185–201, 2003.
[21] T. Lutellier *et al.*, "Comparing software architecture recovery techniques using accurate dependencies," in *Proc. IEEE/ACM 37th Int. Conf. Softw. Eng.*, May 2015, pp. 69–78.
[22] T. Lutellier *et al.*, "Measuring the impact of code dependencies on software architecture recovery techniques," *IEEE Trans. Softw. Eng.*, vol. 44, no. 2, pp. 159–181, Feb. 2018.
[23] A. S. Mamaghani and M. R. Meybodi, "Clustering of software systems using new hybrid algorithms," in *Proc. 9th IEEE Int. Conf. Comput. Inf. Technol.*, Oct. 2009, pp. 20–25.
[24] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner, "Bunch: A clustering tool for the recovery and maintenance of software system structures," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Aug./Sep. 1999, pp. 50–59.
[25] O. Maqbool and H. A. Babri, "The weighted combined algorithm: A linkage algorithm for software clustering," in *Proc. 8th Eur. Conf. Softw. Maintenance Reengineering*, Mar. 2004, pp. 15–24.
[26] O. Maqbool and H. Babri, "Hierarchical clustering for software architecture recovery," *IEEE Trans. Softw. Eng.*, vol. 33, no. 11, pp. 759–780, Nov. 2007.
[27] B. Mitchell, M. Traverso, and S. Mancoridis, "An architecture for distributing the computation of software clustering algorithms," in *Proc. Work. IEEE/IFIP Conf. Softw. Archit.*, Aug. 2001, pp. 181–190.
[28] B. S. Mitchell, "A heuristic approach to solving the software clustering problem," in *Proc. Int. Conf. Softw. Maintenance*, Sep. 2003, pp. 285–288.
[29] D. Rayside, S. Reuss, E. Hedges, and K. Kontogiannis, "The effect of call graph construction algorithms for object-oriented programs on automatic clustering," in *Proc. 8th IEEE Int. Workshop Program Comprehension*, Jun. 2000, pp. 191–200.
[30] M. Shtern and V. Tzerpos, "Refining clustering evaluation using structure indicators," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2009, pp. 297–305.
[31] M. Shtern and V. Tzerpos, "On the comparability of software clustering algorithms," in *Proc. IEEE Int. Conf. Program Comprehension*, Jun./Jul. 2010, pp. 64–67.
[32] M.-A. D. Storey, K. Wong, and H. A. Müller, "Rigi: A visualization environment for reverse engineering," in *Proc. 19th Int. Conf. Softw. Eng.*, May 1997, pp. 606–607.
[33] P. Tonella and A. Potrich, *Reverse Engineering of Object Oriented Code*. Springer, 2005.
[34] *Tool. Bunch*. [Online]. Available: https://www.cs.drexel.edu/~spiros/bunch
[35] *Tool. Count Lines of Code*. [Online]. Available: https://github.com/AlDanial/cloc
[36] *Tool. Dependency Finder*. [Online]. Available: http://depfind.sourceforge.net
[37] *Tool. Doxygen*. [Online]. Available: http://www.doxygen.nl
[38] *Tool. Sonarqube*. [Online]. Available: https://www.sonarqube.org
[39] V. Tzerpos and R. C. Holt, "MoJo: A distance metric for software clusterings," in *Proc. 6th Work. Conf. Reverse Eng.*, Oct. 1999, pp. 187–193.
[40] V. Tzerpos and R. C. Holt, "ACCD: An algorithm for comprehension-driven clustering," in *Proc. 7th Work. Conf. Reverse Eng.*, Nov. 2000, pp. 258–267.
[41] Y. Wang, P. Liu, H. Guo, H. Li, and X. Chen, "Improved hierarchical clustering algorithm for software architecture recovery," in *Proc. Int. Conf. Intell. Comput. Cogn. Inform.*, Jun. 2010, pp. 247–250.
[42] Z. Wen and V. Tzerpos, "An effectiveness measure for software clustering algorithms," in *Proc. 12th Int. Workshop Program Comprehension*, Jun. 2004, pp. 194–203.
[43] Z. Wen and V. Tzerpos, "Evaluating similarity measures for software decompositions," in *Proc. 20th IEEE Int. Conf. Softw. Maintenance*, Sep. 2004, pp. 368–377.
[44] J. Wu, A. E. Hassan, and R. C. Holt, "Comparison of clustering algorithms in the context of software evolution," in *Proc. 21st IEEE Int. Conf. Softw. Maintenance*, Sep. 2005, pp. 525–535.
[45] C. Xiao and V. Tzerpos, "Software clustering based on dynamic dependencies," in *Proc. 9th Eur. Conf. Softw. Maintenance Reengineering*, Mar. 2005, pp. 124–133.

XIANGLONG KONG received the bachelor's degree in computer science from Southeast University, Nanjing, China, in 2009, where he is currently pursuing the Ph.D. degree with the Computer Science and Engineering School, under the supervision of Prof. B. Li at the Software Engineering Institute. He has studied under the supervision of Prof. W. E. Wong and L. Zhang at the Department of Computer Science, The University of Texas at Dallas, from 2014 to 2016. His research interests include architecture recovery, program repair, and mutation testing.

BIXIN LI is currently a Professor with the Computer Science and Engineering School, Southeast University, Nanjing, China. He also leads the Software Engineering Institute, Southeast University, and over 20 young men and women are hard working on national and international projects. He has published over 90 articles in refereed conferences and journals. His research interests include program slicing and its application, software evolution and maintenance, and software modeling, analysis, testing, and verification.



LULU WANG received the bachelor's degree in computer science and the Ph.D. degree in software engineering from Southeast University, Nanjing, China, in 2006 and 2012, respectively. He is currently an Associate Professor with the Computer Science and Engineering School, Southeast University. His research interests include path profiling, program analysis, and program slicing.

WENSHENG WU received the master's degree from the University of Science and Technology of China in 1996. He is currently the Chief Architect with Huawei Technologies Co., Ltd., Shenzhen, China. He is also a famous solution expert in China. He has over 20-year rich experiences in software design and development and project management. His main research interests include the design, measurement, guarding, and evolution of software architecture and products.