Architecture Recovery Based on Design Rule Hierarchy

Yuanfang Cai  
Dept. of Computer Science  
Drexel University  
Philadelphia, PA, USA  
yfcai@cs.drexel.edu

Hanfei Wang  
Dept. of Computer Science  
Nanjing University  
Nanjing, China  
wanghanfei1988@seg.nju.edu.cn

Sunny Wong  
Siemens Healthcare  
Malvern, PA, USA  
sunny.wong@siemens.com

Linzhang Wang  
Dept. of Computer Science  
Nanjing University  
Nanjing, China  
lzwang@nju.edu.cn

Abstract—Software clustering techniques have been extensively studied for the purpose of retrieving a high-level structure of a software system. While prior work has been effective for legacy systems, we observe that a key feature of modern software architectures has not been exploited to improve architecture recovery from code. In this paper, we contribute new architecture recovery techniques based on design rule theory. The rationale is that modern software architecture often features architecture-level design rules that are implemented as special program constructs, such as abstract interfaces. These design rules decouple the rest of the system into components, rather than belonging to one of them. We contribute a family of architecture recovery techniques based on this rationale, enabling the combination of design rule based clustering with other clustering techniques, and the splitting of a large system into subsystems. We evaluated our approach both quantitatively and qualitatively, using eight subjects including open source and real industrial software projects.

Keywords—Architecture Recovery, Design Rule, Modularity, Reverse Engineering

I. INTRODUCTION

Software clustering techniques have been extensively studied for the purpose of retrieving a high-level structure of a software system [1]–[6]. Numerous techniques have been proposed to aggregate entities, such as procedures and source code files, into modules based on different rationales, such as coupling and cohesion [3], [4], [7], naming patterns [8], and minimizing information lost [9]. These techniques are often evaluated and compared using large-scale legacy systems, such as Linux and Mozilla [6].

We observe that important features of modern software architecture are not exploited to improve architecture recovery techniques. Most software systems developed using object-oriented programming languages (e.g. Java and C#) are designed with one or more architectural or design pattern (e.g. publisher-subscriber pattern, model-view-controller pattern, abstract factory pattern). These systems often use a set of interfaces or abstract classes to feature the pattern in use. For example, the application of abstract factory pattern often has a class or interface taking the role of abstract factory [10] that decouples subsystems, such as concrete factories.

According to Baldwin and Clark’s theory [11], these architecture level interfaces are design rules, defined as stable design decisions that decouple other parts of the system into modules. From an architecture point of view, these design rules should not belong to any modules. Instead, they decouple subordinate modules and frame the modular structure of the system. We observe that using traditional software clustering rationales, such as coupling and cohesion, the strong characteristics of design rules cannot be fully exploited and the recovered architecture may not reflect their splitting effects.

In this paper, we first contribute a new clustering approach, which we call an Architectural Design Rule Hierarchy (ArchDRH). Compared with the concept of Design Rule Hierarchy we proposed before [12], ArchDRH shares the same rationale of using design rules to split modules, but uses classes/interfaces as atomic elements to recover architecture, rather than using decisions as basic elements to maximize task parallelism. Our ArchDRH algorithm ensures that architectural design rules, such as the abstract class or interface taking the role of abstract factory, will show up at the top layers of the hierarchy. The subsequent layers contain mutually independent modules that depend on design rules in the upper layers only. Using ArchDRH alone, however, may generate large modules without revealing sufficient lower-level architecture modules.

We thus contribute a family of ArchDRH-based architecture recovery techniques that combine ArchDRH with other clustering methods, such as ACDC [8] and Bunch [7]. The reason is that the design rationale within a module may vary. Some modules may follow strong naming convention while others may be designed based on coupling and cohesion principles. Only the designer can tell which clustering method fits best. Our ArchDRH tools are configurable to support such variations. We also created a method to extract a subsystem that contains only modules that follow the specified set of architectural design rules. Using this technique, the user can conduct various analysis, for example, to see which modules participate in which design patterns.

In this paper, we introduce the following ArchDRH-based clustering family members: (1) ArchDRH-Re, the algorithm that applies ArchDRH recursively within modules, (2) ArchDRH-ACDC, the algorithm that applies ACDC clustering within ArchDRH decoupled modules, (3) ArchDRH-
Bunch, the algorithm that applies Bunch clustering within ArchDRH decoupled modules, and (4) ArchDRH-Split, the algorithm that extracts a subset of the system based on a set of selected design rules. This family is extensible to integrate ArchDRH with other clustering methods.

We evaluated the ArchDRH architecture recovery techniques quantitatively and qualitatively using eight software projects with different sizes and domains, including both open source and real industrial software. For quantitative analysis, we ask our collaborators to provide an authoritative clustering per system to reﬂect their understanding of the architecture. After that, we use the structure indicators proposed by Shtrm and Tzerpos [13], Extraneous Cluster Indicator, Lost Information Indicator, and Fragmentation Indicator, to measure the quality of clustering produced by Bunch, ACDC, ArchDRH-Re, ArchDRH-ACDC, and ArchDRH-Bunch, against the given authoritative clustering. For qualitative evaluation, we either present the ArchDRH-based clustering to our collaborators to get their opinion, or compare the clustering results with existing documentation.

The quantitative evaluation results show that ArchDRH-based techniques can recognize more authoritative components but split a system into more modules. Using ArchDRH-Split may offset this effect by aggregating together modules following same design rules. The results also demonstrate the possibility of using ArchDRH to better understand the implementation when the design model is obsolete, and to identify implementation errors manifested as unexpected dependencies to design rules.

