UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

VANIUS ZAPALOWSKI

Understanding and Recovering Architecture Rules

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Advisor: Prof. Dr. Ingrid Oliveira de Nunes
Coadvisor: Prof. Dr. Daltro José Nunes

Porto Alegre
March 2021

CIP — CATALOGING-IN-PUBLICATION

Zapalowski, Vanius

Understanding and Recovering Architecture Rules / Vanius Zapalowski. – Porto Alegre: PPGC da UFRGS, 2021.

143 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2021. Advisor: Ingrid Oliveira de Nunes; Coadvisor: Daltro José Nunes.

1. Software architecture. 2. Architecture rules. 3. Architecture recovery. 4. Module dependency. I. Oliveira de Nunes, Ingrid. II. Nunes, Daltro José. III. Título.

“If I have seen farther than others,
it is because I stood on the shoulders of giants.”
— SIR ISAAC NEWTON

ACKNOWLEDGEMENTS

During my Ph.D., I have had the pleasure of interacting with many great people. The ones that I own much are my advisors Professor Ingrid Nunes and Daltro Nunes for their patience, guidance, and support through the seven years—actually nine years including my master's degree– of my work. Also, I would like to thank the members of my examining committee, Professors Cláudio Sant'Anna, Ricardo Terra, and Érika Cota, for the valuable contributions, giving me feedback to improve my research.

I wish to express my gratitude to my parents, Irani and Walter. Without my parents' support, I would not get this accomplishment, neither the ones that will succeed.

My most sincere thanks to all my friend that had long talks and funny moments to relieve my mind during this seven years. I especially thank Paula that always supported me in my decisions. I must namely thanks my friends from Prosoft and Informatics Institute Borba, Biel, Jhonny, João, Serjão, Frederico, Fernando, Jacob, Carlos and Vinícius.

My thanks to the Informatics Institute that supported me in all the ways from my undergraduate to the end of my Ph.D.

Finally, I would like to thank all the people that the name is not in this text but have helped me in any way.

ABSTRACT

Software architecture is fundamental to clearly present the most important structures, roles, and rules of a software, which collectively are used to guide software design and implementation. Thus, the existence of reliable architecture documentation is critical to develop and maintain software in a controlled way. Despite the importance of architecture documentation, there are systems without proper documentation, as their documentation is often outdated or nonexistent. To help overcome this problem, many studies investigate how to maintain conformity between architecture documentation and the source code. However, most of these studies provide a low precision demanding an expert post-verification, which is an error-prone and time-consuming task, to provide useful architecture documentation. To support architecture documentation, we propose the Weighted-graph-based (WGB) method to recover architecture rules. Our method is based on the idea that high-level architecture rules can be derived through the investigation of source code dependencies, thus decreasing the effort of providing useful architecture rules with a reduced need for human verification. To achieve our goals, we investigate the source code dependencies and the architecture differences between them. Based on this investigation, we propose the WGB method that relies on the module dependency strength metric and linear equation solver to provide relevant architecture rules. It is domain-independent because it needs only the source code as information to execute. We evaluate our proposed method with a case study presenting details of how it works, an offline study presenting the application of our method in six subject systems, and a user study analyzing the rules extracted using our method in two commercial systems from the perspective of the developers. The results show that our method extracts useful and appropriate architecture rules using only the source code as information, thus supporting the task of recovering the architecture rules. Furthermore, the results of the user study present the preference of the developers for rules extracted using our method when compared against their manually recovered rules in most of the comparisons.

Keywords: Software architecture. architecture rules. architecture recovery. module dependency.

Compreendendo e Recuperando Regras de Arquitetura

RESUMO

A arquitetura de software é fundamental para documentar as estruturas, papéis e regras mais importantes de um software, que são usadas no projeto e na implementação. Assim, ter uma arquitetura documentada que reflete o que está implementado é importante para o desenvolvimento e evolução de forma controlada. Apesar da importância de ter a arquitetura documentada corretamente, é comum que os sistemas não tenham uma documentação arquitetural, ou que que ela esteja desatualizada. Para minimizar esse problema, muitos estudos avaliam como manter a conformidade entre a documentação arquitetural e o código-fonte. No entanto, a maioria desses estudos tem uma baixa precisão necessitando uma verificação dos seus resultados, que é uma tarefa demorada e que pode conter erros, para fornecer uma documentação útil. Para dar suporte à documentação da arquitetura, nós propomos o método *Weighted-graph-based* (baseado em grafos com pesos) para recuperar regras arquiteturais. Nosso método é baseado na ideia de que regras arquiteturais de alto nível podem ser derivadas através da análise das dependências do código-fonte. Isso reduz o esforço de documentar as regras arquiteturais por diminuir a necessidade de verificação de um especialista. Para atingir nossos objetivos, nós analisamos as dependências do código-fonte e as diferenças arquiteturais. Com base nessa análise, nós desenvolvemos o método *Weighted-graph-based* que se baseia na métrica de força de dependência entre módulos, também proposta nesta trabalho, e em um resolvedor de equações lineares para obter regras arquiteturais mais relevantes. Nosso método também é independente de domínio e específico para cada software visto que não depende de parâmetros e documentações. Nós avaliamos nosso método usando um estudo de caso para apresentar como o nosso método funciona, um estudo *offline* detalhando a aplicação de nosso método em seis sistemas e um estudo com usuários realizado para analisar o uso prático das regras extraídas usando nosso método em dois sistemas comerciais. Os resultados mostram que nosso método extrai regras arquiteturais que são úteis, eficientes e apropriadas. Além disso, os resultados apresentam a preferência dos desenvolvedores por regras extraídas usando nosso método quando comparadas com as regras recuperadas manualmente.

Palavras-chave: Arquitetura de Software, Regras Arquiteturais, Recuperação de Arquitetura, Dependência entre Módulos.

LIST OF ABBREVIATIONS AND ACRONYMS

A	Agreement
B	Beginning
C	Comparison
C-Rules	Number of Conceptual Rules
Cl	Number of Classes
Cl-Dep	Number of Class Dependencies
CONCEP	Conceptual Rule
E	End
GQM	Goal-Question-Metric
INTRA	Intra-module rule
KLOC	Kilo Lines of Code
M	Metric
Max	Maximum
Med	Median
MDS	Module Density Strength
Min	Minimum
MIS	Mismatch rule
MVC	Model-view-controller
P	Participant
Pkg	Number of Packages
Pkg-Dep	Number of Package Dependencies
RQ	Research Question
SUB	Specialization Rule
SUPER	Generalization Rule

SD	Standard Deviation
WGB	Weighted-graph-based

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Since the 1980s—when software architecture foundations began as a separate topic of study—to nowadays, when very large-scale systems are a reality, software architecture has gained attention in different computer science areas (SHAW; CLEMENTS, 2006). Researchers are mainly interested in investigating how software architecture increases the software quality and contributes to a controlled evolution. Furthermore, the software industry sees software architecture as a model that brings quality to its products, providing a controlled development. Both academia and industry have interest in software architecture because it provides fundamental information about the set of structures needed to reason about the system, such as module roles and how software elements should interact (BASS; CLEMENTS; KAZMAN, 2012b). Even when these roles and rules are unplanned or not explicitly documented, the system implementation has roles played by its elements and communication rules to define how its modules interact (PARNAS, 1994). Ideally, the first architecture of a system is planned and documented presenting its constraints, roles and rules. Along the system life-cycle, software changes as a consequence of its evolution, and these changes should be documented in the software architecture to maintain a controlled evolution of the software.

Regardless of the importance of having a documented architecture in conformance with the system's source code, changes may not be properly documented, causing a mismatch between the source code and the documented software architecture. This mismatch occurs as a consequence of two situations: (i) the documented architecture was not updated, which causes *lack of proper architecture documentation*; or (ii) source code disrespects documented architecture, which consists of *architectural violations*. In both cases, these undocumented changes may lead developers to introduce architectural violations (SILVA; BALASUBRAMANIAM, 2012) causing major problems to the structure of the system, to evolve the software architecture in an unplanned way (PERRY; WOLF, 1992), and to lack evidences of the reasoning of the previous architecture decisions (JANSEN; BOSCH, 2005).

Aiming to support the task of keeping the source code and the architecture in conformance, much research has been done on reverse engineering methods to (semi-) automatically reconstruct the software architecture (DUCASSE; POLLET, 2009). These proposed techniques and tools achieved significant results to support software architecture recovery. However, they still demand much human verification to provide a useful model, which makes them error-prone and time-consuming. Furthermore, most of existing approaches focus on recovering architecture modules and do not investigate architecture rules.

Given this context, in this thesis, we investigate how to support the recovery and conformance of architecture rules. To achieve this objective, we analyze *extracted rules*, which are rules extracted from the source code that may be architecture rules. Based on this analysis, we investigate the relationship of *implemented rules*, which are high-level rules that are present in the source code, and *conceptual rules*, which are those that are in developers' mindset, establishing characteristics to identify conceptual rules in the implemented rules. Additionally, we will also analyze the characteristics of architecture dependencies that violate software architecture. In the remainder of this chapter, we organize the main ideas of this thesis. First, we specify the addressed problem and detail limitations of existing work in Section 1.1. To tackle the problem, we introduce our proposed solution and give an overview of the contributions of this thesis in Section 1.2. Finally, in Section 1.3, we present the outline of this thesis.

1.1 Problem Statement and Limitations of Existing Work

This thesis aims at tackling three main challenges to keep an updated and reliable architecture documentation: (i) how to retrieve architecture rules based on the communication between architecture elements using solely source code dependencies; (ii) how to distinguish which implemented dependencies are in fact conceptual architecture rules; and (iii) how to support the identification of violations. Based on these three issues, we define our research question as follows.

Research Question. *How to extract architecture rules adopted in a software system based on the source code at the right level of granularity?*

Given our research question, we detail the two main challenges related to this research question. First, we present the limitations in the identification of architecture rules in Section 1.1.1. Second, we detail the current opportunities of research on the detection of architectural violations in Section 1.1.2.

1.1.1 Identification of Architecture Rules

Aiming to tackle the lack of architecture documentation, much research has been done on reverse engineering methods to (semi-)automatically discover the relationship between architecture elements. Many approaches (TZERPOS; HOLT, 2000; SANGAL et al., 2005; ZAPALOWSKI; NUNES; NUNES, 2014) focus on the extraction of source code information to group architecture elements by their similarities to compose architecture modules. Such approaches aim to derive architectural modules helping in the task of recovering modules. However, these approaches usually present low precision in their results (GARCIA; IVKOVIC; MEDVIDOVIC, 2013) and do not present architecture rules explicitly.

Alternatively, there are approaches addressing architecture conformance (TERRA; VALENTE, 2009; BISCHOFBERGER; KÜHL; LÖFFLER, 2004). These approaches rely on a well-documented set of conceptual rules to compare against the implemented rules. In most of these approaches, such as *reflexion models* (MURPHY; NOTKIN; SULLIVAN, 2001), the conceptual rules are written by an expert, which is error-prone because it depends on their subjective experience in the system. Furthermore, the time demanded to perform this activity is proportional to the system size. Thus, in large-scale systems, the task of specifying conceptual rules is frequently impractical due to time constraints and the constant evolution of the software.

Another problem related to the changes in the software is the software architecture evolution. The architecture evolves along the life-cycle due to changes in requirements, which demand significant modifications (NISTOR et al., 2005) in its modules organization and communication, such as remodularization. Thus, the software architecture evolves and, to have a controlled architecture evolution, always knowing the current architecture contributes to understand how to evolve it.

Based on the previous work on the recovery of software architecture and current associated problems, we state two main challenges to improve the scenario of the identification of architecture rules below.

i) *Support to the specification of architecture rules.* The effort needed to specify architecture rules is a significant barrier to the adoption of reverse engineering approaches. Reducing the effort needed to document such rules facilitates the task of keeping the architecture documentation updated. Consequently, it will improve the reliability of the documentation.

ii) *Analysis of source code dependencies.* An evaluation of which source code interactions are part of the conceptual architecture is needed to better comprehend the relationship between the source code and the software architecture. Using the source code as information leads us to a domain-independent analysis of conceptual rules.

These two limitations can be faced improving the recovery of the architecture rules implemented in the source code. The knowledge of the implemented architecture improves the documentation and helps maintain and evolve software architectures (conceptual and implemented).

1.1.2 Detection of Architectural Violations

Software architecture is a high-level representations of the source code . However, this model is usually not completely followed during the software evolution. As a consequence of the uncontrolled evolution, developers tend to include violations, which diverge from the software principles early planned. These violations are introduced by different causes, such as deadline pressure and conflicting requirements, but they degrade software maintainability (KNODEL; POPESCU, 2007). Thus, the process of frequently adding architectural violations transforms a well-organized software into an unmanageable block (SARKAR et al., 2009).

Violations in the software architecture are problems in software structure and communication that demand more effort to be fixed than common source code bug fixes due to the complexity and time to perform them. The detection of software architectural violations aims to tackle this kind of problem preventing major software problems to be propagated along software versions. With the detection of architectural violations, we have indications of high-level refactoring points to maintain a software well-organized along the evolution (BOURQUIN; KELLER, 2007).

Architectural violations may not have a relationship with source code quality problems because developers can provide low-level design that has quality, but their source code is not consistent with the conceptual architecture. Thus, architecture conformance approaches focus on detecting architectural violations based on comparing the conceptual and implemented architecture using a set of mapping rules. These approaches rely on this set of rules, which are usually written by system experts (DUCASSE; POLLET, 2009).

There is a variety of approaches to help developers to create these rules. However, these approaches still demand much manual effort.

An evaluation of extracted dependencies allows the detection of architecture patterns used in the project scope based on rule similarly and quality. Moreover, this kind of evaluation may reduce the effort needed to detect architectural violations and undocumented architecture rules. Based on the evaluation of extracted dependencies, we can derive three types of rules: (i) documented conceptual rules; (ii) undocumented conceptual rules; and (iii) rules that are violations. These three situations of extracted dependencies should be investigated to provide more detailed information about the implemented architecture rules provided in an architecture rules recovery approaches.

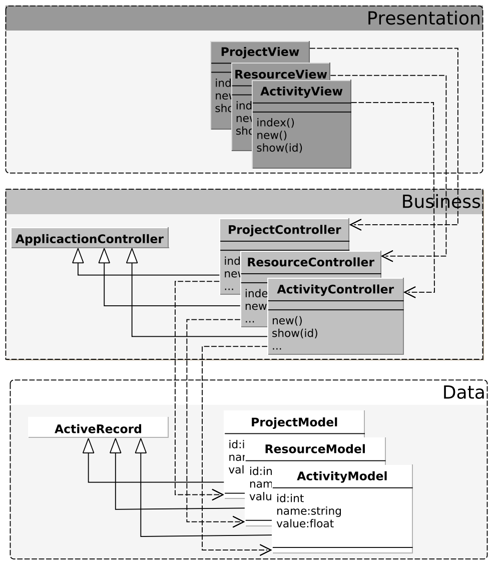
1.2 Proposed Solution and Contributions Overview

Given the limitations described in the previous section, we present our proposed solution. The underlying idea is to improve the reliability and conformance of architecture documentation, providing a method to support the recovery of architecture rules, which is domain-independent. Our method identifies architecture rules based on frequent dependencies implemented in the software source code and their abstraction level according to the source code structure. The main advantage of our method is that it needs neither a knowledge base nor the prior documentation of the software architecture. Based on this idea, we next present our research hypothesis.

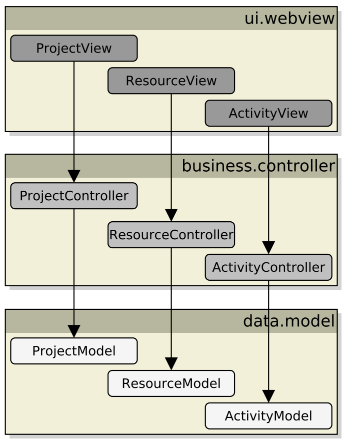
Research Hypothesis. *By analyzing patterns of dependencies and clustering them, it is possible to derive architecture rules that are considered at an adequate level of granularity.*

To explain our proposed solution, we take, as an example, the layered architecture (HENNEY; SCHMIDT; BUSCHMANN, 2007) widely used to implement web systems. In this kind of architecture, the software is structured in layers. We assume three layers in our example: *Presentation*, *Business* and *Data*. The main rule of this architecture is that its modules can only depend on modules immediately below them, as illustrated in Figure 1.1a. The architecture presented in this figure has two explicit architecture rules $Presentation \rightarrow Business$ and $Business \rightarrow Data$; and implicit rules that every other communication is prohibited. In the source code, the architecture elements are organized

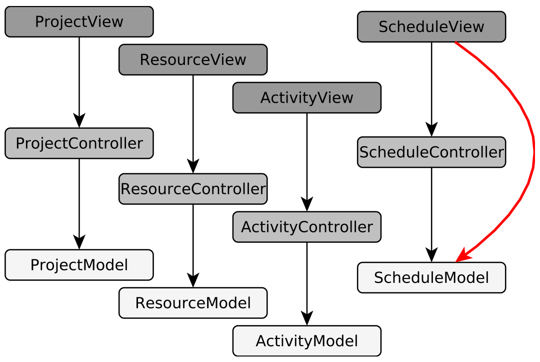
Figure 1.1: Implemented Rules Identification and Violation of Implemented Rules.