The rest of this paper is organized as follows: Section II explains how our approach differs from other related work. Section III illustrates our approach using a running example. Section IV introduces our algorithms. Section V presents our evaluation results. Section VI discusses threats to validity and future work. Section VII concludes.

II. RELATED WORK

Numerous techniques have been proposed to recover software architecture by clustering low-level entities, such as procedures and classes, into higher-level components. In this section, we discuss representative architecture recovery techniques based on their underlying rationales and methodologies, and how our ArchDRH-based approaches differ. We will also discuss the relation between ArchDRH with our own prior work.

Based on coupling-and-cohesion principle, various clustering algorithms have been proposed [1]–[6], [14], [15]. For example, single-linkage algorithm (SLA) [3] and complete linkage algorithm (CLA) [4] are used to calculate distances between clusters based on their similarity, that is, whether they assess the same set of global variables.

Viewing software clustering as an optimization problem, researchers have used genetic algorithms to decompose a system to modules, such as the work of Harman et al. [16]–[18] and Mancoridis et al. [7]. Bunch [19] is a representative tool using genetic algorithm to conduct clustering-based architecture recovery, aiming to optimize coupling-and-cohesion based quality measures. Bunch has demonstrated encouraging results and fast performance. In this paper, we compare it with our ArchDRH-based approaches.

While previous work has been effective for programs written in procedural languages, such as C, our work aims to analyze modern software systems where architectural or design patterns are applied. The only principle we follow is using design rules to decouple modules, without the need of calculating weights, distances, or association coefficients.

Tsantalis et al. [20], [21] proposed methods to identify design patterns. Tonella and Potrich [22] introduced the techniques of reverse-engineering object-oriented diagrams from code. Adnan et al. [23] used cluster analysis to improve the design of component interfaces. By contrast, we use interfaces or classes that lead a pattern as the basis to decompose a system into modules.

People have used more information and analysis techniques to improve clustering methods, such as concept analysis [24]–[26], latent semantic analysis (LSA) [27] and concern analysis [9], various patterns [8] and data mining techniques [28]–[30].

In this paper, we compare our approach with ACDC [8], a comprehension-driven clustering method that aims to provide effective cluster naming and bounded cluster cardinality. ADCD first clusters a system based on a number of subsystem patterns, such as directory structure pattern and body-header pattern, and then uses orphan adoption methods to aggregate leftover elements. Different from these approaches, our approach is lightweight in that we only need dependencies between classes.

In our prior work [12], we proposed the concept of design rule hierarchy (DRH) for the purpose of maximizing task parallelism. Different from the concept of ArchDRH proposed in this paper that uses class/interface as atomic elements, the elements used in the original DRH are design decisions. For example, we model a class using at least two decisions: a visible interface decision and a hidden implementation decision. Another key difference is that the original DRH uses dependencies derived from an augmented constraint network [31], [32], which captures a lot more indirect dependencies that are not used in ArchDRH. As a result, the resulting clustering are fundamentally different.

In our recent educational paper [33], we reported our experience of directly applying the original DRH algorithm to analyze syntactical dependency among classes. The result was far from satisfactory: the original DRH often generates very large or very small modules even for toy systems. The experiment reported in [33] was conducted manually. The ArchDRH algorithm introduced in Section IV improves the original DRH for the purpose of architecture recovery.
III. ILLUSTRATIVE EXAMPLES

In this section, we use a running example first to introduce the background knowledge of our work, the design structure matrix (DSM) [11] that we use to represent recovered architecture in a scalable way, and the concept of design rule hierarchy [12]. We will also use this example to illustrate the key difference between ArchDRH clustering and two other prevailing clustering techniques: Bunch [7] and ACDC [8].

The example we use is an implementation of a maze game, whose design is slightly modified from the classic GoF design pattern book [10]. A maze consists of multiple rooms, each with walls and a door to another room. The game is designed to support two variations by applying an abstract factory pattern. This example was used as a lab assignment at Drexel and we select the best implementation as the running example. The source code is written in Java, and has 15 classes. Since the implementation has been confirmed to be correct, we expect that an effective clustering technique should be able to faithfully recover the designed architecture (e.g. revealing the existence of two separated concrete factories).

A. Background Concepts

Design Structure Matrix (DSM). In this paper, we represent all the clusterings recovered from source code using DSMs [11], [34]. A DSM is square matrix. Its columns and rows are labeled with the same set of elements in the same order. If an element on row $x$ depends on the element on column $y$, then the cell $(rx, cy)$ will be marked.

Figure 1 depicts a DSM reverse-engineered from the maze game implementation. This DSM lists the 15 classes in three layers (the box with solid border) and five modules (the inner blocks along the diagonal), showing a hierarchical structure. In this paper, we view the most inner blocks as modules.

![Figure 1: ArchDRH clustering of Maze Game](image)

We use our tool Titan [35] to visualize DSMs. Titan accepts a .dsm file, which only represents the dependency relation between elements, and a .clsx file that represents the clustering (decomposition) of these elements. A hierarchical structure is represented as a tree structure. In Titan, the user can collapse or expand the tree nodes, and view the corresponding DSM. All the DSMs in this paper are exported from Titan. When comparing multiple clustering methods, we transform the output of these clustering tools into .clsx files, and these files share the same .dsm file because they all work on the same dependency relation.

As a result, we were able to use Titan to view DSMs clustered by different architecture recovery tools. Figure 2(b) shows the DSM transformed from the output of Bunch and Figure 2(c) shows the DSM transformed from the output of ACDC.