(a) 3-layer Architecture Pattern



(b) Dependency Subgraphs



(c) Violation

in structures, such as packages or folders, that are different from architecture modules. These source code structures represent groupings that are more fine-grained than architecture modules.

To extract architecture rules from the source code, we assume that source code elements that belong to the same module are in the same source code structure. For instance, in Figure 1.1b, `ProjectModel`, `ResourceModel` and `ActivityModel` elements are grouped in the package `data.model`, which corresponds to the *Data* module in the source code, and they depend mainly on the package `business.controller`, which corresponds to the *Business* module. Based on these source code dependencies, we can derive extracted rules, which are commonly fine-grained abstractions. Therefore, to recover conceptual rules, this thesis aims to investigate the relationship among extracted rules, implemented rules and conceptual rules. We analyze characteristics of extracted rules to propose a method that recover architecture rules at an adequate level of granularity.

1.2.1 Contributions

The main contribution of this thesis is *a method to automate the documentation of software architecture identifying architecture rules*. Its specific contributions are presented next.

i) *An empirical evaluation of the conformance between architecture documentation and the source code* of software systems, which provides evidence of the gap between them based on the characteristics of architecture rules and source code dependencies.

ii) *A system-specific metric* to measure the dependency strength and granularity level of source code dependencies to provide support to find the most appropriate level of granularity to represent architecture rules.

iii) *An offline evaluation* presenting the efficiency and effectiveness of our proposed method comparing the architecture documentations obtained using our method and the conceptual architectures of the systems under analysis.

iv) *A user study* that evaluates our proposed method based on the perspective of the developers providing a detailed analysis of their opinion about the extracted architecture rules.

1.3 Outline

This thesis is organized as follows. In Chapter 2, we discuss the background and related work. Next, in Chapter 3, we present an investigation of the architecture rules to analyze the difference in the abstraction level between architecture rules and source code dependencies. In Chapter 4, we describe our method, named Weighted-graph-based (WGB), that automatically recovers architecture rules from source code. Then, in Chapter 5, we provide three evaluations of the WGB method. Finally, we provide, in Chapter 6, the conclusions of our investigations.

2 BACKGROUND AND RELATED WORK

Every implemented software system has an *software architecture*, which structures the system in terms of pieces of software (i.e. *architecture modules*) split according to some criteria and dictates *architectural rules* that specify allowed dependencies among them, such as when classes can depend on each other (BASS; CLEMENTS; KAZMAN, 2012a). This information should govern the development of software systems, making their implementation in accordance with the specified architecture for a healthy software evolution. Ideally, modules should be made explicit in the source code, e.g. by means of packages, which contain finer-grained elements (e.g. classes) that depend only on elements that they are allowed to, according to rules.

Despite the importance of software architecture, many systems do not have reliable architecture documentation, because it is outdated or nonexistent, leading to *architecture violations*, which are dependencies in the source code that are not allowed by the architecture. These violations contribute to *architecture erosion* (PERRY; WOLF, 1992), which involves the introduction of undesired dependencies among modules. Furthermore, while evolving the source code, developers may change the software architecture without updating its documentation leading to *knowledge vaporization* (JANSEN; BOSCH, 2005), which consists of not documenting the decisions to consult in the future. Another problem related to the lack of architectural documentation is the *architectural drift* (PERRY; WOLF, 1992), which is the gradual divergence between the conceptual architecture and the implemented architecture along the evolution of the system. Therefore, keeping the source code in conformance with the software architecture with updated documentation is fundamental to avoid these problems.

In this thesis, we address three main problems related to having a useful and reliable software architecture: (i) maintaining conceptual and implemented architecture in conformance; (ii) the recovery of conceptual rules based on evaluations of extracted and implemented rules; and (iii) the detection of architectural violations. Therefore, we present these problems and approaches related to them pointing out their strengths and weaknesses. In Section 2.1, we present work on architecture recovery that focuses on mining architecture information. In Section 2.2, we introduce the approaches to keep architectures in conformance. Next, in Section 2.3, we discuss methods related to the detection of architectural violations. Finally, in Section 2.4, we provide our final remarks of the problems presented and the related work discussed in this chapter.

2.1 Architecture Recovery

Architecture recovery is the process of reverse engineering an implemented software system to obtain or support having an architecture of this system (DUCASSE; POLLET, 2009). Research on architecture recovery has been developed to handle the lack of architecture documentation retrieving information present in the source code (semi-) automatically. Aiming to have an architectural structure of the systems, dependency-based approaches investigate similarities among architecture elements to identify modules based on metrics, such as coupling and cohesion. Complementary, there is dependency-based work that focuses on recovering architecture rules of conceptual architectures. Then, in this section, we present work related to architecture recovery divided into two categories. In Section 2.1.1, we describe work related to the identification of software architecture modules. Then, in Section 2.1.2, we present studies that aim to recover architecture rules.

2.1.1 Architecture Module Approaches

The main objective of the *recovering of architecture modules* is to analyze the system elements to group them into architecture modules (DUCASSE; POLLET, 2009). Many approaches have been proposed to group elements of the system based on different aspects of the system elements, such as similar dependencies, names, architecture styles, and architecture patterns. The result of the module recovery approaches is a classification of the source code elements into modules.

One of the first tools developed to extract dependency similarities among architecture elements to find architecture modules is the Bunch tool (MANCORIDIS et al., 1998). It specifies an objective function to the software decomposition problem. This tool uses a modularization quality metric, also proposed by the authors, applying clustering techniques to organize architecture elements in modules. The proposed modularization metric is based on an architecture module analysis of the intra-connectivity (when an element of a module depends on an element of the same module) and the inter-connectivity (when an element of a module depends on an element of another module) of the system modules. Then, to decide to which module each element belongs, Bunch iteratively applies a genetic algorithm to all architecture elements calculating the intra-connectivity and inter-connectivity of architecture modules in each iteration. Based on the mutations made by the genetic algorithm, it evaluates the prediction using the quality metrics and keeps the solution with the best result. Constantinou, Kakarontzas and Stamelos (2011) proposed a method that is similar to the Bunch tool. Their method groups architectural elements using a clustering algorithm based on source code dependencies and metrics. The main difference between their work and the former is that Bunch has a quality threshold to determine the number of modules, while the latter fixed the number of architecture modules to be discovered. In their evaluations, both methods produce more architecture modules than the conceptual architecture during the clustering phase. Then, to reduce the number of modules, a hierarchical clustering algorithm is applied to merge similar modules based on their degree of similar intra-connectivity dependencies.

To enrich the source of information to group elements, Corazza et al. (2011) and Belle, Boussaidi and Kpodjedo (2016) proposed methods based on the lexical information elements. Corazza et al. (2011) evaluate the parts of the source code and how they affect the recovery of modules. Belle, Boussaidi and Kpodjedo (2016) developed a method to recover architecture modules assuming that the elements of a module use the same set of words and theses elements should be grouped to form a module. Additionally, Belle, Boussaidi and Kpodjedo (2016) method has a refinement of these modules built based on the lexical similarity using source code structures, e.g., packages, to infer a layered architecture.

In general, these methods provide inaccurate results because they analyze only the architectural roles ignoring further information that can be extracted from the code, such as architecture rules. Furthermore, these methods recover the architecture of software that follows a hierarchical architecture, such as layered patterns (HENNEY; SCHMIDT; BUSCHMANN, 2007). Thus, software implementing non-hierarchical architecture patterns, such as Model-View-Controller (MVC) (BASS; CLEMENTS; KAZMAN, 2012a), are incorrectly recovered using these methods. Consequently, the architecture rules derived from these approaches are inaccurate too.

Looking for complementary information related to the architecture dependencies to improve the precision of recovered architectures, Xiao and Tzerpos (2005) investigated the extraction of dynamic dependencies at runtime. In their study, they investigated architecture recovery by applying different clustering algorithms using static and dynamic dependencies. In addition, they ran experiments varying the degree of element dependencies to reduce the search scope of the dependency graphs. They concluded that the dynamic dependencies provided a better model using different weights according to runtime dependencies between elements. However, the extraction of runtime dependencies relies on the exercised zones of the source code, i.e. only the system parts activated during runtime execution are evaluated. Then, the data collected is proportional to the activation of system parts, i.e. their approach is limited to the exercised parts of source code. Furthermore, they did not report a coverage analysis of their experiments.

To comprise a complete software coverage and to understand module roles, we, in a previous work (ZAPALOWSKI; NUNES; NUNES, 2014), exploited the usefulness of adopting a set of code-level characteristics to group elements into architecture modules. We evaluated the relationship between different sets of characteristics (source code metrics and object-oriented properties) and the accuracy achieved by an unsupervised learning algorithm for the identification of architecture modules. By the use of code-oriented information, our previous approach achieved a significant average accuracy evaluating the prediction of architecture modules, which indicates the importance of the selected information to recover software architecture. However, the identification of modules is a sub-problem of recovering the whole software architecture (modules and rules). Then, our previous approach lacks information about architecture rules to provide a more useful architecture documentation.

Recently, Kong et al. (2018) proposed a technique to improve the information provided by the source code dependencies based on a directory-based metric to recover architecture modules. Their metric is based on thresholds to determine these (sub-)modules analyzing inter-coupling and intra-coupling metrics to group files and directories into (sub-)modules. Their method is applied before the well-known recovering methods to reduce the number of elements analyzed to compose modules. Their results present an improvement in the module classification but still are relatively low, ranging from 24% to 69% depending on the threshold and metrics used to measure.

Given the variety of sources of information and results reported, Garcia, Ivkovic and Medvidovic (2013) presented a standardized comparison of six architecture recovery techniques using three metrics to measure differences between software architectures. They concluded that ACDC (TZERPOS; HOLT, 2000) and ARC (GARCIA et al., 2011) perform better than the other techniques. Both these approaches exploit the similarities in architecture element dependencies and textual information by evaluating two different aspects. On the one hand, ACDC proposes seven patterns based on expert knowledge providing a more general approach to group architecture elements into modules. On the other hand, ARC performs a statistical evaluation of individual projects using information retrieval and graph similarity techniques to provide a more narrow classification of elements. Although these two approaches achieved the best precision compared to the other four techniques, Garcia, Ivkovic and Medvidovic (2013) reported that all techniques still provide poor precision. It occurs due to the complexity to group architecture elements without domain-specific knowledge, such as the number of modules a software has and their roles. In a more recent comparison, Lutellier et al. (2015) investigated the effect of accurate dependencies on the metrics proposed to recover architecture modules. They concluded that the quality of the modules recovered varies according to the system size and the granularity of the dependencies used to recover architecture modules. Furthermore, they presented limitations, such as the mismatch and non-conformance of the architectural elements and source code elements of the analyzed methods.

In general, studies that recover architecture modules provide a high-level structure of the architecture organization and many of them use source code dependencies to generate their architecture organization. However, they do not provide information about the rules that architecture elements should respect. Consequently, the architecture comprehension gain is limited due to the need for manual investigation of elements to know each module communication rules. Additionally, violations of communication rules result in an inaccurate grouping of elements because they drift or erode the architecture (MEDVIDOVIC; TAYLOR, 2010). The violations induce the recovery techniques to classify elements in wrong modules considering just the architecture dependencies to recover modules.

2.1.2 Architecture Rule Approaches

The *recovery of architecture rules* aims to derive high-level rules based on the analysis of source code dependencies grouping them according to some criteria (ZAPALOWSKI; NUNES; NUNES, 2018). The problem of recovering architecture rules rises because it is complex to examine all source code dependencies of large-scale system and decide which dependencies should be presented, omitted, or merged together in an architecture documentation. Considering that a large-scale software is complex and evolves quickly, architecture documentation is rarely updated. This updating process should be done frequently and fast to keep track of the changes of the source code in the architectural level. Therefore, approaches to recover architecture rules aim to extract source code dependencies and analyze their occurrence in the software. For this purpose, these approaches mine for frequent communications to establish architecture rules.

Few approaches focus on recovering implemented rules, as ours. One of the first studies based on this idea is a tool called PR-Miner, proposed by Li and Zhou (2005). They analyzed the dependency between functions and variables in procedural programming languages. Based on the element dependencies, they mined the frequent itemsets of the function and variable calls to establish implicit code rules. They evaluated the rules found by their method using three large-scale systems considering, the confidence, support and size of the frequent itemsets. Their results suggest that the retrieved rules are relevant to software organization, but these rules are at the code-level. Due to this low-level of abstraction, their approach provides too many rules. PR-Miner lacks abstraction in the retrieved rules because it found 6K implicit rules in a system with 381K of lines of code, i.e. one rule to each 62 lines of code. Thus, architects still must verify manually 6K rules and their relevance to the software architecture. To improve the usefulness of PR-Miner, the number of rules extracted should be refined.

Using a similar idea of PR-Miner, Hora et al. (2013) proposed an approach to mine system specific rules from change patterns. Their work focuses on the analysis of patterns along the software evolution considering the changes in application programming interface (API) calls in different releases of a system. They compared the relevance of specific rules based on the recurrent changes considering just one software at a time against generic bug-fix rules, such as a suggestion of good practices or reduction of method complexity. Evaluating a closer scope, they found that specific rules are more relevant than general rules because the information provided by specific rules is more contextualized than that of generic rules. The fundamental idea of their approach is to analyze a narrowed scope to provide more useful information. However, the provided specific rules are related just to API communication. Then, the rules extracted are related to how to communicate to external parts of the software and lack an analysis of the communication between internal modules. It is important to evaluate internal rules because software architecture erodes mainly based on the violations added to the internal modules.

Focusing on the lack of evaluation of internal rules, Maffort et al. (2013b) presented a data mining approach for architecture conformance based on a combination of static and historical software analysis. Their approach aims to identify architectural violations, but, to do that, they need the architecture rules to perform an architecture conformance check. Then, they divided their methodology into three main components: *Code Extractor* is a source code parser; *Architecture Miner* mines the architecture rules according to their frequency in a project; and *Violation Detector* matches the architecture rules provided by the Architecture Miner against the source code. Despite their focus on the architectural violations, the main contribution of their work is the statistical evaluation of the architecture rules mined in the Architecture Miner component. The assessment of rules is important to their approach because they are the basis of the architecture conformance check. Despite the relevance of their assessment, the mining process performed could be better exploited considering an evaluation of individual architecture modules instead of the entire software. A key limitation of this approach is that it requires the specification of a threshold to define when the frequency of dependencies is high enough to be considered a rule. This threshold is essentially the manual definition of which dependencies correspond to rules.

There are also approaches that rely on the Dependency Structure Matrix (DSM) (HUYNH et al., 2008; PAIVA et al., 2016; SANGAL et al., 2005; WONG et al., 2011; MO et al., 2015), which captures source code dependencies. The visualizations provided by DSMs ease the analysis of dependencies that occur in the source code to identify implemented architecture rules. However, this identification process is manual, thus demanding significant effort, mainly in large-scale software systems.

2.2 Architecture Conformance

Given that we have a documented architecture, a verification is needed to check whether the conceptual architecture is matching the implemented architecture. So, a process that matches the conceptual architecture against the implemented architecture verifies the conformance, process that is referred as to *architecture conformance*. Architecture conformance approaches require the specification of the software architecture to compare it with the implementation. Some of which include sophisticated means of specifying architectural modules and rules. One important premise of these approaches is the need for a conceptual architecture, which is not commonly found.

Reflexion models (MURPHY; NOTKIN; SULLIVAN, 1995) is a technique that represents the fundamental concept of architecture conformance. To check the conformance of a conceptual architecture against an implemented architecture, reflexion models requires a list of rules, which map architecture elements to source code elements that should exist, as input. Reflexion models try to match the architecture elements of the list with source code based on this mapping list. To perform this matching of rules, the Murphy, Notkin and Sullivan (1995) described two basic entity types: an *high-level model*, which is parsed to a high-level model entity (`HLMENTITY`) format; and *source code*, which is parsed to a source code model entity (`SCMENTITY`). They defined a format to state the mapping rules, which is similar to a regular expression. After submitting the entity models to the matching process, it returns a high-level model and three variables that report *convergence*, *divergence* and *absence* of the rules given. Each variable has an important meaning in the process: *convergence* indicates a correct matching between high-level and source code entities; *divergence* describes when a source model does not match the high-level model; and *absence* reports when a high-level dependency is missing on the source code. With the output of the matching process, it is possible to analyze the results and change the implementation or update the architecture to achieve a better consistency between them.

Aiming to reduce the manual effort needed to apply reflexion models during an architecture conformance process, the Human-Guided Mapping Generation Method (HuGMe) (CHRISTL; KOSCHKE; STOREY, 2005; CHRISTL; KOSCHKE; STOREY, 2007) algorithm was proposed to assist software architects in the task of mapping conceptual elements to implemented elements. HuGMe is an algorithm to bind `HLMENTITY` and `SCMENTITY` based on similarities between entity dependencies. By the results obtained, HuGMe reduces the effort needed to match the elements considering software that is in conformance between conceptual and implemented architectures. However, software with many architectural violations or in disagreement with the conceptual architecture tends to produce poor matching because the conceptual and implemented elements have different dependencies and, consequently, the algorithm does not bind them correctly.

Bittencourt et al. (2010) proposed another approach to ease the effort of stating the mapping rules of reflexion models. Their approach relies on the textual similarities of conceptual and implemented elements facing design documentation and source code files as text information. They use information retrieval techniques to associate the elements based on a predefined threshold of similarity needed to establish a relationship between implemented and conceptual elements. This threshold is a key point of their approach because a relaxed threshold produces a general bind where the elements are almost one to one binding, and a strict threshold produces few bindings. Analyzing the textual information, they depend on the use of textual conventions in both implemented and conceptual documents. Furthermore, the architecture view produced is driven by the textual terms used. For instance, when analyzing a point of sale software structured in Model-View-Controller architecture, their approach tends to present an architecture organized in domain words, such as User, Sale and Payment, instead of modules with independent roles, such as Presentation, Business and Data.

As an alternative to reflexion models, Terra and Valente (2009) proposed a domain-specific language, called DCL (Dependency Constraint Language), to conformance check context. They developed a tool, named DCLcheck, that implements their language. This tool focuses on providing conformance mapping of object-oriented concepts, such as packages, classes and methods, to the binding process between conceptual and implemented architecture elements. One of the main contributions of these approaches is the standardization of how to bind source code and the conceptual architecture of different granularities. The improvement of their binding process is that it supports conformance of systems written in object-oriented paradigm and has a simple syntax providing a variety of dependency types. For instance, we can specify not only allowed dependencies (can) but also prohibited dependencies (cannot). Schröder and Riebisch (2017) also proposed an approach to architectural conformance. Their approach is based on the description logics aiming to provide an ontology to the architectural conformance process. They provide a set of definitions written in SROIQ to form a ontology that should be used to recover architectures. The main difference from Terra and Valente (2009) is that Schröder and Riebisch (2017) set of definitions may be extended using the SROIQ language.

Despite their contributions, one main limitation of these approaches is the need for a high-level model to extract the architectural rules and parse them to their specific language. Having a conceptual architecture properly documented is a challenge due to many problems that may occur during the software development, which causes architecture erosion and drift. In both approaches, if we do not have an architecture model well-documented and updated, the architecture rules should be created based on a manual investigation of the code or the architects knowledge of the software. Both activities are time-consuming and error-prone, considering thousands of lines of code and complex software structures. Consequently, the applicability of this kind of approach is limited due to the lack of reliable architectural rules.

2.3 Architectural Violation Identification

Software architectural violations are implemented elements that are not allowed by the documented architecture (MURPHY; NOTKIN; SULLIVAN, 1995), such as a dependency between two modules that is not listed in the set of rules of the documented architecture. Many violations are evidence of architecture erosion. The mismatching between conceptual and implemented architecture decreases the maintainability and increases the cost to evolve software. Thus, to avoid these problems, much work has been done to facilitate the task of detecting architectural violations. The detection of architectural violations without a set of specified architectural rules is a challenge because it is complex to discover which dependencies are not allowed without knowing which dependencies are allowed in the architecture. To detect architectural violations, studies investigate source code characteristics of architectural violations to relate or predict architectural violations based on these characteristics.

Brown et al. (1998) provides a catalog of architecture anti-patterns. Based on their experience on handling different systems, they documented recurring solutions that are applied, but decrease software quality at an architecture level. Another similar idea is the architecture smells (GARCIA et al., 2009). These are also recurrent problems at an architecture level, but they are usually related to trade-off decisions that may lead to an architecture problem. The architecture smells help to find design-level opportunities to improve the architecture quality. Both these approaches are related to architectural violations because both investigate architecture flaws using architects' past experiences. Mo et al. (2019) propose a catalog of architecture anti-patterns based on their empirical investigation. Their definition of anti-patterns are highly dependent on the thresholds of each pattern. Therefore, it requires a calibration, which is system specific, to identify anti-patterns. Commonly, these catalog approaches report general architecture problems and, to provide a more useful detection of architectural violations, a specific investigation is needed or a set of threshold should be defined.

Macia et al. (2012a) investigated the relationship between architecture problems and source code problems. Their first study focused on establishing the effect of architectural violations in the source code. They performed an empirical study evaluating the origin of architecture problems. They concluded that 78% of the architecture problems were generated by code anomalies. Additionally, they analyzed system evolution and, even in the system with frequent refactoring activities, the code anomalies persisted during many versions. Their second study (MACIA et al., 2012b) evaluated the opposite direction of the relationship, i.e. if all code anomalies have a relationship with architecture problems. They investigated automatically-detected code anomalies and their effect in higher level models, such as software architecture. Their experiments evidenced that the techniques used to detect code anomalies automatically must be improved to detect anomalies related to architecture problems, since their results showed that most of the code anomalies detected are not related to architecture problems.

Assuming that architectural violations occur with low frequency and are specific to each software, Maffort et al. (2013a) performed a study that proposed heuristics to discover architectural violations. They implemented a prototype tool called ArchLint to evaluate the effectiveness of their four heuristics. The heuristics idea is similar to architecture smells proposed by Garcia et al. (2009), because both specified the smells or heuristics based on recurrent experience stated by software architects that usually led to violations. The difference between these approaches is that the heuristics can be calibrated with a set of thresholds to refine the detection of architectural violations instead of having fixed structures matched to the code. Furthermore, ArchLint needs a conceptual architecture to perform the detection of architectural violations. Thus, ArchLint assists software architects in the task of detecting architectural violations but only if: (i) architects have knowledge about the system architecture to state a conceptual architecture and set proper thresholds; (ii) the system under evaluation is partially eroded, in a system with many violations, the violations are frequent; and (iii) the system have the violations specified in the heuristics. An extension of ArchLint was proposed to help architects in the task of adjusting the thresholds (MAFFORT et al., 2015). They provide an iterative method to guide the architect to find a proper value based on the number of false positives detected by ArchLint. ArchLint extension helps architects to set the thresholds, but the decision if the threshold is appropriate is still based on the architect knowledge.

2.4 Final Remarks

In this chapter, we presented the related work to architecture conformance, architecture recovery, and architectural violations analyzing their strengths and weaknesses. Much research has been done to improve the results obtained in these topics, but there are relevant challenges to be faced to apply the proposed methods. Most of the problems arise from inaccurate or insufficient architecture documentation, which demands the reverse engineering of architectures. Due to the poor results provided by recovery methods, they assist architects in a manual investigation. Furthermore, the poor results of these methods are a consequence of the lack of information about the rules that architecture elements should respect and lack of an analysis of the communication between internal modules. Assuming that architecture recovery provides useful, but not reliable, results, architects perform an architecture conformance to check differences between implemented and conceptual architectures. To check the conformance of these two different views of the software, architecture rules should be stated. This specification of rules is usually based on a manual investigation of the code or expert knowledge of the software making this task time-consuming and error-prone. A misunderstanding in architecture conformance or software development could lead to a violation in the conceptual architecture. Architectural violations are complex to detect due to the lack of architecture documentation and limited evaluation of architecture rules implemented in the source code.

In order to provide a better understanding of the fundamental differences among existing approaches, we summarize them in Table 2.1, highlighting their required input and output, and, in Table 2.2, the performed analysis. Architecture conformance approaches and module identification approaches are grouped in the first and second rows of this table, respectively. The remaining rows refer to approaches that focus on architecture rules.

Table 2.1: Related work input and output

Approach	Input	Output	
		Type	**Description**
Reflexion Models and DSLs	architecture modules architecture rules Code dependencies	Architecture absences Architecture compliance Architecture divergences	A list of architecture rules and code dependencies, each with a label to indicate if it is a case of absence, compliance, or divergence.
Module Identification	Code dependencies Code metrics Hierarchical module structure Lexical information	architecture modules	A mapping from code elements to architecture modules indicating to which module they belong.
DSM	Code dependencies Design rules Hierarchical code structure	architecture modules Architecture divergences Matrix visualization	A visualization of the dependencies highlighting the divergences.
PR-Miner	Function calls	Code-level rules Violations in function calls	A list of rules with support values and a list of violations based on the violations of the rules found.
Hora et al.	Code dependencies CVS repository	Code-level Rules	A set of system-specific rules regarding API usage.
Architectural Miner	architecture modules Code dependencies CVS repository Thresholds	Architecture Absences Architecture Divergences	A list of architecture absences and divergences according to the identified patterns.
WGB Method	Code dependencies Hierarchical module structure	architecture rules	A set of architecture rules.

Table 2.2: Related work analysis

Approach	Analysis	
	Type	**Description**
Reflexion Models and DSLs	Conformance checking Clustering algorithms to modules	Verification of compliance between the architectural specification and the implemented code to check their divergences. Some approaches rely on clustering algorithms to assist the specification of modules.
Module Identification	Clustering algorithms	Algorithms that cluster source code elements into modules using different types of information.
DSM	Conformance checking	Conformance checking of the dependencies and structure of the source code according to design rules. A matrix representation of dependencies supports the specification of design rules based on the hierarchical organization of the code.
PR-Miner	Association rule algorithm Comparison of support values	Use of an association rule algorithm to find frequent itemsets using the function call paths, which are the sequences of calls in function.
Hora et al.	Frequent patterns Historical changes	Mining of similar refactorings over the system evolution related to API usage.
Architectural Miner	Association rule algorithm Conformance checking	Analysis of frequent structural or historical actions. Actions whose frequency is above a threshold (informed by an expert) are patterns. This is used for performing a conformance check.
WGB Method	Dependencies clustering Hierarchical clustering Linear equation solver	Use of the Module Dependency Strength (MDS) metric to prune dependencies and specify a linear equation problem to discard redundant dependencies.

3 A STUDY OF THE GAP BETWEEN ARCHITECTURE RULES AND IMPLEMENTED MODULE DEPENDENCIES

Evaluations of approaches and tools that assess the matching between a conceptual software architecture and its implementation show that divergences typically occur. However, there is limited investigation of the nature of these divergences. Such an investigation can reveal underlying problems, e.g. the use of inadequate granularity to document the architecture. In this chapter, we evaluate and characterize the divergences between conceptual architecture rules and dependencies among modules implemented in the source code, potentially reflecting implemented rules. This was done by means of a study involving six subject systems, in which we extracted source code dependencies and compared them against architecture rules using an association rule algorithm that provides a metric based on frequency. Thus, we detail the settings of this study in Section 3.1. Next, we present, analyze and discuss the results obtained during the study in Section 3.2.

3.1 Study Settings

The previous chapter introduced many approaches that aim to support software architects to document architectures and keep them consistent with the implemented code. However, as discussed, the gap between conceptual architecture rules and implemented module dependencies is large. In this Section, we analyze this gap to understand why it occurs, so that we can identify whether there are problems associated with the way conceptual rules are documented and also to understand properties associated with module dependencies that help automatically recover rules. Our analysis is done based on an empirical study described in this section.

3.1.1 Goal and Research Questions

We adopted the widely used Goal-Question-Metric (GQM) (SOLINGEN et al., 2002) paradigm to design our study. We thus structured it with a goal statement, research questions to achieve this goal, and metrics that provide the necessary information to answer our research questions. We next state our study goal.

To *understand why there is a (large) gap between conceptual architecture rules and implemented module dependencies*, *evaluate and characterize their divergences* from the point of view of *the researchers* in the context of a *multi-project study.*

To achieve our goal, we formulated three research questions (*RQs*) considering the previously discussed challenges, and selected metrics (*M*) to answer them, as shown next. We detail the metrics in Table 3.1 associating to which RQ they are related according to the column Research Question, e.g. M1 in related to RQ1 in the Table 3.1.

RQ1. What is the gap between conceptual architecture rules and implemented module dependencies?

RQ2. How can implemented module dependencies be categorized in relation to conceptual architecture rules?

RQ3. Are implemented module dependencies distinguishable considering their categorization?

Table 3.1: Empirical Study Metrics.

Metric	**Research Question**	**Description**
M1	RQ1	Architecture conformance: fraction of implemented module dependencies that are in conformance with architecture rules.
M2	RQ1	Rule conformance: fraction of allowed dependencies that are implemented
M3	RQ2	Number of implemented dependencies by identified category.
M4	RQ3	Support of implemented dependencies.

Although it is known that implemented software often does not strictly follow its conceptual architecture, having many more dependencies than planned (HORA et al., 2013)—and that is the motivation for architecture conformance approaches—with RQ1, we aim to identify *how large* is the gap between architecture rules and implemented dependencies. While RQ1 focuses only on analyzing the conformance level, our goal with RQ2 is to provide a detailed analysis of the dependencies and classify them with respect to rules. This classification allows us to understand the types of divergences that occur between rules and dependencies, which can help identify missing information in architecture documentation. Finally, with RQ3, we focus on verifying whether identified types of divergences have particular properties that characterize them, so that it is possible to (semi-)automatically distinguish them. For example, being able to distinguish dependencies that are architecture violations from undocumented architecture rules.

3.1.2 Procedure

In this section, we describe the steps of the procedure adopted to perform our study. In a nutshell, we first recover the conceptual architecture of our subject systems. Next, we extract dependencies among classes, which are used to build a dataset to obtain the module dependencies using an association rule algorithm (that gives M3, the support metric). This extraction of dependencies is independent of the manual architecture recovery. Recovered rules are then matched against extracted dependencies. We provide details of each of the steps of our study procedure as follows.

Manually Recover System Architecture. We recovered the architecture of our subject systems based on the guidelines presented by Garcia et al. (2013). The foundation to the recovery process consists of previous architecture documentation and the knowledge provided by information providers (software architects and developers) of the systems. We first identified software modules by identifying packages (considering the Java language) of a particular level of the package hierarchy of the system that correspond to architecture modules. For example, a package named `data` has different sub-packages such as `data.dao` and `data.hibernate`. Our information providers informed that the *data* package is an architecture module. As a consequence, sub-packages are sub-modules. Another consequence of specifying `data` as a module is that all dependencies between sub-modules of `data` are allowed in our recovered architecture. Next, information providers stated conceptual rules that inform whether modules can depend on elements of another module. For example, if a *Presentation* module depends on a *Business* module, classes of the former (or of sub-modules) can, e.g., invoke methods of classes of the latter (or of sub-modules). We represent this rule as *Presentation* $\rightarrow$ *Business*. In some cases, we had available architecture documentation (mainly for the large subject systems) and in others we drafted modules and rules, and validated them by consulting the system architects and developers to reduce the effort needed from them. As a result of this task, we obtained modules and architecture rules, which comprise the architecture of our subject systems.

Extracting Module and Class Dependencies Based on the source code of each subject system, we extract dependencies using the Classycle[1] tool, which automatically extracts dependencies between classes for each system class. The extraction of dependencies considers any type of dependency between classes, which in Java means that a class depends on all classes that it imports. Based on the graph built considering these dependencies, we derived module dependencies, considering to which module each class belongs.

Calculating Support. In order to analyze dependencies, we investigate the frequency in which classes of a module depend on classes of another. This analysis considers the specific module to which a class belongs and also its parent modules. For example, consider a class c_A located in the child module A_C, part of the parent module A_P. c_A depends on the class c_B, located in the child module B_C, part of the parent module B_P. Therefore, we investigate the dependency considering four perspectives: $A_P \rightarrow B_P$, $A_P \rightarrow B_C$, $A_C \rightarrow B_P$ and $A_C \rightarrow B_C$. If there are more modules in the module hierarchy, we consider classes as part of any module. These different perspectives allow us to evaluate all possible representations of architecture rules with different granularity levels.