Design Rule Hierarchy (DRH). In our prior work [12], we proposed the concept of DRH for the purpose of maximizing task parallelism. The elements of the DRH are design decisions. The top layer contains design decisions that decouple the rest of the system. The subsequent layers contain design decisions that only depend on decisions in upper layers. The most unique characteristic of DRH is that modules within a layer are independent from each other so that they can be implemented concurrently and in parallel by different teams.

B. Design Rule Based Architecture Recovery

Our experience shows applying the original decision-based DRH algorithm to classes and their syntactical dependencies often results in large modules that need to be further decomposed. We thus create a ArchDRH algorithm, tailored from the original DRH algorithm, for the purpose of architecture recovery. Based on this new ArchDRH algorithm, we propose a family of ArchDRH-based clustering approaches that allow the user to further cluster a module using another clustering method, such as ACDC or Bunch, based on the design rationale of the system. For example, if the system follows strict naming convention, applying ACDC within a module may better facilitate the users’ understanding of the architecture. The user may also choose to apply ArchDRH recursively within a module till no new modules are discovered. We call the algorithms that apply ArchDRH recursively, combine with Bunch and combine with ACDC as ArchDRH-Re, ArchDRH-Bunch and ArchDRH-ACDC respectively.

Figure 1 depicts a DSM reverse-engineered from the maze game implementation using our ArchDRH-Re algorithm. In this DSM, class MapSite and its subclasses, Wall, Door, and Room, form the first module (r1-5, c1-5). The interface MazeFactory takes the role of the abstract factory in the pattern and is also listed in the top layer. The DSM also shows that these first two modules decouple the elements in the second layer into two separate modules: one module (r7-10, c7-10) contains the classes implementing a concrete factory, BlueMazeFactory, including BlueWall, GreenRoom, and BlackBlueDoor. The other module module (r11-14, c11-14) contains the classes implementing another concrete factory, RedMazeFactory, including RedWall, RedRoom, and
BrownDoor. The bottom layer only contains one element, class SimpleMazeGame. This class has the static main() function that depends on many other classes.

We observe that this clustering faithfully reflects the designed architecture: the two concrete factory modules are completed separated by the abstract interface and the base classes. The class contains the main() function acts as a controller class and should not belong to any other modules.

The ArchDRH-ACDC generates exactly the same clustering as ArchDRH-Re, meaning that ACDC similarly cannot split these modules further. Figure 2 (a) depicts the clustering generated by ArchDRH-Bunch, which splits the first module into two inner modules.

Now we compare these DSMs to demonstrate their differences. Figure 2 depicts the DSMs produced by ArchDRH-Bunch, Bunch and ACDC respectively. Comparing Figure 1 with Figure 2 (c), we can see that ACDC generates a very similar clustering with ArchDRH: the classes of the two concrete factories are aggregated into separate modules. The difference is that the control class containing the main() function was aggregated with architectural level design rules. Another difference is that ACDC does not generate a hierarchical structure. Bunch clusters the system quite differently from the other two approaches. Although Bunch clustering optimizes coupling-and-cohesion quality measure, it is hard to find the architecture of the design. For example, SimpleMazeGame and Maze are aggregated together, but this module does not represent any meaningful components of the system.

In summary, we observe that ArchDRH-Re correctly detects components in the system that are decoupled by architectural design rules. ACDC works similarly in this example because the source code also demonstrates strong naming pattern that matches ACDC’s clustering rationale. But ACDC does not separate the control class containing the main() function, which should not belong to any components. Clusters generated by Bunch are less useful because coupling and cohesion are not the dominating design rationale of the maze game. The combination of ArchDRH and Bunch further splits the first module produced by ArchDRH-Re into two inner modules, but these inner modules do not contribute much: MapSite and Wall do not form a meaningful module. As a result, it is reasonable to consider the DSM shown in Figure 1 to be the authoritative clustering of this maze game implementation.

IV. ARCHDRH ALGORITHMS

In this section, we use the maze game example to illustrate the ArchDRH algorithm and the family of architecture recovery techniques based on ArchDRH. Figure 3 depicts the dependency graph of the maze game example, reverse-engineered using our tool Moka [35]. Each oval models a Java class or interface. A directed edge \((u, v)\) models that \(u\) influences \(v\) (e.g. Mapsite influences Room). We use this dependency graph to illustrate our architecture recovery algorithms in the following subsections.

A. ArchDRH Algorithm

The ArchDRH algorithm consists of three major steps:

1) Step 1. Identifying function modules: We define a set of classes working together for a function as a function module, and leverage Cai and Sullivan’s decomposition algorithm [31], [32] to identify them. The basic idea is to take the dependency graph as input, and compute its condensation graph, in which each vertex represents a strongly-connected
component (i.e., comprising classes that are closely coupled by cyclical dependency). A condensation graph is a directed acyclic graph (DAG). We aggregate all the vertices along the paths ending with the same minimal elements together as a function module.

2) Step 2. Identifying shared components: Some classes/interfaces may be shared by multiple function modules. For example, the MazeFactory interface is shared by both concrete factories. We thus identify a hierarchy from the condensation graph using Wong et al.’s [12] design rule hierarchy algorithm. Intuitively, their algorithm identifies each region of intersection (i.e., the set of classes shared by multiple function modules) in the condensation graph and separates it into an individual module. Taking each module as a node, this algorithm will produce a new directed acyclic graph. By applying a modified breadth first search of its vertices, we can get a partial ordering that forms the design rule hierarchy, as defined in Wong et al.’s work [12].