To investigate the dependency frequency, we calculate the percentage of classes within a source module that depends on classes of a target module, considering all possible perspectives described above. This calculation was made by means of the Apriori, an association rule mining algorithm (TAN; STEINBACH; KUMAR, 2005), more specifically, the GNU-R[2] implementation. The *support* metric given by this algorithm, using a particular setup, corresponds to dependency frequency. For example, if a module *A* has 100 classes from which 75 depend on classes of a module *B*, the dependency $A \rightarrow B$, in this case, has a support of 75%.

Analyzing Extracted Dependencies. To answer our research questions, we analyze module and class dependencies and compare them with architecture rules obtained from the manual architecture recovery. This comparison is made using the metrics specified in the previous section.

[1]<http://classycleplugin.graf-tec.ch>
[2]<https://CRAN.R-project.org/package=arules>

3.1.3 Subject Systems

We selected six subject systems to be investigated in our study. They were selected because they matched the criteria of having available source code and a recoverable or updated architecture documentation. In addition to these criteria, variations on the nature of the software, such as adopted architectural patterns, domain and size, was important to have representative software systems.

Table 3.2 and 3.3 summarize key characteristics of our selected systems, and further information can be obtained in Appendix A. In Table 3.2, the first column introduces the system name followed by the term used to refer to the system throughout the thesis highlighted in italics in Table 3.2. In addition to their names, we detail their domains, the architecture adopted in their development, and a brief description of their main purpose. In Table 3.3, we detail quantitative data about the systems: (i) number of lines of code (*KLOC*); (ii) number of classes (*Cl*); (iii) number of class dependencies (*Cl-Dep*); (iv) number of packages (*Pkg*); (v) number of packages dependencies (*Pkg-Dep*); (vi) number of manually recovered conceptual rules (*C-Rules*); and (v) number of manually recovered architecture modules (*Modules*). Moreover, by analyzing the architecture of subject systems, we observed significant differences in architectural aspects due to their different purposes.

To have representative systems in our study, we selected systems that are from different domains, sizes and architecture organizations. On the one hand, EC, Metrics and OLIS have traditional architectures, with elements concentrated in business modules. On the other hand, ArchStudio, AspectJ and RecSys have more complex architectures, which integrate many architecture styles.

In order to obtain the conceptual architecture of our subject systems, we obtained information with architects and developers of each system when possible, used available documentation, and followed existing guidelines (GARCIA; IVKOVIC; MEDVIDOVIC, 2013). ArchStudio had its architecture recovered in a prior study (GARCIA; IVKOVIC; MEDVIDOVIC, 2013), while AspectJ had available architecture documentation which was refined based on the manual investigation of specific modules. Further information regarding the subject systems and their architectures can be seen in Appendix A.

Table 3.2: Characteristics of our Subject Systems.

Name	**Short Name**	**Domain**	**Architecture**	**Description**
ArchStudio 4	*ArchStudio*	Software Development	Heterogeneous	Open-source software to support the development of system architectures.
AspectJ Compiler and Weaver	*AspectJ*	Software Development	Heterogeneous	Eclipse platform based tool for aspect-oriented software development.
Expert Committee	*EC*	Conference Management	Layered	Web conference management system.
Eclipse Metrics Plugin Continued	*Metrics*	Software Development	Extended MVC	Eclipse plugin for calculation of project metrics.
OnLine Intelligent Services	*OLIS*	Personal Assistance	Layered	Web application to provide personal assistant services.
Recommender System	*RecSys*	Recommender Systems	Heterogeneous	Desktop-based decision making system.

Table 3.3: Subject Systems and their Characteristics.

System	**KLOC**	**Cl**	**Cl-Dep**	**Pkg**	**Pkg-Dep**	**C-Rules**	**Modules**
ArchStudio	236.9	2308	15894	319	2580	53	57
AspectJ	217.9	1667	11581	215	1779	31	15
EC	11.7	195	1095	52	309	19	6
Metrics	15.6	150	654	28	156	8	4
OLIS	11.4	211	798	45	221	13	5
RecSys	22.8	404	1625	92	930	19	10

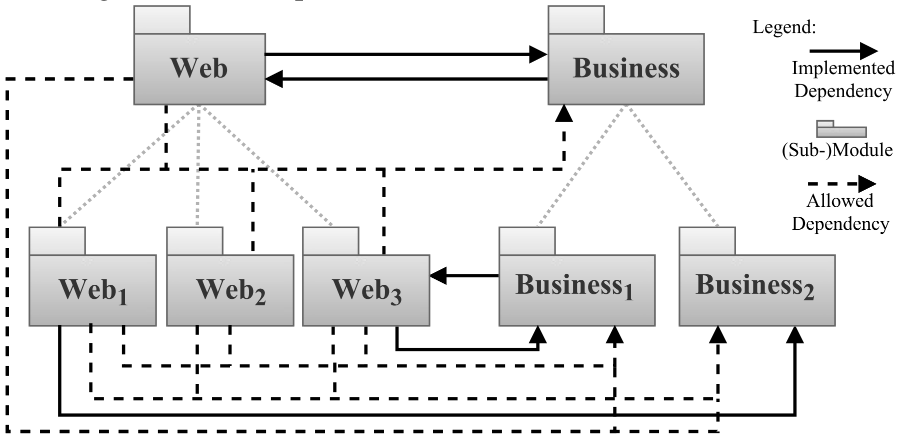
3.2 Results and Analysis

We next present and discuss our study results, answering each of our research questions.

3.2.1 Gap between conceptual architecture rules and implemented module dependencies

As discussed, previous work (BRUNET et al., 2012; LUTELLIER et al., 2015) has shown that there is a large gap between conceptual architecture rules and implemented module dependencies. Differently from previous analyses, we investigate this gap from two perspectives. First, similarly to previous work, we observe how implemented dependencies match architecture rules. Considering the semantics of architecture rules, we evaluate which implemented dependencies are allowed. Second, we make this analysis in the opposite direction. We evaluate which allowed dependencies are implemented. We refer to these two evaluations as architecture and rule conformance, respectively, corresponding to our M1 and M2 metrics. To illustrate, assume that a system, shown in Figure 3.1, has a rule $web \rightarrow business$. This rule leads to 12 possible allowed dependencies. Further assume that implemented dependencies are the arrows in bold. As a result, the *architecture conformance* is 60% (3 of 5 implemented dependencies are in conformance with the rule) and 2 (40%) are architectural violations. The *rule conformance* is 25% (3 of 12 allowed dependencies are implemented).

Figure 3.1: Example of Architecture and Rule Conformance.



Results obtained for each subject system are shown in Table 3.4. We show the number of allowed module dependencies according to the conceptual architecture rules

Table 3.4: Architecture Conformance (M1) and Rule Conformance (M2) of each Subject System.

System	**Dependencies**		**Conformance**	
	Allowed	**Implemented**	**Architecture**	**Rule**
ArchStudio	16764	1178	26.1%	1.8%
AspectJ	1398	683	28.7%	14.0%
EC	756	135	94.1%	16.8%
Metrics	63	45	55.6%	39.7%
OLIS	701	86	93.0%	11.4%
RecSys	923	375	36.0%	14.6%
AVG			55.6%	16.4%
SD			31.2%	12.8%

and those implemented. The metrics derived from them are shown in the last two columns. We also show in the last two rows the average (AVG) and standard deviation (SD) of these two metrics.

The results associated with architecture conformance corroborates with existing results, that is, architecture conformance is low. The number of architectural violations ranges from 44.4%–73.9% of the implemented dependencies for the majority of the systems. Exceptions are EC and OLIS, in which developers apparently followed architecture rules while implementing the system, having almost 95% of its implemented dependencies in accordance with rules.

Regarding rule conformance, no system achieved high results, being the highest 39.7% (Metrics). The low results of this metric show that there is a significant number (in the worst case, 98.2%) of dependencies that are allowed by rules but do not actually occur in the code. This result indicates that architecture rules are possibly too permissive. Analyzing our subject systems, we observed that the architecture modules referred in rules are often coarse-grained and have a high number of sub-modules organized in a module hierarchy. Therefore, the architecture corresponds to an abstract model of the system, without specifying fine-grained architecture decisions. As a consequence, a significant portion of the system is developed without any governance, which can lead to an emergent organization that can include violations to initially planned rules. Although an abstract model of the system is important to understand the system as a whole, this scenario gives evidence of the need for finer-grained architecture rules.

The only system that has higher rule conformance is Metrics, which is our smallest system. This level of conformance is explained by the fact that it has a low number of modules (in comparison with the other systems). Therefore, the granularity level of modules referred in rules is closer to that of implemented modules.

The analysis of the conformance between conceptual architecture rules and implemented module dependencies revealed a large gap between them, with divergences ranging from 5.9% to 73.9%. Consequently, the number of architectural violations is high. Moreover, we observed that the amount of allowed dependencies that never occur in the code is even higher, ranging from 60.3% to 98.2%. This indicates that rules should be finer-grained and more restrictive than they are.

3.2.2 Implemented module dependencies categorization

Our previous analysis allowed us to observe that there are many divergences between conceptual architecture rules and implemented module dependencies. We now further investigate them to understand the nature of these divergences, in order to answer RQ2.

The analysis of this section takes into consideration the previously introduced definition of architecture rules that we use in this chapter. Rules are specified in terms of modules, such as a rule $A \rightarrow B$, meaning that any class of A (including those in A's sub-modules) can depend on any class of B (including those in B's sub-modules). Moreover, within a module, dependencies are allowed. Based on this definition, we identified *four* categories of dependencies, with respect to their relationship with conceptual architecture rules.

Conceptual dependencies are those that *exactly* match conceptual rules of a subject system. For instance, dependencies from classes of the `preference.explanation` package to the `preference.reasoner` package, which correspond to the $Explanation \rightarrow Reasoner$ rule.

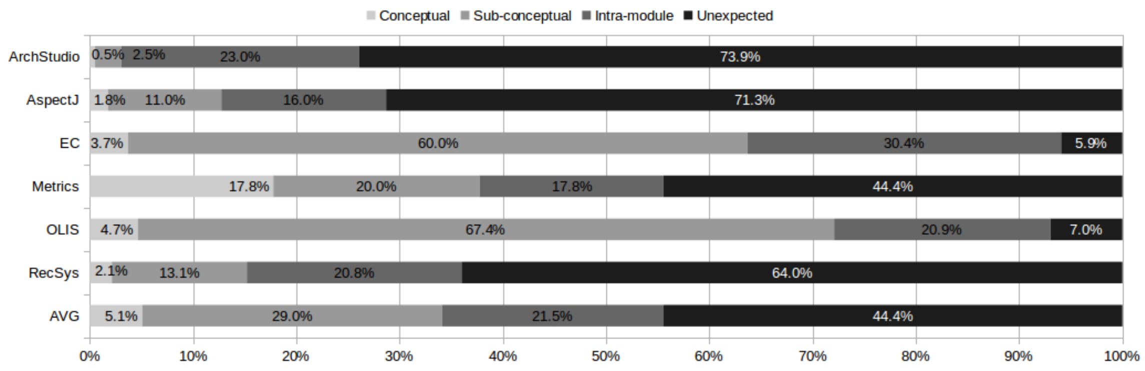
Sub-conceptual dependencies are those that are in accordance with conceptual rules but do not exactly match them. For example, the dependency $preference.reasoner.labreuche \rightarrow preference.explanation.domain$ does not exactly match the $Reasoner \rightarrow Explanation$ rule. However, both source and target packages correspond to sub-modules of modules referred in the rule, no finer-grained rule exists. Therefore, this dependency is in accordance with the rule, but occurs at a finer-grained granularity.

Intra-module dependencies are those that occur between sub-modules within the same architecture module. Architecture rules refer to modules at some level of granularity, and dependencies within modules at a finer-grained granularity level are considered allowed, although not explicitly stated. We take the *View* module of RecSys as an example. No conceptual rule specifies allowed dependencies within this module, but there is the dependency $ucpb.view.action \rightarrow ucpb.view.state$, which is an intra-module dependency.

Unexpected dependencies are all those remaining. Consequently, if a module dependency cannot be classified in one of the categories above, it is considered unexpected, because it neither corresponds to a conceptual rule nor is derivable from them. By analyzing dependencies of this type, we observed that they can be: (i) *violations* of architecture rules; or (ii) dependencies that should be documented as conceptual rules (i.e. *undocumented or unknown architecture rules*). Considering the same subject system used to exemplify the rule categories above, the dependency $database \rightarrow domain.mobile$ is classified as unexpected.

Based on the categories identified in our systems, we investigate the frequency of each category in our subject systems. In Figure 3.2, we show the results of this investigation indicating the percentage of dependencies that are present in each identified category.

Figure 3.2: Distribution of Implemented Module Dependencies by Category for each Subject System.



As expected, the percentage of conceptual dependencies is low across all subject systems, varying from 0.5% (ArchStudio) to 17.8% (Metrics). This low percentage is expected because rules typically correspond to generalizations of sub-conceptual and intra-module dependencies, which occur among sub-modules of modules referred in rules. Some of our subject systems have no classes in the modules referred in rules and, consequently, it is impossible to have dependencies directly among them. However, small systems, such as Metrics, do not have a deep module hierarchy causing the percentage of conceptual dependencies to be higher.

Dependencies that are in accordance with architecture rules thus mostly occur as sub-conceptual and intra-module dependencies. The majority of dependencies of the systems that are in conformance with their architectures, namely EC and OLIS, are sub-conceptual, while in the remaining cases they range from 2.5% to 20.0%. The amount of intra-module dependencies is mostly consistent across the different systems, ranging from 16.0% to 30.4%.

Although these dependencies are in accordance with architecture rules, the cases in which they are representative in subject systems reveal two potential problems. First, the high amount of sub-conceptual rules may indicate that architecture rules are at a too coarse-grained level of granularity. For example, EC implements a *Facade* pattern to the *Presentation* module use the *Business* module. Although the architecture rule informed by developers is $Presentation \rightarrow Business$, dependencies (should) occur to the *business.facade* module. Therefore, the rule allows dependencies that should not be present in the code. This issue was discussed in the previous section, which showed that the number of dependencies allowed by rules is much higher than what is actually implemented. Second, the modest high amount of intra-module dependencies also indicates the possible need for refinement of architecture rules. All the dependencies that occur within modules specified in rules are allowed. However, the number of dependencies is high, and this means that there are many dependencies that occur without any governance, as mentioned earlier, which can lead to a disorganized system evolution.

Not surprisingly, the number of unexpected dependencies in most of the systems is high, representing up to 73.9% of the dependencies. This is expected because we identified in the previous section that many of the systems are not in conformance with their architectures. Only two systems have a low number of unexpected dependencies, EC and OLIS, because they are mostly in accordance with their architecture, as discussed. This high number is justified not only by architectural violations but also by architecture rules that, based on code inspection, we believe are undocumented. This is the case of AspectJ and RecSys. Moreover, we observed that dependencies to utility modules were often ignored when specifying architecture rules, leading to unexpected rules.

Four categories of implemented module dependencies were identified, namely conceptual, sub-conceptual, intra-module, and unexpected. The high number of sub-conceptual (AVG = 29.0%) and intra-module (AVG = 21.5%) dependencies indicates that there is need for finer-grained architecture rules. These are in conformance with the architecture but not explicitly documented.

3.2.3 Automated distinction of implemented module dependencies

Our previous results suggest that the granularity used in architecture rules may be inadequate. In some occasions, as exemplified, a better granularity might correspond to a sub-conceptual or intra-module rule. In order to investigate if it is possible to provide some automation in the identification of an adequate granularity level for rules, we evaluate the frequency of dependencies between classes of modules. The selection of an adequate granularity level is done using the following reasoning. Consider a module A that has two sub-modules A_1 and A_2. If many classes of A_1 depend on classes of a module B but this is not the case of classes in A_2, we assume that the best granularity for a rule would be $A_1 \rightarrow B$. However, if classes of A_2 also depend on classes of B, the best granularity would be $A \rightarrow B$.