3) Step 3. Identifying conceptual modules: The function modules detected from step 1 and step 2 usually are sub-modules decoupled by the application of architecture design rules. For example, a strategy interface decouples multiple concrete strategies into their own concrete strategy function modules. From the architectural point of view, the functions that follow the same interfaces can be viewed as one bigger module because they implement the same concept.

We use the design of a decorator pattern to illustrate this point. Figure 4 depicts the design of a coffee/tea ordering system using decorator pattern, an example modified from [36]. In this design, the Beverage class takes the role of component, the Ingredient class leads the role of decorator, and the CoffeeBeverage and TeaBeverage are concrete components. In this case, although the Ingredient interface separates the five concrete ingredients, these ingredients can be considered as a Ingredient module since they all implement the same concept and form an architectural module.

Accordingly, the third step of our ArchDRH algorithm searches for the function modules within the same layer of a design rule hierarchy and aggregate all the function modules that follow the same set of design rules into a conceptual module. The modules (r5-7, c5-7), (r8-12, c8-12), and (r13-15, c13-15) in Figure 5 are the conceptual modules formed this way, modeling the existence of coffee beverage module, tea beverage module and ingredients module.

The complexity of the ArchDRH algorithm is bounded by the original DRH algorithm [12], which is $O(|V|^2)$, where $|V|$ is the number of classes/interfaces in the source code.

B. ArchDRH Family

Our ArchDRH algorithm identifies function modules by aggregating a set of classes used by an minimal element class in the condensation graph. As a result, if a class uses (i.e., depends on) a large number of other classes, then it tends to aggregate many classes into a big module. The most prominent example is the main() function of a control class that usually depends on many other classes and thus aggregates a large modules. In the decorator and abstract factory pattern example, applying ArchDRH alone will produce just one big module due to the existence of the control classes with a main() function (SimpleMazeGame and StarBuzz). We call such a minimal element class as a control class. Since the modules formed by these control classes can involve many classes with their own modular structure, we thus propose a family of ArchDRH based architecture recovery techniques to address this problem.

1) Recursive ArchDRH: ArchDRH-Re algorithm is based on ArchDRH. For each function module identified by ArchDRH, ArchDRH-Re ignores its control class and applies ArchDRH again on the graph formed by the rest of the vertices. Figure 3 depicts the decomposition generated by ArchDRH-Re. After the first run of ArchDRH, ArchDRH-Re ignores SimpleMazeGame and processes the rest of the graph. At this time, there are two minimal elements BlueMazeFactory and RedMazeFactory, which lead to two function modules and two design rule modules. ArchDRH-
Re will again ignore the control class of each of the four new modules, but no new modules can be found because it reaches the stop condition of the recursive function.

ArchDRH-Re stops when the density of a module is too low or too high. A very high density suggests that the elements in this module are highly cohesive with each other and they should be clustered into a module. On the other hand, a very low density suggests that the information in this graph is so little that further clustering cannot produce meaningful modules. In the current implementation, for module \( m \) with \( |E| \) dependencies among \( |V| \) classes, we define the stop condition as: \(|E| < |V|\) (too sparse) or \(|E| > |V|^{1.5}\) (too dense). We discuss the selection of this stop condition in Section VI.

The complexity of ArchDRH-Re is calculated as follows: the number of recursions is bounded by the number of clusters ArchDRH-Re produced in the previous round. The number of clusters produced is bounded by the number of classes in the dependency graph. As we discussed before the complexity of ArchDRH alone is \( O(|V|^2) \), thus the complexity of ArchDRH-Re is \( O(|V|^3) \).

2) Combined ArchDRH: Instead of using ArchDRH-Re to further decompose large function modules into smaller ones, we can also combine ArchDRH with other clustering methods to accommodate different design rationale within modules. In this paper, we explore the combination of ArchDRH with Bunch and ACDC (introduced in Section II). The user can choose to run ArchDRH once or run ArchDRH-Re multiple times before applying Bunch or ACDC to inner modules. Figure 2 (a) is produced by applying ArchDRH-Re two rounds to split the system into five modules and then apply Bunch within each module.

C. ArchDRH-Split

In a software design where multiple patterns are applied, it is normal for a function module to participate in multiple design patterns, and a class may take roles of different patterns by implementing different interfaces. In this case, it can be difficult to understand the modular structure framed by these design rules in a large system.

We contribute a ArchDRH-Split algorithm to produce a partial view of the overall system based on a specified set of design rules. This algorithm takes as input a set of specified design rules and a DSM clustered by one of the ArchDRH family members. ArchDRH-Split will extract a subset of the DSM in which the specified design rules appear at the top of the DSM, followed by the function modules the depend on the specified design rules. Figure 7 depicts a DSM resulting from ArchDRH-Split, containing all the function modules that depend on Tool.

V. Evaluation

To evaluate the effectiveness of the ArchDRH-based architecture recovery approaches, we conducted both quantitative and qualitative evaluations using eight software projects of different sizes and domains\(^1\).

A. Subjects

Table I lists the basic information of the eight subject systems we use. The first three subjects are software projects built at Drexel by Dr. Wong when he was a Ph.D. student. Subjects 4-6 are the three versions of a real industrial project for which we keep their name anonymous. Subject 7 is an open source project and Subject 8 is provided by our academic collaborators. We now briefly introduce the subject systems as follows:

<table>
<thead>
<tr>
<th>Subject</th>
<th>LOC</th>
<th>Classes</th>
<th>Language</th>
<th>Eval</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Moka</td>
<td>3280</td>
<td>Java</td>
<td>Quant.</td>
</tr>
<tr>
<td>2</td>
<td>Titan</td>
<td>2032</td>
<td>35</td>
<td>Java</td>
</tr>
<tr>
<td>3</td>
<td>Minos</td>
<td>1697</td>
<td>Java</td>
<td>Quant.</td>
</tr>
<tr>
<td>4</td>
<td>Sprint1</td>
<td>18.5K</td>
<td>391</td>
<td>C#</td>
</tr>
<tr>
<td>5</td>
<td>Sprint2</td>
<td>25K</td>
<td>857</td>
<td>C#</td>
</tr>
<tr>
<td>6</td>
<td>Sprint3</td>
<td>25K</td>
<td>861</td>
<td>C#</td>
</tr>
<tr>
<td>7</td>
<td>JHotDraw</td>
<td>5290</td>
<td>171</td>
<td>Java</td>
</tr>
<tr>
<td>8</td>
<td>CourseQ&amp;A</td>
<td>3873</td>
<td>32</td>
<td>Java</td>
</tr>
</tbody>
</table>

\(Moka\) \(^2\) is a program that extracts a UML model from compiled code. It has been used to extract dependency relation from the source code used in this study and our prior work \([12], [37], [38]\).

\(Minos\) \(^3\) is a program that takes as input design models represented in \textit{augmented constraint networks} (ACNs) \([31], [32]\) and generates DSMs with precise semantics.

\(Titan\) \(^\text{[35]}\) is a DSM tool that allows the user to manipulate the hierarchical structure, visualize DSMs in a scalable way, and export a DSM into spreadsheet.

\(Sprint1, Sprint2\) and \(Sprint3\) are three consecutive releases of an industrial project, which we refer as \(Sprint\). This is an application used to manage and render data using spreadsheet from multiple data sources. This project has gone through five years of evolution. \(Sprint2\) evolves from \(Sprint1\) by adding new features. In \(Sprint3\), the architecture was refactored by applying several design patterns for the purpose of easing future maintenance.

\(JHotDraw\)\(^3\) (JHD) is a Java framework for creating graph. We chose this subject because it was designed to exploit many well-known design patterns. The first version of this system is well-documented \([39]\) and studied \([20]\).

\(CourseQ&A\) is web-based course management system used that allows teachers to create exams and allows students to take exams.

\(1\) All the DSMs used in this paper can be found at https://www.cs.drexel.edu/ yfcai/icse2012/data
\(2\) Moka, Minos and Titan can be downloaded at http://rise.cs.drexel.edu/projects/
\(3\) http://www.jhotdraw.org/
The qualitative evaluation results obtained from our collaborators show that ArchDRH-Split is the most effective among other clustering methods, but we were not aware of any evaluation methods that can be used to compare ArchDRH-Split with other clustering methods because it requires the user to specify design rules and generates a partial view of the system. As a result, we only evaluate ArchDRH-Split qualitatively and evaluate ArchDRH-Re, ArchDRH-ACDC and ArchDRH-Bunch quantitatively. We ran our experiments on a Windows PC with 2.30GHz AMD Turion(tm) II P530 dual-Core processor and 4GB of RAM. The longest time used to run an ArchDRH algorithm was 13 seconds.

B. Quantitative Evaluation

Quantitative comparison of different clustering methods requires the existence of an authoritative clustering that reflects the designers' understanding of the architecture. We chose the first six subjects for quantitative analysis because our collaborators were able to provide authoritative clusterings for them. We then use the structure indicators proposed by Shtern and Tzerpos [13] to quantitatively compare the clustering generated by Bunch, ACDC, ArchDRH-Re, ArchDRH-Bunch, and ArchDRH-ACDC, against the authoritative clustering. We chose this evaluation method because they address the issues of other prevailing evaluation methods. More discussion will follow in Section VI.

Table II: Module Count

<table>
<thead>
<tr>
<th>Subject</th>
<th>Auth.</th>
<th>ACDC</th>
<th>Bunch</th>
<th>DRHR</th>
<th>DRHB</th>
<th>DRHA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moka</td>
<td>20</td>
<td>15</td>
<td>21</td>
<td>25</td>
<td>29</td>
<td>27</td>
</tr>
<tr>
<td>Titan</td>
<td>13</td>
<td>3</td>
<td>22</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>Minos</td>
<td>9</td>
<td>7</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>Sprint1</td>
<td>29</td>
<td>42</td>
<td>96</td>
<td>88</td>
<td>94</td>
<td>69</td>
</tr>
<tr>
<td>Sprint2</td>
<td>51</td>
<td>100</td>
<td>204</td>
<td>230</td>
<td>186</td>
<td>152</td>
</tr>
<tr>
<td>Sprint3</td>
<td>67</td>
<td>100</td>
<td>192</td>
<td>200</td>
<td>189</td>
<td>157</td>
</tr>
</tbody>
</table>

Following the definition of Shtern and Tzerpos, we refer the five clustering methods as test clusterings. If a module, \( T_i \), in a test clustering contains more than 50% of the elements of a module, \( A_i \), in the authoritative clustering, then \( T_i \) is defined to be a segment of \( A_i \). The structure indicator consists of three values:

—**Extraneous Clustering Indicator** (E) measures the number of modules in the test clustering that are not segments of any modules in the authoritative clustering. That is, the number of modules in the test clustering that are not meaningful.

—**Lost Information Indicator** (L) measures the number of modules in the authoritative clustering that do not have any segments in the test clustering. That is, the number of authoritative modules that are not recognized.