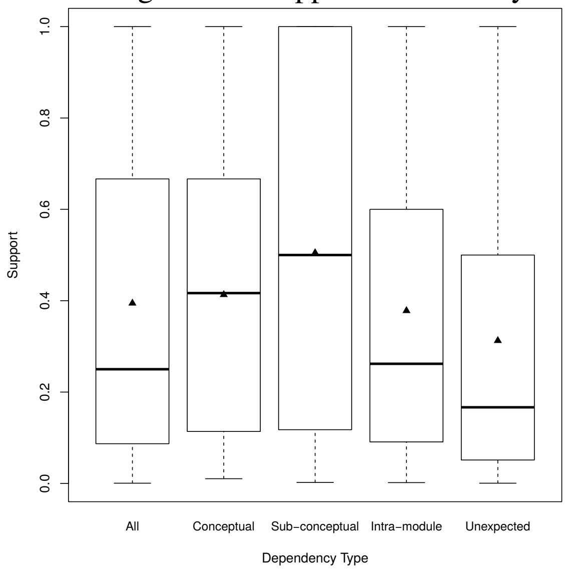
To make this evaluation, we use the support metric, as mentioned in our study procedure. As explained, we calculate the support not only of individual modules, but also modules combined in a super-module. We analyze obtained support values and also verify whether dependencies of different categories are distinguishable by their support metric, that is, there are statistically significant differences among support values. If this is the case, this metric is a potential candidate to be used as input to a method that automatically recommends the granularity level to be used in architecture rules.

We report the values obtained for the support metric considering each dependency category and subject system in Table 3.5, where AVG stands for the average, SD stands for the standard deviation, Med stands for the median, and Min and Max stand for the minimum and maximum values, respectively. These data are also shown in Figure 3.3, which presents a box plot of the values obtained for each subject system by dependency category.

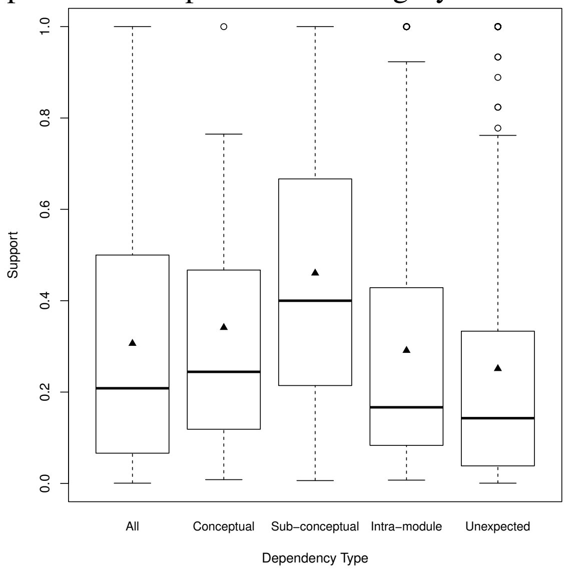
Table 3.5: Support Metric by Dependency Category for each Subject System.

	Dependency Category	**AVG** (%)	**SD** (%)	**Med** (%)	**Min** (%)	**Max** (%)
ArchStudio	Conceptual	41.3	30.1	41.7	1.0	100.0
	Sub-conceptual	50.5	38.1	50.0	0.2	100.0
	Intra-module	37.8	33.4	26.2	0.2	100.0
	Unexpected	31.3	33.6	16.7	0.1	100.0
	All	39.5	35.1	25.0	0.1	100.0
AspectJ	Conceptual	34.1	27.9	24.4	0.8	100.0
	Sub-conceptual	46.0	32.1	40.0	0.6	100.0
	Intra-module	29.1	28.0	16.7	0.7	100.0
	Unexpected	25.1	27.2	14.3	0.1	100.0
	All	30.6	29.7	20.8	0.1	100.0
EC	Conceptual	55.3	33.3	46.9	11.1	100.0
	Sub-conceptual	60.2	31.7	50.0	1.9	100.0
	Intra-module	58.3	32.0	50.0	7.7	100.0
	Unexpected	35.6	29.7	50.0	2.0	100.0
	All	55.6	32.6	50.0	1.9	100.0
Metrics	Conceptual	53.3	29.1	50.0	16.7	100.0
	Sub-conceptual	39.6	29.9	22.2	13.3	100.0
	Intra-module	46.6	37.5	25.4	14.3	100.0
	Unexpected	23.2	24.3	14.3	1.0	88.9
	All	30.5	28.4	18.3	1.0	100.0
OLIS	Conceptual	46.6	34.9	37.5	4.4	100.0
	Sub-conceptual	56.9	35.9	62.5	1.5	100.0
	Intra-module	61.0	43.9	100.0	4.0	100.0
	Unexpected	9.1	8.7	6.3	2.5	42.9
	All	52.2	38.0	50.0	1.5	100.0
RecSys	Conceptual	48.1	32.9	30.4	5.6	100.0
	Sub-conceptual	58.4	34.9	50.0	2.6	100.0
	Intra-module	35.1	28.0	25.0	1.1	100.0
	Unexpected	27.1	25.4	20.0	0.7	100.0
	All	33.3	29.7	25.0	0.7	100.0

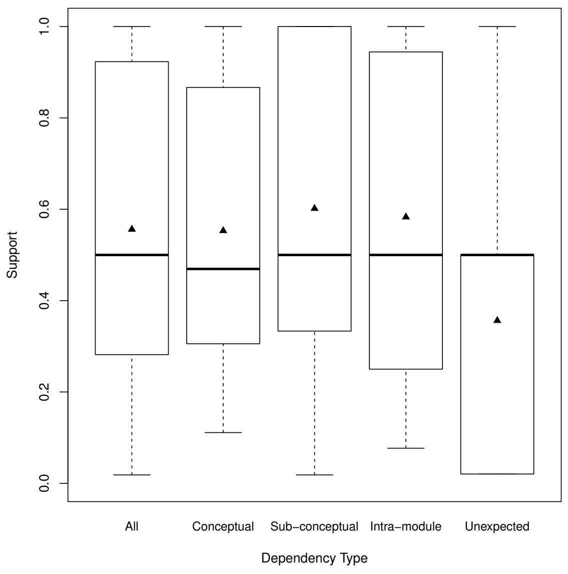
Figure 3.3: Support Variation by Implemented Dependencies Category.



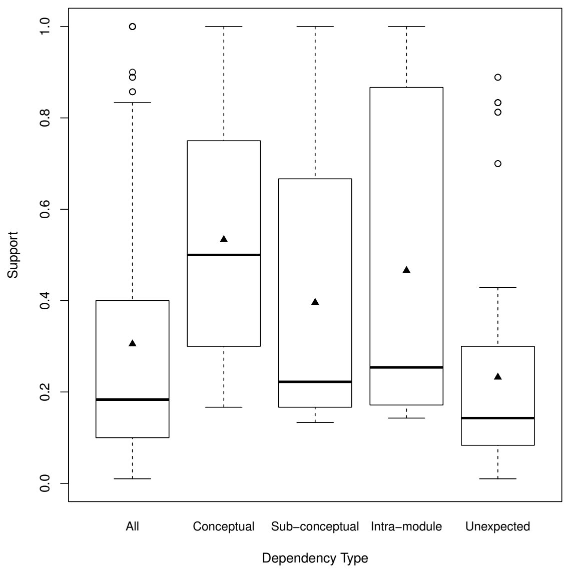
(a) ArchStudio



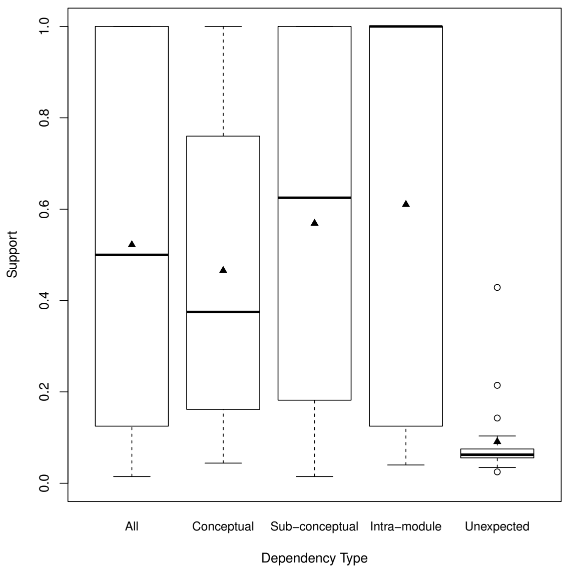
(b) AspectJ



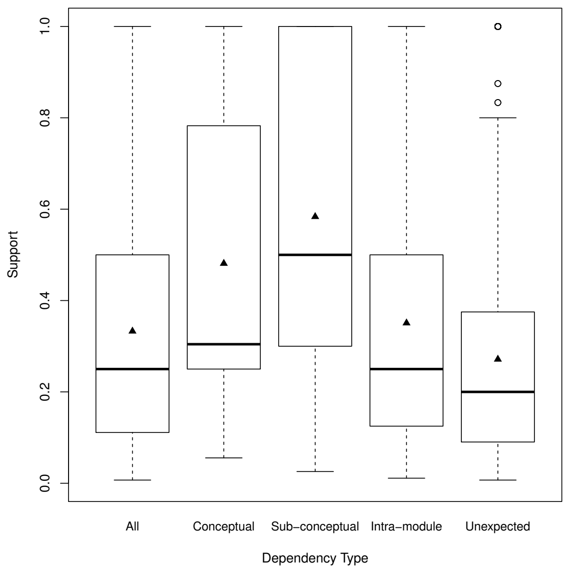
(c) AspectJ



(d) Metrics



(e) OLIS



(f) AspectJ

Our first observation is that both average and median of the support of sub-conceptual dependencies are higher than conceptual dependencies for all systems but Metrics. When we analyze dependencies throughout the module hierarchy, all analyzed levels are sub-conceptual except the conceptual level. Consequently, many of the sub-conceptual levels can possibly have low support because they might not correspond to a pattern in the implementation, that is, when many classes from a module depend on another. Despite this, which justifies a lower support for some of the sub-conceptual dependencies (leading to higher variance), conceptual dependencies still have generally lower support. This is thus a supporting argument to sustain our claim that architecture rules should be finer-grained because it indicates that only a subset of sub-modules of a module referred in a conceptual rule consistently depends on the same module. Therefore, rules would better reflect what should be in the implementation if they were expressed in terms of these sub-modules. The Metrics system does not have this behavior because it has a flat module hierarchy with few modules and has dependencies at the conceptual level, as shown in Figure 3.2.

With respect to intra-module dependencies, we observed that all systems are different from each other. There are three different cases. First, within a module, there are many dependencies among sub-modules, all of them with not so high support (lower than 50%). Therefore, the lack of rules governing dependencies among sub-modules might be the adequate choice. Second, there are specific cases where a sub-module strongly depends on another (support higher than 50%). In this case, rules indicating allowed dependencies within these modules would have helped to limit dependencies to these specific cases, preventing violations. Third, there are situations similar to the second case, but dependencies can be generalized to a coarse-grained rule. For example, $A \rightarrow B$, $A_1 \rightarrow B$, and $A_2 \rightarrow B$ have all high support; therefore, a single rule $A \rightarrow B$ captures these dependencies.

Unexpected dependencies generally have low support, being most of them violations according to our analysis. This can be clearly seen in OLIS, which is the system with the smallest gap between conceptual rules and implemented dependencies. However, there are few exceptions. For example, EC has unexpected dependencies with 100% of support, which are a consequence of two possible high-level undocumented rules. After consulting developers, no consensus was reached regarding this because although the dependency expressed in these two rules is a violation (as no rule allows it), EC developers did not consider a violation if only *getters* are invoked (no business operation is executed).

In order to verify whether the support metric significantly varies across the different dependency categories, we performed a Kruskal-Wallis test for each system (a Shapiro-Wilk test revealed that the distribution of the support values is not normal for all systems, at a significance level of $\alpha = 0.05$). As result, the Kruskal-Wallis tests indicated that all support values are significantly different across the different dependency categories ($p < 0.05$), for all systems. We then conducted a Nemenyi post-hoc test, which are presented in Table 3.6, to identify significant differences. Unexpected dependencies have a support significantly different in most of the cases, indicating that the support metric can be used to help identify architectural violations or undocumented rules, because low support likely corresponds to unexpected dependencies.

Table 3.6: Nemenyi Post-hoc Test by Pairs of Dependency Category for each Subject System.

	Dependency Category	**Sub-conceptual**	**Intra-module**	**Unexpected**
ArchStudio	Conceptual	0.72	0.91	0.18
	Sub-conceptual	-	*<0.01*	*<0.01*
	Intra-module	-	-	*<0.01*
AspectJ	Conceptual	0.48	0.85	0.31
	Sub-conceptual	-	*<0.01*	*<0.01*
	Intra-module	-	-	*<0.01*
EC	Conceptual	0.94	0.99	0.19
	Sub-conceptual	-	0.98	*<0.01*
	Intra-module	-	-	*<0.01*
Metrics	Conceptual	0.74	0.92	*<0.01*
	Sub-conceptual	-	99.0	*<0.01*
	Intra-module	-	-	0.08
OLIS	Conceptual	0.71	0.61	*<0.01*
	Sub-conceptual	-	96.0	*<0.01*
	Intra-module	-	-	*<0.01*
RecSys	Conceptual	0.88	0.25	*<0.01*
	Sub-conceptual	-	*<0.01*	*<0.01*
	Intra-module	-	-	*<0.01*

The support of conceptual dependencies is generally lower than the support of sub-conceptual dependencies (except for systems with a flat and small module hierarchy), confirming that architecture rules should be finer-grained. The support of unexpected dependencies is significantly lower than the support of other dependency categories, indicating that this metric can be used to identify groups of dependencies that correspond to architectural violations or undocumented rules.

3.3 Threats to Validity

We identified a set of threats to the validity of our study. We next state them and describe how they were mitigated. An internal threat is associated with our manual architecture recovery process performed with our subject systems, because the same system can be viewed by different architects or developers in different ways, leading to alternative architectures and architecture rules. The two largest systems had their documented architecture available and we used them (GARCIA et al., 2013). To increase the reliability of the recovered architectures of the other systems, we contacted key architects and developers directly involved in their development. Moreover, we validated the correctness of the recovered conceptual architecture following guidelines presented in a study done by Garcia et al. (GARCIA et al., 2013), regarding the provision of ground-truth software architectures.

An external threat that may compromise the generalization of our results is the number of systems investigated. Systems that share many commonalities, e.g. domain, and architecture style, are likely to produce similar results, limited to systems with their characteristics. To mitigate this problem, we selected systems with different characteristics. Our subject systems are from different domains, have different sizes and adopt different architecture styles, as discussed in Section 3.1.3. Our findings provide evidence of system diversity, and results differ across the systems, which is important due to the exploratory nature of our study. We emphasize that our goal is to observe key reasons that explain the divergence between conceptual software architectures and implemented dependencies rather than to test a hypothesis.

3.4 Final Remarks

In this chapter, we presented an exploratory study that assesses and investigates the gap between conceptual architecture rules and dependencies among modules implemented in the source code. Six subject systems had their conceptual architecture recovered and compared with implemented module dependencies. As expected, many of the systems have many divergences between rules and dependencies. While investigating these divergences, we identified that not only there are dependencies that do not comply with rules (thus typically being considered violations) but also that many allowed dependencies do not occur in the code. This suggests that many rules are more general than they should be, and could have been specified in a finer-grained way.

Based on the analysis of these four categories, we identified particularities to characterize them. The key characteristic that was analyzed was the support metric, which is associated with the dependency frequency. As result of this analysis, key findings are: (i) support of conceptual dependencies varies according to system organization, and is also influenced by the support of sub-conceptual and intra-module dependencies; (ii) sub-conceptual and intra-module dependencies are two types of dependencies typically not investigated, which can provide information about the quality of the system architecture, giving evidence with respect to coupling and cohesion; and (iii) unexpected dependencies, which can be identified by the support metric, are useful to identify unplanned rules that should be documented as conceptual or violations. In the next chapter, we present our proposed method to recover architecture rules, which was developed based on the findings of this analysis.

4 THE WGB METHOD

In this chapter, we present our WGB method. The WGB method chooses *a set of architecture rules to represent an implemented software architecture taking as input a given software module organization (e.g., package structure) and dependencies among module elements* (e.g., classes). These recovered rules are a coarse-grained representation of the implemented architecture that are an architectural view of the system. Our method is composed of three sequential steps: (i) calculation of a metric that captures the dependency strength between every two modules, considering dependencies between elements (Section 4.1); (ii) pairwise clusterization of dependencies based on this metric, considering neighbor module levels (Section 4.2); and (iii) selection of the set of rules that maximizes the dependency strength without redundancy, which we assume as the correct rule granularity to represent element dependencies (Section 4.3). In Section 4.4, we discuss scenarios in which our method can be used.