—**Fragmentation Indicator** (F) measures the average number of segments for all authoritative modules recognized in the test clustering. The larger the value of F, the more segments an authoritative module has, meaning that an authoritative module are separated into more parts.

Table III: Extraneous Cluster Indicator

<table>
<thead>
<tr>
<th>Subject</th>
<th>ACDC</th>
<th>Bunch</th>
<th>DRHR</th>
<th>DRHB</th>
<th>DRHA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moka</td>
<td>10(15)</td>
<td>13(21)</td>
<td>3(25)</td>
<td>11(29)</td>
<td>10(27)</td>
</tr>
<tr>
<td>Titan</td>
<td>1(3)</td>
<td>4(22)</td>
<td>1(15)</td>
<td>2(15)</td>
<td>1(15)</td>
</tr>
<tr>
<td>Minos</td>
<td>2(7)</td>
<td>2(7)</td>
<td>2(10)</td>
<td>2(11)</td>
<td>2(11)</td>
</tr>
<tr>
<td>Sprint1</td>
<td>10(42)</td>
<td>29(96)</td>
<td>11(88)</td>
<td>21(94)</td>
<td>14(69)</td>
</tr>
<tr>
<td>Sprint2</td>
<td>23(100)</td>
<td>65(204)</td>
<td>27(230)</td>
<td>36(186)</td>
<td>34(152)</td>
</tr>
<tr>
<td>Sprint3</td>
<td>30(100)</td>
<td>69(192)</td>
<td>37(200)</td>
<td>40(189)</td>
<td>42(157)</td>
</tr>
</tbody>
</table>

Table IV: Lost Information Indicator

<table>
<thead>
<tr>
<th>Subject</th>
<th>ACDC</th>
<th>Bunch</th>
<th>DRHR</th>
<th>DRHB</th>
<th>DRHA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moka</td>
<td>15(20)</td>
<td>13(20)</td>
<td>5(20)</td>
<td>7(20)</td>
<td>8(20)</td>
</tr>
<tr>
<td>Titan</td>
<td>12(13)</td>
<td>7(13)</td>
<td>3(13)</td>
<td>3(13)</td>
<td>3(13)</td>
</tr>
<tr>
<td>Minos</td>
<td>5(9)</td>
<td>4(9)</td>
<td>3(9)</td>
<td>2(9)</td>
<td>2(9)</td>
</tr>
<tr>
<td>Sprint1</td>
<td>12(29)</td>
<td>4(29)</td>
<td>4(29)</td>
<td>5(29)</td>
<td>9(29)</td>
</tr>
<tr>
<td>Sprint2</td>
<td>25(51)</td>
<td>6(51)</td>
<td>6(51)</td>
<td>5(51)</td>
<td>18(51)</td>
</tr>
<tr>
<td>Sprint3</td>
<td>37(67)</td>
<td>16(67)</td>
<td>18(67)</td>
<td>11(67)</td>
<td>26(67)</td>
</tr>
</tbody>
</table>

If a clustering matches exactly the authoritative clustering, then it will have a ELF vector of (0, 0, 1). Please note that a clustering with a ELF of (0, 0, 1) does not mean that the clustering is exactly the same as the authoritative one. For example, we consider an authoritative clustering with six elements split into two modules: \( A = \{a_1, a_2, a_3, a_4, a_5, a_6\} \). If a test clustering is \( T = \{a_1, a_2, a_4, (a_3, a_5, a_6)\} \), then its ELF will be (0, 0, 1). Although they are similar, they are not exactly the same. The larger the EFL values, the more dissimilar the two clustering are.

Table II lists the number of modules in the authoritative decomposition and the clusterings generated by the five architecture recovery approaches. Table III, IV and V compares the five clustering methods using Extraneous Clustering Indicator, Lost Information Indicator and Fragmentation Indicator respectively. In these tables, DRHR, DRHA and DRHB means ArchDRH-Re, ArchDRH-ACDC, and Arch-Bunch respectively, and a gray color cell contains the best result of the row.

For example, in Table III, the first cell is 10(15), meaning that ACDC generates 15 modules from Moka source code, and 10 of them are extraneous (i.e. do not form meaningful components.). In Table IV, the first cell is 15(20), meaning that all of the 20 modules in Moka authoritative clustering, 15 of them were not recognized by ACDC. In Table V, the first cell is 1.000, meaning that each of the five authoritative modules recognized by ACDC has one segment in ACDC clustering of Moka. Comprehensively, ACDC does not appear to be an effective architecture recovery approach for Moka even though it has the best F value.

We make the following observations from these tables. First, ACDC tends to generate fewer modules than Bunch or
ArchDRH clustering. ACDC also generates fewer extraneous modules except for Moka. ArchDRH-Re is similar to ACDC in terms of number of extraneous modules, but considering that ArchDRH-Re has more modules, the percentage of extraneous modules is actually smaller than that of ACDC.

Although ACDC generates fewer extraneous modules, it failed to recognize more than half of the authoritative modules for all the subject systems. Bunch and ArchDRH family generate similar number of modules, and recognize similar number of authoritative modules except for Titan and Moka where ArchDRH family recognized more authoritative modules. For the three industrial projects, Bunch generates twice as many as extraneous modules than ArchDRH-Re.

Table IV shows that our ArchDRH-based clusterings are the winner in terms of identifying more authoritative modules. The best clustering method varies with subjects. ArchDRH-Bunch seems to fit Titan, Minos, Sprint1 and Sprint2 better, and ArchDRH-ACDC and ArchDRH-Re appear to be more effective for Moka, Titan and Minos.