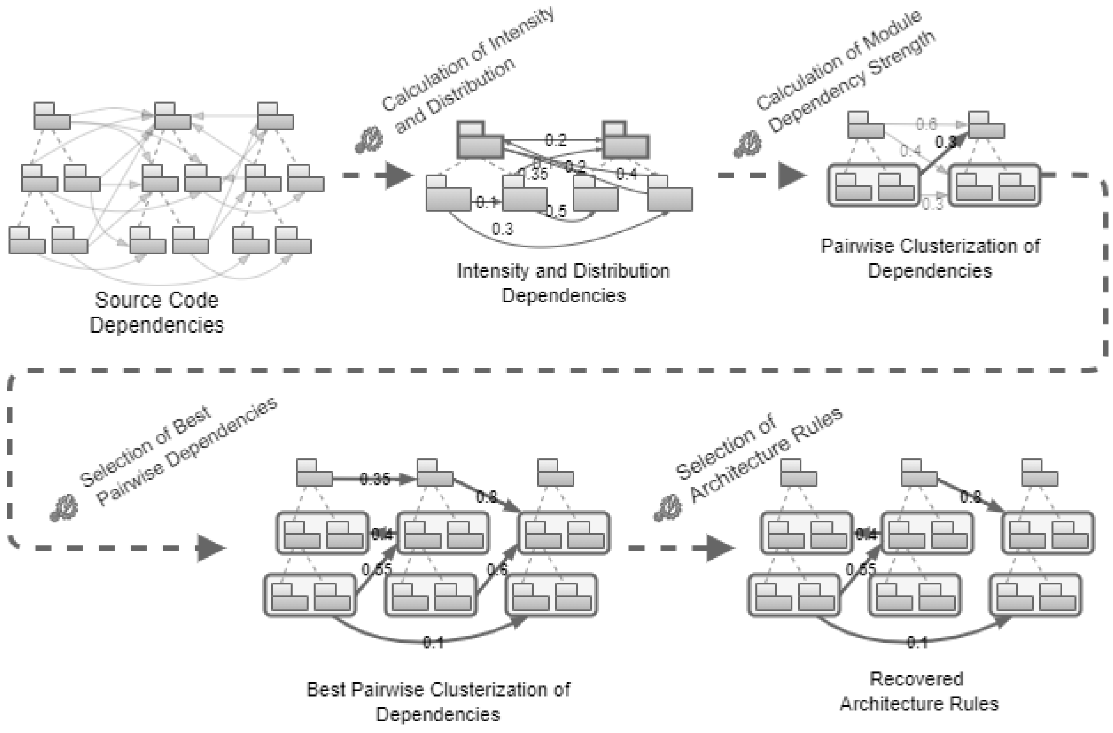
These steps and associated inputs and outputs are illustrated in Figure 4.1. Based on source code dependencies, we extract metrics (intensity and distribution), which are combined in a single metric, the Module Dependency Strength (MDS). The calculated MDS is then used to prune less representative dependencies. Finally, from candidate rules, we select a set of rules with no redundant information and has the highest MDS.

4.1 Module Dependency Strength

As discussed, dependencies from elements of a given module to elements of another can be used to evaluate the dependency between both these modules, and their parent modules. Such dependencies can be used to calculate a metric that indicates the *dependency strength* between modules in different levels of the module organization structure. This metric captures how strong the dependency between two modules is, being composed of two key components, namely dependency *intensity* and dependency *distribution*.

The reasoning of this metric is twofold. First, the *higher* the *percentage* of elements of a module X that depend on elements of a module Y, as well as the percentage of such Y's elements; the *higher* the dependency *intensity*. Second, the dependency from X to Y when X's elements depend on Y's elements is *stronger* than when the elements of X's *siblings* also *depend* on Y's elements, or X's elements also depend on the elements of Y's siblings. The opposite must occur when evaluating the dependency between

Figure 4.1: Overview of the WGB Method.

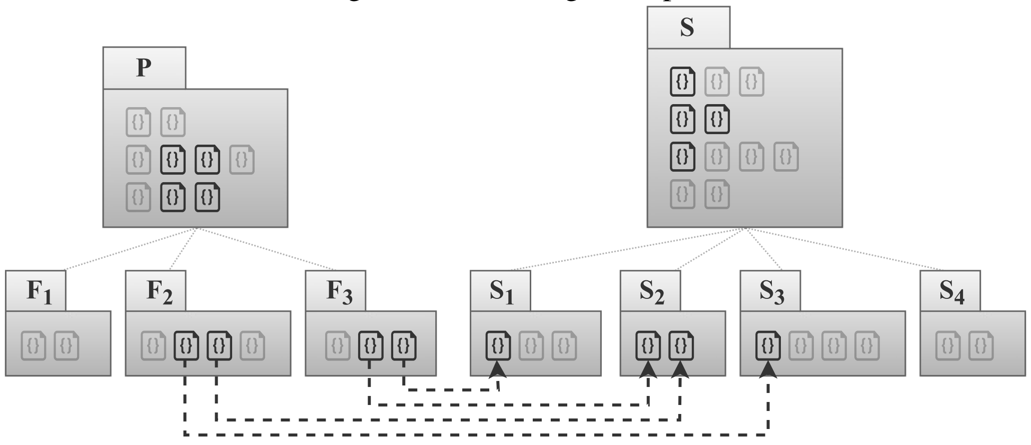

Source Code Dependencies
Calculation of Intensity and Distribution
Intensity and Distribution Dependencies
Calculation of Module Dependency Strength
Pairwise Clusterization of Dependencies
Selection of Best Pairwise Dependencies
Best Pairwise Clusterization of Dependencies
Selection of Architecture Rules
Recovered Architecture Rules

X's and Y's parents. This is captured by the dependency *distribution*. We next detail how this metric—named *module dependency strength*, or $MDS(X, Y)$, where X and Y are modules—is calculated for parent and child modules using a bottom-up approach. It mainly differs from module coupling (ALLEN; KHOSHGOFTAAR; CHEN, 2001; OFFUTT; HARROLD; KOLTE, 1993) by having low values when the dependency between a pair of modules that have high coupling are due to parent modules.

Before detailing our metric, we introduce the adopted notation where:

- $X, Y, ...$ are modules in a module set M;
- X_P is used to denote a module being analyzed as a parent module;
- X_C is used to denote a module being analyzed as a child module;
- X_{C_i} is the i^{th} child of X_P;
- $x, y, ...$ are element of the element set E;
- $x, y, ...$ are elements of the modules $X, Y, ...$, respectively;
- The parent function $p : M \to M$ gives the parent module of a given module; and
- The children function $c : M \to \wp(M)$ gives its children.

Figure 4.2: Running Example.



Given that we assume a hierarchy in the form of a tree, a module has only one parent and can have many children. Moreover, the element function $el : M \rightarrow E$, described in Equation 4.1, gives all elements in a module hierarchy.

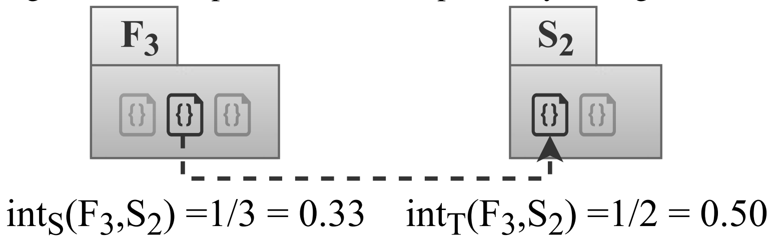
$$el(X) = \begin{cases} \{x | x \in X\}, & \text{if } c(X) = \emptyset \\ \bigcup_{X_{C_i} \in c(X)} el(X_{C_i}), & \text{otherwise} \end{cases} \tag{4.1}$$

For now, assume that parent modules do not have elements within it, that is, $el(X_P) = \emptyset$. Later, we explain how this is dealt with. Finally, $x \rightarrow y$ indicates that an element x depends on an element y, and $X \rightarrow Y$ indicates a module dependency, i.e. there is a $x \in el(X)$ and $y \in Y$, such that $x \rightarrow y$.

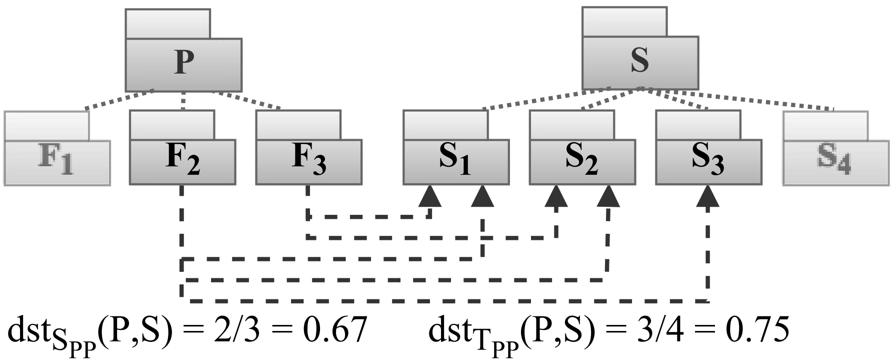
Using this notation, we also introduce an example that is used throughout this section to illustrate our approach. Consider a system that has two main modules, Presentation, or P for short, and Service, or S for short. P implements three features, each implemented by elements within their respective modules, F_1,F_2, and F_3. S provides different services, implemented in four different modules, namely S_1, S_2, S_3, and S_4. This structure of modules with their elements is presented in Figure 4.2. The dependencies between elements of child modules are depicted with arrows. The elements of parent modules, P and S, are the union of children's elements—we highlight in gray those elements in a parent module that have a dependency (or is a dependee).

We first describe how the dependency intensity is calculated. The intensity $int_S(X, Y)$ of the dependency of a dependent module X with respect to a dependee module Y is the percentage of X's elements that depend on Y's elements, as shown in Equation 4.2. This is illustrated in Figure 4.3a, which shows that one of F_3's elements depend on one of S_2's elements. Therefore, $int_S(F_3, S_2) = 0.33$ in this example. Similarly, the inten-

Figure 4.3: Components of the Dependency Strength Metric.



(a) Intensity



(b) Distribution

sity $int_T(X, Y)$ of the dependency of a dependee module Y with respect to a dependent module X is the percentage of Y's elements on which X's elements depend, as shown in Equation 4.3. According to the scenario shown in Figure 4.3a, $int_T(F_3, S_2) = 0.5$, because there are one S_2's element (out of two) on which F_3's element depend. The subscripts S and T are used in the dependency intensity to indicate whether the source or target modules, respectively, are being considered.

$$int_S(X, Y) = \frac{|\{x | x \in el(X) \wedge y \in el(Y) \wedge x \to y\}|}{|el(X)|} \tag{4.2}$$

$$int_T(X, Y) = \frac{|\{y | y \in el(Y) \wedge x \in el(X) \wedge x \to y\}|}{|el(Y)|} \tag{4.3}$$

The dependency distribution, in turn, does not take into account a single module, but a set of modules. It evaluates how much the dependency of a module on another is spread among its children. Therefore, the base of this metric component is evaluated considering one of the modules as a parent. The distribution $dst_{S_P}(X_P, Y)$ of a source parent module X_P is the percentage of its children that depend on the Y module, while the dependency $dst_{T_P}(X, Y_P)$ of a target parent module Y_P is the percentage of its children on which X depend, as shown in Equations 4.4 and 4.5, respectively. Figure 4.3b illustrates this metric component.

$$dst_{S_P}(X_P, Y) = \frac{|\{X_C | X_C \in c(X_P) \wedge X_C \rightarrow Y\}|}{|c(X_P)|} \tag{4.4}$$

$$dst_{T_P}(X, Y_P) = \frac{|\{Y_C | Y_C \in c(Y_P) \wedge X \rightarrow Y_C\}|}{|c(Y_P)|} \tag{4.5}$$

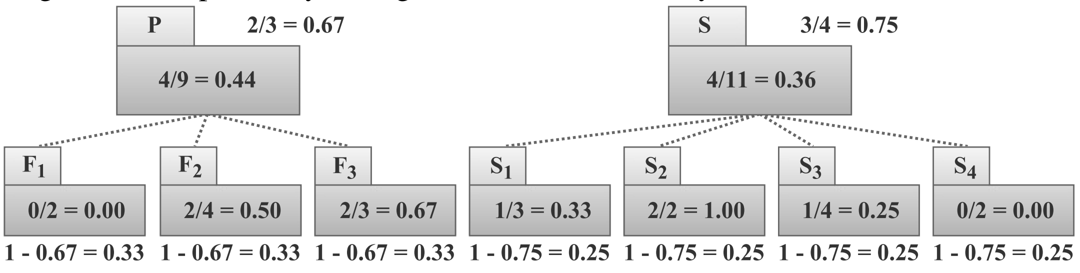
To take distribution into account, we must know whether the dependency strength is being calculated for a parent or a child module. The higher the number of children of a parent module that depend on another, the higher the dependency strength, from a parent perspective; while the lower the number of siblings of a child module that depend on another, the higher the dependency strength, from a child perspective. Therefore, the distribution of a module being analyzed as a parent module corresponds to the equations above, while the distribution of a module being analyzed as a child module is the inverse. Consequently, the dependency distribution is given by the equations below, in which a level function $l : M \rightarrow \{P, C\}$ is used to tell whether a module $X \in M$ is being analyzed as a parent (P) or a child (C) module.

$$dst_S(X, Y) = \begin{cases} dst_{S_P}(X, Y), & \text{if } l(X) = P \\ 1 - dst_{S_P}(p(X), Y), & \text{if } l(X) = C \end{cases} \tag{4.6}$$

$$dst_T(X, Y) = \begin{cases} dst_{T_P}(X, Y), & \text{if } l(Y) = P \\ 1 - dst_{T_P}(X, p(Y)), & \text{if } l(Y) = C \end{cases} \tag{4.7}$$

After introducing the intensity and distribution components, we can proceed to our MDS metric. Approaches that mine dependencies to derive architecture rules only consider the dependency intensity between a source (sub-)module to another. This only serves as an indication that there must be a rule to represent the coupling between a source module to a target module. However, this information is not enough to decide whether (1) this coupling occurs solely from a particular sub-module to another, making a rule that refers to this sub-module more adequate, or (2) from many sibling modules to another, making a rule that refers to the parent module more adequate. Therefore, it is crucial to use the dependency distribution to indicate if the dependencies are localized

Figure 4.4: Dependency Strength Metric with Intensity and Distribution Calculation



in a particular (sub-)module (leading to a high intensity associated with this module) or are also present in sibling modules, suggesting that a coarse-grained rule would be more adequate to represent the dependencies in an architecture model.

The dependency strength between two modules thus takes into account how frequent the dependency occurs from both the source and target module perspectives (intensity), adjusted by how this dependency is distributed. As a consequence, our metric is *context-sensitive*, in the sense that it takes into account the dependency that occurs between surrounding modules. The distribution is used as a weight of the intensity, as detailed in Equation 4.8. Given that the dependency strength must be on a similar scale so that we can compare its value calculated for different pairs of modules, weights are normalized so that the dependency strength $MDS(X,Y) \in [0,1]$.

$$MDS(X,Y) = \widehat{dst_S(X,Y)} \times int_S(X,Y) + \widehat{dst_T(X,Y)} \times int_T(X,Y) \tag{4.8}$$

where $\widehat{dst_S(X,Y)}$ and $\widehat{dst_T(X,Y)}$ are normalized $dst_S(X,Y)$ and $dst_T(X,Y)$, respectively, such that $0 \leq \widehat{dst_S(X,Y)}, \widehat{dst_T(X,Y)} \leq 1$, and $\widehat{dst_S(X,Y)} + \widehat{dst_T(X,Y)} = 1$.

In order to illustrate the module dependency strength metric, we use the example scenario presented in Figure 4.2. Based on the equations above, intensity and distribution are calculated, and they are detailed in Figure 4.4. Values are shown in P and S considering $MDS(P,S)$, in F_i considering $MDS(F_i,S)$, and in S_i considering $MDS(P,S_i)$.

With these values, we can calculate the dependency strength between parent modules, child modules, parent-child modules and child-parent modules. For example:

- $MDS(P,S) = 0.47 \times 0.44 + 0.53 \times 0.36 = 0.21 + 0.19 = 0.40$, where 0.47 and 0.53 are the normalized distribution values 0.67 and 0.75, respectively; and

- $MDS(F_2,S) = 0.40 \times 0.67 + 0.60 \times 0.18 = 0.27 + 0.11 = 0.38$, where 0.40 and 0.60 are the normalized distribution values 0.33 and 0.50, respectively.

Table 4.1: Alternative Implemented Architecture Rules.