Table V: Fragmentation Indicator

<table>
<thead>
<tr>
<th>Subject</th>
<th>ACDC</th>
<th>Bunch</th>
<th>DRHR</th>
<th>DRHB</th>
<th>DRHA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moka</td>
<td>1.000</td>
<td>1.143</td>
<td>1.467</td>
<td>1.385</td>
<td>1.417</td>
</tr>
<tr>
<td>Titan</td>
<td>1.000</td>
<td>1.000</td>
<td>1.300</td>
<td>1.300</td>
<td>1.400</td>
</tr>
<tr>
<td>Minos</td>
<td>1.250</td>
<td>1.000</td>
<td>1.333</td>
<td>1.286</td>
<td>1.286</td>
</tr>
<tr>
<td>Sprint1</td>
<td>1.882</td>
<td>2.680</td>
<td>3.080</td>
<td>3.042</td>
<td>2.750</td>
</tr>
<tr>
<td>Sprint2</td>
<td>2.962</td>
<td>3.089</td>
<td>4.511</td>
<td>3.261</td>
<td>3.576</td>
</tr>
<tr>
<td>Sprint3</td>
<td>2.333</td>
<td>2.412</td>
<td>3.327</td>
<td>2.714</td>
<td>2.805</td>
</tr>
</tbody>
</table>

Table V shows that ArchDRH-based clusterings always produce more fragments than ACDC and Bunch. The reason is that ArchDRH separates interfaces or classes that are architecture design rules into separate modules. Figure 6 depicts the top layers of the JHotDraw DSM in Titan. The figure shows that many design rules are aggregated into single element modules, which explains why ArchDRH based clustering always has more modules and more fragments. As we will discuss next, larger number of fragment can be offset by the fact that ArchDRH-Split can aggregate functions following the same design rules together.

C. Qualitative Evaluation

Due to the different nature of the subject systems we use and the availability of the designers, we discuss the qualitative evaluation results for each subject system separately.

1) JHotDraw 5.2: This application is designed using multiple design patterns, such as strategy, composite, decorator, factory method, observer, and template methods. These design patterns are all implemented using interfaces or abstract classes. For example, the composite pattern used in this project is governed by architectural design rules

(2) Functional Module Identification. The names assigned to these 42 modules hint at the functions they model. We use a sub-DSM extracted by ArchDRH-Split to see if these modules truly reflect meaningful functions. From the description of the JHD design, the drawing editor functions are implemented using strategy pattern with the Tool class as the strategy interface. Figure 7 depicts the DSM extracted by ArchDRH-Split, showing all the function modules that depend on Tool.

In this DSM, the rows with shaded background indicate the names of the module given by ArchDRH-Re. The first module that contains Tool and AbstractTool dominates (influence) 11 functional modules. Some of them
Figure 7: Partial View of JHotDraw

are concrete strategies, such as SelectionTool (r3-6,c3-6), ConnectionTool (r7,c7), TextTool (r8-11, c8-11), and some of them are the client of the strategy pattern, such as DrawApplication (r24, c24) (a folded module with 14 elements in it). The numbers in the cell model the number of dependencies between modules. For example, cell (r24, c22) is 3, meaning that there are three dependencies from the DrawApplication module to the StandardDrawingView module.

For the sake of space, the last five modules are folded. It is easy to see that each module does reflect one editing function as designed. This DSM also shows that the client of the strategy pattern and the concrete strategy modules are completely separated, as in its intended design [39].

(3) Pattern Participation. By looking at which architectural design rules a module follows (depends on), we can easily identify which patterns it participates in. For example, the BorderTool module follows DrawingView, DrawingEditor, Figure, FigureChangeListener, AbstractTool and Drawing, hence participating in the design patterns they lead.

2) Sprint: As an industrial project with customer obligations, Sprint often sacrifices reliable documentation for meeting deadlines. Using ArchDRH-Re, we clustered all three releases of the Sprint project. Sprint1 DSM has 4 layers and 120 modules, Sprint2 DSM has 6 layers and 315 modules, and Sprint3 DSM has 9 layers and 298 modules. When we presented the full DSMs to our collaborator, the response we got was that "The clustering makes a lot of sense", but no more observation can be made quickly given the size of the project.

Next we asked our collaborator to provide some design rules they used to see if splitting the big DSM to smaller ones can provide more insight. Although there is no documentation and the architecture decisions used in the first two versions were hard to be retrieved, our collaborator, who is also the project manager of the project, was able to recall the design patterns applied to refactor Sprint2 into Sprint3. Using Titan, he was able to identifying the interfaces/classes that lead these design patterns easily because they are all aggregated at the top the DSM.

Our collaborator specified the following five sets of design rule interfaces: DR1: IResultSet, DR2: IMetadata and IMetadataProvider, DR3: Expression DR4: DataSource, IDataProvider and IContentStore, DR5: IReportBlock, IReportAxis and CDLAbasReport. We extracted five smaller DSMs from Sprint3 using ArchDRH-Split. These DSMs have 19, 46, 34, 32, 42 modules respectively, each following one set of design rules.

We presented these much smaller sub-DSMs to our collaborator and he was able to identify implementation errors by discerning modules that should not depend on the given design rules. Concretely, he identified 2 erroneous and 1 questionable (need to be confirmed) dependencies on DR1, 1 erroneous and 1 questionable dependencies on DR2, 4 erroneous dependencies on DR3, 1 questionable dependency to DR4, and 2 erroneous dependencies on DR5. Our collaborator also identified several modules that are too large or contain multiple functions that may be caused by more subtle but suspicious dependencies.