Parent-to-Parent	**Parent-to-Child**	**Child-to-Parent**	**Child-to-Child**
$P \rightarrow S(0.40)$	$P \rightarrow S_1(0.20)$	$F_2 \rightarrow S(0.31)$	$F_2 \rightarrow S_2(0.40)$
	$P \rightarrow S_2(0.43)$	$F_3 \rightarrow S(0.38)$	$F_2 \rightarrow S_3(0.25)$
	$P \rightarrow S_3(0.17)$		$F_3 \rightarrow S_1(0.33)$
			$F_3 \rightarrow S_2(0.43)$
AVG = 0.40	AVG = 0.27	AVG = 0.34	AVG = 0.35

Note that $MDS(P, S) > MDS(F_2, S)$ and, if the dependency distribution were not used as a weight, the values obtained for $MDS(P, S)$ and $MDS(F_2, S)$ would be 0.40 and 0.43, respectively, causing $MDS(F_2, S) > MDS(P, S)$. This means that, although the intensity absolute values from F_2 to S are higher indicating a stronger coupling between them than between P and S, there are siblings from F_2 that also depend on S, leading to a high distribution. This indicates that the overall dependency from P to S is higher and that a rule from P to S would better capture the dependencies from many of its children to S, instead of individual rules from (some of) its children to S.

4.2 Pairwise Clusterization of Dependencies

We now explain how we use the dependency strength metric to select the appropriate granularity to represent implemented architecture rules. We first make pairwise comparisons between adjacent levels in the module hierarchy in order to decide whether to cluster children dependencies in a parent module (for both source and target), and then select the best non-conflicting set of rules. We focus in this section on explaining the former. This step also helps reduce the number of considered implemented rules in the next step, which has a higher computational cost.

It is possible to represent the implemented architecture rules of the example presented in Figure 4.4 in four different ways, detailed in Table 4.1. In a real software system, we make a similar analysis, extracting all possible portions of the module organization, i.e. if there is an element of a module X_C that depends on the element of a module Y_C, we analyze two trees of two-adjacent-level modules: one with X's parent with its children and another with Y's parent with its children.

To choose between alternatives of architecture rules, we evaluate the dependency strength between every two pairs of modules using our metric. Obtained MDS values for our example, with intermediate calculated components, are shown in Table 4.2 and summarized next to the architecture rules presented in Table 4.1. When there is no dependency between two modules, the metric value is 0, and these cases are omitted in Table 4.2. Based on obtained metric values, we take the average MDS of each set of possible rules, which is shown in the last row of Table 4.1. The set with the highest average is assumed to have the most appropriate granularity level to represent existing dependencies as implemented architecture rules. In our example, the best representation is using the parent-to-parent level, that is, $P \to S$.

4.3 Selection of Architecture Rules

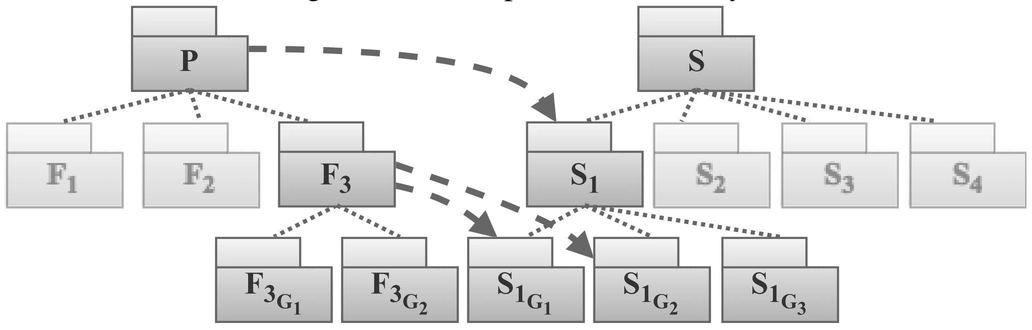
As result of the pairwise clusterization of dependencies, sets of dependencies that are candidates to represent implemented architecture rules are already discarded—three of the four options presented in Table 4.1. However, the resulting collection of sets of dependencies may still include redundant information.

Assuming that after performing the step above we have the following scenario, illustrated in Figure 4.5: (i) a parent module P depends on a child module S_1, after the analysis of P and S, and their children; and (ii) a child module F_3 depends on grandchildren modules $S_{1_{G_1}}$ and $S_{1_{G_2}}$, after the analysis of F_3 and S_1, and their children. This consists of a redundant scenario because, if the first alternative is considered an implemented architecture rule, module elements in the module hierarchy of P are allowed to depend on module elements in the module hierarchy of S_1. Because F_3 is in the hierarchy of P, and $S_{1_{G_1}}$ and $S_{1_{G_2}}$ are in the hierarchy of S_1, adding $F_3 \to S_{1_{G_1}}$ and $F_3 \to S_{1_{G_2}}$ as architecture rules would be redundant. Previous work on software rule mining results in a large number of rules, which is often large enough to become infeasible to be useful in practice, thus it is crucial to eliminate this redundant information.

Table 4.2: Values of the Module Dependency Strength (MDS) Metric.

Level	**Source**	**Target**	$int_S(X,Y)$	$int_T(X,Y)$	$dst_S(X,Y)$	$dst_T(X,Y)$	$\widehat{dst_S(X,Y)}$	$\widehat{dst_T(X,Y)}$	$MDS(X,Y)$
Parent-to-Parent	P	S	$4/9 = 0.44$	$4/11 = 0.36$	$2/3 = 0.67$	$3/4 = 0.75$	0.47	0.53	**0.40**
Parent-to-Child	P	S_1	$1/9 = 0.11$	$1/3 = 0.33$	$1/3 = 0.33$	$1/4 = 0.25$	0.57	0.43	**0.20**
	P	S_2	$2/9 = 0.22$	$2/2 = 1.00$	$2/3 = 0.67$	$1/4 = 0.25$	0.73	0.27	**0.43**
	P	S_3	$1/9 = 0.11$	$1/4 = 0.25$	$1/3 = 0.33$	$1/4 = 0.25$	0.57	0.43	**0.17**
Child-to-Parent	F_2	S	$2/4 = 0.50$	$2/11 = 0.18$	$1/3 = 0.33$	$2/4 = 0.50$	0.40	0.60	**0.31**
	F_3	S	$2/3 = 0.67$	$2/11 = 0.18$	$1/3 = 0.33$	$2/4 = 0.50$	0.40	0.60	**0.38**
Child-to-Child	F_2	S_2	$1/4 = 0.25$	$1/2 = 0.50$	$1/3 = 0.33$	$2/4 = 0.50$	0.40	0.60	**0.40**
	F_2	S_3	$1/4 = 0.25$	$1/4 = 0.25$	$2/3 = 0.67$	$2/4 = 0.50$	0.57	0.43	**0.25**
	F_3	S_1	$1/3 = 0.33$	$1/3 = 0.33$	$2/3 = 0.67$	$2/4 = 0.50$	0.57	0.43	**0.33**
	F_3	S_2	$1/3 = 0.33$	$1/2 = 0.50$	$1/3 = 0.33$	$2/4 = 0.50$	0.40	0.60	**0.43**

Figure 4.5: Example of Redundancy.



To solve this problem, we model our scenario as an optimization problem, formalized next, in which our goal is to select a dependency set that maximizes the dependency strength, subject to the elimination of dependencies that lead to redundant rules.

$$\begin{aligned} &\max \sum_{i=1}^{|D|} in_i \times \big(\omega(d_i) \times MDS(d_{i_S}, d_{i_T})\big) \\ &\quad in_x + in_y \leq 1, \forall d_x, d_y(redundant(d_x, d_y)) \end{aligned} \tag{4.9}$$

where D is the set of dependencies that are candidates to be implemented architecture rules, $d_i \in D$ having d_{i_S} and d_{i_T} as source and target, respectively, $in_i \in [0, 1]$ indicates whether the dependency d_i is selected (1) or not (0) as an architecture rule, $\omega(d_i)$ is a weight function for MDS, and $redundant(d_x, d_y)$ is true when the dependencies d_x and d_y are redundant. The weight function $\omega(d_i)$ is needed to adjust MDS according to how many dependencies a coarser-grained dependency is representing. Consider the example of Figure 4.5. All the three candidate dependencies have a value in the range $[0, 1]$. Consequently, if we simply use the sum of MDS values to be maximized, finer-grained rules would be selected, because the sum of the dependencies of many finer-grained dependencies is often higher than that of a single coarser-grained dependency. Therefore, the weight function multiplies the MDS value by the number of finer-grained dependencies it represents, i.e. those that are eliminated if this coarser-grained dependency is selected.

By solving this optimization problem, we obtain a set of values for in_i, which gives us the set of dependency representations selected as implemented architecture rules, i.e. those that have $in_i = 1$. This resulting set of rules gives the *implemented or concrete software architecture*. The modules referred in selected rules are those that should be explicitly documented, and rules indicate allowed dependencies between modules.

Finally, we detail how we deal with parent modules that contain elements. Assume that a module X_P has both elements and children X_{C_i}. While modeling the module hierarchy, we add an extra child module to X_P, which we denote as X_{C^*}, that contains X_P's elements. If, as result of our method, the dependency $X_P \rightarrow Y$ is selected as an implemented architecture rule, all elements of X_P's hierarchy, including those within X_P can depend on Y. While, if the dependency $X_{C^*} \rightarrow Y$ is selected, only the elements within X_P can depend on Y. We represent such rules as $X_P \xrightarrow{*} Y$.

4.4 Use Cases of the Method

As described in the previous sections, our method recovers a set of architecture rules based on the source code dependencies from an implemented system that has a source code with a hierarchical structure without any additional information. The main objective of our method is to recover this set of rules to be used as information to build the architecture of a system, which makes our method an architecture recovery approach if used considering only a snapshot of the source code.

Our method can also support the detection of violations based on a previously extracted set of architecture rules. By applying our method frequently, such as to every change in the source code, the developers are able to detect changes in the set architecture rules and decide if the changes should be integrated as new rules of the architecture or should be refactored to respect the previous architecture of the system. Therefore, this kind of comparison helps keep the conformance of the architecture and the source code because the developers can detect changes in the architecture and decide whether they are violations.

Our method can also supports the analysis of how the architecture of a system evolved. Our method can be applied in different snapshots of the source code to compare and understand the changes made during the evolution concerning the architecture to understand the reasoning that lead to the current architecture. Comparing different sets of architecture rules of two versions may provide evidence of the decisions made during the development that lead to a remodularization or which changes made the architecture erode or drift.

4.5 Final Remarks

In this chapter, we proposed a novel method that automatically recovers implemented architecture rules, named the Weighted-graph-based (WGB) method. Our method does not require the specification of any threshold, or system-specific customizations. Our method includes the calculation of a proposed metric, module dependency strength (MDS), between module pairs and the resolution of an optimization problem. MDS not only takes into account dependencies within a module, but also its context, i.e. its surrounding modules. Given our method, in the next chapter, we evaluate the WGB method presenting a case study, an offline study and a user study.

5 EVALUATION

Our WGB method was designed based on the analysis of different alternatives in many distinct hypothetical scenarios aiming to recover the more appropriate granularity to represent implemented architecture rules. There are fundamental differences between existing module recovery approaches and our method to recover architecture rules, which limit us to make a qualitative rather than an experimental comparison between them. In the module recovery, the approaches do not provide rules or restrictions on the dependencies between modules. They focus on the classification of source code elements into modules. Considering the conformance approaches, the main difference that limits the comparison is the output of the WGB method. Our method output is a set of architecture rules extracted from the source code that are not previously defined or classified as in the approaches discussed in Chapter 2.

To provide evidence that our method recovers an adequate set of architecture rules, we performed three evaluations: a case study; an offline study; and a user study. The case study consists of the use of the method to recover rules from an existing system and a discussion of obtained results taking into account the feedback of the system developers, which is presented in Section 5.1. Next, we present a quantitative and qualitative evaluation, in which we assess the method *efficiency* and *effectiveness* in Section 5.2. Finally, we evaluate our method based on the opinion of the developer about the usefulness and appropriateness of the rules recovered using the WGB method in Section 5.3.

5.1 Case Study

In order to demonstrate our method in action, we selected as a case study an ongoing software project that focuses on an evolving system named MDD4ABMS, which provides support to agent-based modeling and simulation. The project currently has two developers, and received contributions from other two past developers. MDD4ABMS is implemented as an eclipse plug-in that provides a domain-specific modeling language to model agent-based simulations. Moreover, it includes other features associated with model-driven development, such as code generation. This system has 41.3 KLOC, 335 classes and 40 packages. There are 1348 and 148 dependencies between classes and packages, respectively.

We selected this system because it is a medium-sized system, had no architecture documentation, and developers showed interest in using our method to produce documentation for future project developers as well as to keep its evolution organized in terms of code structure.

Before executing our method on MDD4ABMS, we asked the two current project developers to sketch the architecture of the system, indicating its modules and allowed dependencies (rules). As result, they listed 14 rules. Then, we recovered architecture rules using the WGB method, which identified 44 rules. Based on these rules, we interviewed the developers, asking them three main questions for each rule, shown as follows.

1. Is this rule useful to understand the system organization?
2. Should this rule be documented in the architecture?
3. How do you evaluate the granularity of this rule?

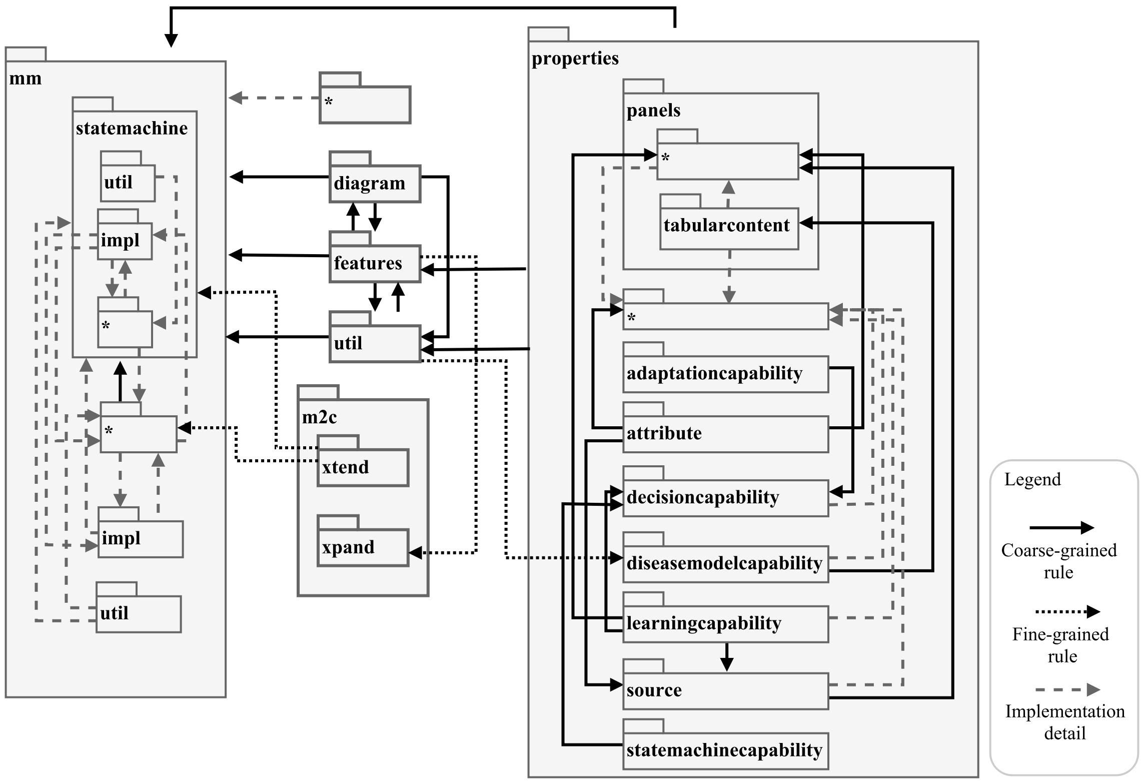
Moreover, to understand the rationale behind the answers, we asked developers to justify them.

We present the resulting architecture documentation in Figure 5.1. It shows the packages referred in rules (which are architecture modules), with rules represented with arrows. We also distinguish rules according to the classification provided by the developers (coarse-grained, fine-grained, or implementation detail).

Almost all of the recovered rules were considered useful to understand the system organization, being only 2 (out of 44) considered useless. Developers considered useful rules that convey information about dependencies that should actually occur and are allowed. For example, rules within the `properties` module reflect dependencies that semantically exist. The two rules that were considered useless are `mm.*` $\rightarrow$ `mm.impl` and `mm.statemachine.*` $\rightarrow$ `mm.statemachine.impl`. These are dependencies caused by a library to automatically generate code. Thus, they are not related to the MDD4ABMS business logic.