3) CourseQ&A: This is a web-based information management system for teachers to create and manage exams and for students to take exams. The system is designed by our academic collaborators and implemented by their students. The design is modeled in a dataflow diagram, and the code is implemented using Java and JSP pages. The high-level design shows 17 processes with 8 databases. Each process is implemented by one or more Java files. Although our collaborator designed the system, the code was implemented by students who were not reachable and we were not able to get an authoritative clustering. Since each process is a relatively independent function, so they expect the number of modules to be around 17.

We ran ArchDRH-Re to generate a DSM and sent it back to our collaborators. To their surprise, the DSM contained 67 modules, many more modules than they expected. After having a closer look at the DSM, they realized that the students implemented many small independent utility functions, such as exporting a gradesheet, searching student by name, finding the highest score. These utility functions are not the main process of the system. They were not modeled in the dataflow diagram, and even the designer were not aware of their existence before they saw the DSM. Our collaborators was also able to identify implementation problems, such as the access of a database table from a class that should not have the right to access that table. Currently they are working on fixing the implementation and complete the design model referencing to the ArchDRH-Re DSM.
VI. DISCUSSION

We now discuss threats to validity and future work.

**Threats to validity.** Our results can be sensitive to the subject systems in use. We only evaluated ArchDRH-based techniques using Java and C# projects, but not using large-scale legacy code against which other prevailing clustering techniques compare with each other, such as Linux. We cannot claim that ArchDRH will be affective for systems that are not designed using architectural design rules.

Our results can be sensitive to the authoritative clustering provided by our collaborators. Our ArchDRH-based techniques separate architectural design rules into separated modules, leading to a large number of small modules, each having a single or a few design rules. This unique feature results in a lot more modules and thus high fragmentation value, because the designers usually do not view each individual design rules as a separate module. From the Sprint project, we observe that the designer often aggregates a set of design rules, such as a group of spreadsheet APIs, into one module. Although there are no dependencies between these APIs, the designer considers them as a module because they share similar concept.

Our results can be sensitive to the quality of the implementation. In our prior work [33], we noticed that if the implementation of the maze game has errors, then none of the clustering methods can produce a decomposition exactly the same as the one in Figure 1. When an implementation is significantly deviated from the intended design, it could be difficult for any clustering methods to work effectively. We will conduct more sensitive analyses in the near future.

Our results can also be different if we compare with other clustering methods other than Bunch and ACDC. More advanced clustering methods, such as concern-based clustering [9], [15], may produce better results with more information. We will explore the possibility of combining ArchDRH with other clustering methods. Our result of ArchDRH-Re may also be sensitive to the selection of stop threshold. We will conduct more experiments for further evaluation.

Our results can be sensitive to the evaluation method in use. We chose the three structure indicator proposed by Shtern and Tzerpos [13] because they address the known issues of other prevailing evaluation methods, such as MojoFM [40] and KE [41]. These measures may lead to significantly different results for the same clustering. Other evaluation methods, such as EdgeSim and McCI [42], may produce different results. In particular, we plan to further evaluate our approach using the method proposed by Wu et al. [43], evaluating our approach in terms of stability, authoritativeness and extremity of cluster distribution.

**Future work.** In the near future, we will explore the possibility of leveraging ArchDRH to detect implementation errors, especially using the ArchDRH-Split approach. It is possible the source code is implemented by multiple developers and contains many code smells or technical debts, making it difficult to understand. But the designers of the system usually should be able to remember the architectural design rules they used, so that subsystems following these design rules can be extracted and examined.

In this paper, classes/interfaces are atomic elements of our clustering method. It is also possible to represent elements at different level of granularity, such as attributes and methods. We only considered direct syntactical dependency in this work. We will also explore the effectiveness of our approach on both indirect and direct dependencies, such as the logical dependencies discovered by ACN modeling.

VII. CONCLUSION

Based on the observation that modern software systems often employ architectural or design patterns implemented with interfaces or abstract classes as architectural design rules, we contribute a suite of design rule based architecture recovery techniques that can combine design rule hierarchy clustering with other clustering methods and split a large system into subsystems based on the design rules they follow.

We evaluated our techniques both quantitatively and qualitatively using in-house, industrial and open source software systems. Our evaluation shows that the ArchDRH based clustering produces fewer extraneous modules and more authoritative modules can be recognized. ArchDRH-Split allows the user to find which classes participated in which patterns, and our industrial collaborator was able to identify several implementation errors by observing modules that depend on design rules that they shouldn’t.

We will further evaluate the effectiveness of ArchDRH architecture recovery techniques in large-scale legacy code written in non-OO languages. We will also compare them with other existing clustering techniques, such as concern-based clustering, and the possibility of combining them to achieve more accurate results. Another future work is to explore the possibility of using ArchDRH to identify implementation errors and architecture deviations.

ACKNOWLEDGMENTS

We thank our industrial collaborator, Mr. Baelen for providing the DSMs and authoritative clusterings of their project. We thank Dr. Xin Peng from Fudan University and Dr. Zhenchang Xing from National University of Singapore for providing the web-based information management system and sharing their opinions about our ArchDRH clustering. We thank Prof. Spiros Mancoridis’s Serg group, in particular, Mr. William Mongan, for providing support on the Bunch tool.

This work was supported in part by the National Science Foundation under grants CCF-0916891, CCF-1065189, CCF-1116980 and DUE-0837665.
REFERENCES


[10] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Nov. 1994.