From the 44 recovered rules, 26 (59%) were selected to compose the architecture documentation of the system, which are the coarse-grained and fine-grained rules in the diagram. From these, 10 are coarse-grained rules that match conceptual rules informed by developers. The 4 remaining conceptual rules were also recovered, but in a finer granularity, i.e. they refer to sub-modules of the modules referred in conceptual rules. For instance, there is a conceptual rule `m2c` $\rightarrow$ `mm` that is implemented using the *xtend* library that caused WGB to recover the rule `m2c.xtend` $\rightarrow$ `mm.statemachine`. The

Figure 5.1: MDD4ABMS Architecture Rules Recovered by the WGB Method and Classified by System Developers.



other 12 rules that are included in the architecture documentation were not initially listed as rules by the developers, but they considered these rules important to be documented. They are mostly related to dependencies between `properties`'s sub-modules and between `mm`'s sub-modules.

The rules classified as implementation details are those that, according to the developers, should not be part of the architecture documentation. The reason for such a classification is that these rules reflect current implementation patterns in the code, but they are not considered architecturally relevant by the developers. First, rules within the `mm` module are implementation patterns but other dependencies that currently do not occur are allowed. Second, dependencies to the classes directly in the `properties` and `panels` packages (that is, in their $\star$ sub-packages) can be omitted because developers assume that classes in sub-packages can access the classes in the $\star$ package (i.e. in their parent package). Our validation also captured one violation ($\star \rightarrow$ `mm`) that should be refactored.

Finally, we asked the most experienced developer to generally evaluate the documentation generated by our method, using a 7-point Likert scale. This developer (strongly) agreed that the generated diagram (1) provides useful information, (2) provides accurate information, (3) reflects what is implemented in the code, (4) will be used as architecture documentation, and (5) facilitates the understanding of the system organization.

5.2 Offline Study

Given the case study presented in the previous section, in this section, we provide a quantitative analysis of the WGB method. We evaluate it to understand the gap between the conceptual rules and the rules extracted using the WGB method analyzing the efficient and the effectiveness of our method to recover architecture rules.

We organized this section as follows. Section 5.2.1 detail the settings of our user study. In Section 5.2.2, we present the results and discuss the findings derived from them. Finally, we present the threats to the validity of our study presenting actions adopted to reduce their impact on our study in Section 5.2.3.

5.2.1 Study Settings

To present the offline study that evaluates the WGB method, we used the same structure of the study detailed in Chapter 3. The GQM paradigm was adopted to present the goal and questions of our study. Next, we detail the procedure used to conduct this study based on the goal and research question. Finally, we presented the subject systems selected to evaluate the WGB method.

5.2.1.1 Goal and Research Questions

We adopted the widely used Goal-Question-Metric (GQM) (SOLINGEN et al., 2002) paradigm to design our study. We thus structured our study stating a goal, asking research questions related to this goal, and metrics that provide the necessary information to answer our research questions. Our goal, according to the GQM template is the following.

To assess our proposed WGB method, *evaluate recovered implemented architecture rules* from a perspective of *the researcher* as they are extracted from the source code using the method in a *multi-project study*.

Based on our goal, we derived two research questions listed below.

RQ1. How *efficient* is our WGB method?

RQ2. How *effective* is our WGB method to recover implemented architecture rules?

With RQ1, we are interested in evaluating our method *performance* to verify the feasibility of our method in practice, as it involves solving an optimization (i.e. combinatorial) problem that is solved using an algorithm with an exponential time complexity. To answer RQ1, we measure the metric M1, described in Table 5.1. With RQ2, we focus on the *quality* of the recovered rules. A key issue to answer this research question is to have a ground truth or gold standard to be compared with rules recovered by our method. There are two main types of information we have available: (i) code dependencies extracted from the source code; and (ii) conceptual rules provided by software developers. Therefore, we measure the quality of recovered rules using both types of information. The former is used to calculate metric M2, and the latter is used to calculate M3, M4 and M5, described in Table 5.1. Metric M5 allows us to make a qualitative analysis of recovered rules, classified according to types, which are introduced later.

There is a need for a qualitative analysis of the rules because the metrics M2, M3 and M4 cannot be interpreted as traditionally, i.e. the higher, the better. Consequently, they are used not as an indication of how good the method is but to understand the divergences between conceptual rules (manually recovered from subject systems) and implemented rules (recovered by our method). Implemented rules have no ground truth, because any set of rules that is in accordance with the code would be correct. The issue is to identify rules that express the highest granularity level as possible and provide useful information about the implemented dependencies. Therefore, using conceptual rules as a reference point, recovered implemented rules that are true positives or true negatives mean positive results, because they are consistent with the code and at a level of granularity that matches what was conceptualized by developers. However, false positives or false negatives point out divergences between conceptual and implemented rules that must be inspected to understand if the granularity level of implemented rules is adequate according to its expected characteristics. This inspection is done using M5.

Note that our research questions do not focus on comparing our method with existing work. Although a baseline would improve our evaluation, there is a lack of approaches that provide solutions similar to ours. Existing approaches can be used in a complementary way, but neither have the same goals nor provide the same outputs. Moreover, our

Table 5.1: Offline Study Metrics.

Metric	**Research Question**	**Description**
M1	RQ1	Time in seconds to execute our method with a given software project.
M2	RQ2	Ratio between the number of recovered rules and source code dependencies.
M3	RQ2	Precision of recovered rules, with respect to conceptual rules.
M4	RQ2	Recall of recovered rules, with respect to conceptual rules.
M5	RQ2	Number of recovered rules by type, categorized in relation to conceptual rules.

evaluation of efficiency does not focus on showing that our method is more scalable than another, but its ability to execute in an acceptable time.

5.2.1.2 Procedure

The key steps of our offline study are detailed next. In short, we recover the conceptual architecture of out target systems. Then, we execute the WGB method using the source code of the subject system and calculate the results of our study. We provide details of each of the steps of our study procedure as follows.

Manually Recover System Architecture. To calculate some of our metrics, we must know the conceptual rules of our subject systems. Therefore, our first step consists of manually recovering the architecture of these systems. This was accomplished based on previous architecture documentation and the knowledge provided by key software architects and developers. We focus on identifying architecture modules and rules based on a particular level of the package hierarchy of the system (assuming the Java language). The tasks to manually recover system architecture are the same as presented in our evaluation of the gap between conceptual rules and source code dependencies in Section 3.1. As a result of this step, we obtained modules and conceptual architecture rules, which comprise the architecture of our subject systems.

Execute the WGB Method. Our method was executed with each of our subject systems, detailed in next section. The method was implemented using a set of available tools. Source code dependencies were extracted using the Classycle[1] Plugin for Eclipse,

[1]Available at: <http://classycle.sourceforge.net/>

which automatically provides an XML file with each class and package dependency for Java projects. To solve the optimization problem, we used an academic version of the IBM ILOG CPLEX Optimization Studio[2]. Additional scripts were written to integrate these tools and implement our method[3].

Calculate Metrics and Analyze Results. Using the manually recovered system architectures and results of our method execution, we calculated our metrics presented in Table 5.1, and analyzed them.

5.2.1.3 Subject Systems

In this study, we selected the subject systems based on the feasibility to manually recover their architecture, and selecting projects that varied in size and architecture organization. Given that the requirements of the subject systems are the same of the empirical study previously presented in Chapter 3, we selected the same subject systems to this evaluation. Therefore, we use the same systems with the same conceptual architectures and source code detailed in Section 3.1.3.

5.2.2 Results and Analysis

We next present the data collected in our study, and analyze and discuss results, by research question.

5.2.2.1 RQ1: Efficiency of WGB method

Following our study settings, we executed our WGB method with the six subject systems, measuring the time it took to provide an output (metric M1). We report the obtained results in Table 5.2, which details time in seconds taken (in total and by each method step) to execute our method. Table 5.3 complements these results with additional data about the systems, which affect these results. Pairwise dependencies are dependencies between packages represented in the way required by our method. Clustered dependencies are those that remain after the pairwise clusterization (step 2). Conflicts correspond to the dependencies that lead to redundant rules, from which some must be selected resulting in our recovered rules.

[2]Available at: <https://www.ibm.com/products/ilog-cplex-optimization-studio>
[3]Available at: <https://www.inf.ufrgs.br/prosoft/projects/wgb-method>

Table 5.2: Time taken to Execute in Seconds (M1).

Subject System	Step 1 (s)	Step 2 (s)	Step 3 (s)	Total (M1)
ArchStudio	44.17	1.19	0.19	45.55s
AspectJ	19.69	0.74	0.14	20.57s
EC	0.38	0.02	0.08	0.48s
Metrics	0.22	0.01	0.07	0.30s
OLIS	0.19	0.01	0.07	0.27s
RecSys	2.92	0.16	0.11	3.19s

Table 5.3: Number of Graph Arrows by Method Step.

Subject System	Pairwise Dependencies	Clustered Dependencies	Conflicts	Recovered Rules
ArchStudio	1872	869	3726	324
AspectJ	1552	684	2021	239
EC	180	88	294	35
Metrics	220	59	260	19
OLIS	204	78	125	27
RecSys	784	292	1350	117

Based on the analysis of our results, it is possible to observe that, for all systems, we obtained acceptable execution time results, i.e. lower than one minute. Small systems (EC, Metrics, and OLIS) are associated with an execution time lower than one second, while with our largest subject system, namely ArchStudio with 236.9 KLOC, the WGB method takes 45.5s to execute. Although these results are good, the time taken largely increases as the number of pairwise dependencies, which is the key factor that influences the processing time, is higher. For example, ArchStudio has approximately 10 times more dependencies than EC, but the method took 95 times more time to execute (0.45s vs. 45.5s). Nevertheless, considering that the absolute time taken is low and ArchStudio can be considered a medium-large project, results indicate that our WGB method is feasible to be used in practice, with industry-sized software systems.

By looking at the time taken to execute individual steps, it is possible to see that step 1, which calculates the MDS metric, is responsible for this increase in the time. For the larger systems, approximately 95% of the total time corresponds to this step. This is mainly due to the many components associated with our metric which, to analyze a pair of modules, takes into account neighbor modules. Our results could be significantly improved if we had used an auxiliary structure to store intermediate values, which are calculated many times. However, given that the absolute time taken is low, such improvements are currently not implemented.

Note that a step that could be critical in our method, involving the resolution of an optimization problem that has an exponential time algorithm in the worst-case scenario, had no significant impact on results and scaled well. This is due to the efficiency of the tool, i.e., the IBM ILOG CPLEX, used to solve the problem instances.

The results of our method efficiency show that it executed in less than one minute considering all systems. Therefore, we consider it efficient enough to be adopted in the software development. As expected, the time taken to run increases as the number of dependencies of the system increases.

5.2.2.2 RQ2: Effectiveness of WGB method

As discussed in the description of our study settings, it is a challenge to evaluate our recovered architecture rules because of the lack of a ground truth of implemented rules. Therefore, we analyze our recovered rules from three perspectives: (i) how much our method is able to abstract from dependencies among modules; (ii) the relationship between the recovered (implemented) rules and conceptual rules; and (iii) the qualitative nature of recovered rules using conceptual rules as a basis.

Dependency Abstraction. Implemented architecture rules reflect dependencies that actually occur in the source code. However, due to the many dependencies, it is practically unfeasible for a developer to analyze all of them. Our method thus aims at reducing the number of dependencies by abstracting them into a set of coarser-grained rules, when appropriate. M2 assesses this reduction provided by our method.

Table 5.4 details how much our method abstracted from dependencies of our subject systems. On average, the reduction from module (package, in our subject systems) dependencies to recovered rules is of 87.6% (SD = 0.7). Consequently, there is a *large reduction in developers' effort to analyze dependencies* in the form of implemented rules recovered by our method. Moreover, the standard deviation in this reduction is low, indicating that our method provides such a reduction consistently across all subject systems.

How this dependency abstraction occurs throughout the execution of our method can be seen in Table 5.3. The method starts with pairwise dependencies (and associated dependency strength) represented as described in the first method step. With pairwise clusterizations, the second step reduces such dependencies in more than a half. Such clustered dependencies have many conflicts (i.e. represent redundant information) and,

Table 5.4: Dependency Abstraction (M2).

Subject System	**Pairwise Dependencies**	**Package Dependencies**
ArchStudio	82.7%	87.4%
AspectJ	84.6%	86.6%
EC	80.0%	88.7%
Metrics	91.4%	87.8%
OLIS	86.8%	87.8%
RecSys	85.1%	87.4%
AVG	85.1%	87.6%
SD	3.9%	0.7%

after the execution of the third method step, we have the recovered rules, which are associated with the reported reduction with respect to package dependencies and also to pairwise dependencies (AVG = 85.1%, SD = 3.9).

Recovered Implemented Rules vs. Conceptual Rules. As discussed in our study procedure, there is no ground truth to which we can match our recovered rules against. We then use conceptual rules as a basis for comparison and calculate obtained precision and recall. This is done using two approaches. First, we use the *syntax* of conceptual rules, i.e. we check which recovered rules exactly match conceptual rules. Precision, in this case, is the fraction of recovered rules that are conceptual, while recall is the fraction of conceptual rules that were recovered. Second, we use the *semantics* of conceptual rules. A set of architecture rules, regardless if they are conceptual or implemented, has implications in the allowed module dependencies: from the set of rules, it is possible to determine whether the dependency between two modules is allowed or forbidden. Consequently, in this second approach, we check whether the recovered rules have the same implications as the conceptual rules. Precision is then the fraction of module dependencies allowed by recovered rules that are also allowed by conceptual rules, while recall is the fraction of module dependencies allowed by conceptual rules that are also allowed by recovered rules.

The precision and recall calculated using these two approaches are shown in Table 5.5. As can be seen, values largely differ across different systems and most of them are not high. *Low values are in fact expected for precision and recall* because implemented architecture rules capture what is actually in the code while the conceptual architecture hides information that is implemented in the source code.

Table 5.5: Precision (M3) and Recall (M4).

System	Exact Matching		Allowed Dependencies	
	Precision	**Recall**	**Precision**	**Recall**
ArchStudio	0.6%	4.4%	14.3%	8.8%
AspectJ	2.9%	25.9%	14.8%	41.9%
EC	22.9%	42.1%	78.8%	40.2%
Metrics	10.5%	25.0%	13.7%	61.9%
OLIS	22.2%	46.2%	84.7%	37.2%
RecSys	6.8%	42.1%	16.2%	36.8%
AVG	11.0%	31.0%	37.1%	37.8%
SD	9.6%	15.8%	34.7%	17.0%

Low precision and recall values indicate different types of divergences between conceptual and recovered rules. Low precision means that there are many recovered rules capturing code dependencies that are not allowed according to conceptual rules, meaning that there are many undocumented rules and architectural violations. When the implemented code reflects the conceptual architecture, higher values are obtained, as it is the case of OLIS and EC. Low recall, in turn, means that there are many specified conceptual rules that are not actually implemented in the code. It may not necessarily be due to divergences that occur between conceptual rules and code dependencies, but because these rules may be too general, allowing many dependencies that should not be allowed, as in our example, in which the rule must allow the *Presentation* module to depend on the *Service* module only, rather than the whole *Business* module.

Differences between conceptual and recovered rules are extremely large in our largest subject system, ArchStudio, which has both low precision and recall. This shows a large mismatch between the conceptual architecture, which has an available ground truth (LUTELLIER et al., 2015), and the implemented architecture. Although the difference is large, the number of allowed dependencies differs only in 6.7% in this system (see Table 5.6), providing further evidence of this mismatch. Table 5.6 indicates the number of dependencies that are allowed, considering every pair of modules. In general, positive differences lead to lower precision (due to occurrences of dependencies inconsistent with conceptual rules), while negative differences lead to lower recall (due to too general conceptual rules).

Nature of Recovered Implemented Rules. To better understand the precision and recall values detailed above, we now make a qualitative analysis of recovered implemented rules, using conceptual rules as a baseline. We classified each of the recovered

Table 5.6: Dependencies Allowed by Architecture Rules.

System	Conceptual		Recovered		Difference	
	#	%	#	%	#	%
ArchStudio	16764	17.2	10257	10.5	-6507	-6.7
AspectJ	1398	3.2	3964	9.0	+2566	+5.8
EC	756	35.3	386	18.0	-370	-17.3
Metrics	63	11.3	284	50.9	+221	+39.6
OLIS	701	45.2	308	19.8	-393	-25.4
RecSys	923	13.1	2097	29.7	+1174	+16.6

rules into five identified categories, which are associated with the relationship between the recovered rule and a conceptual rule. These categories are described next.

Match (CONCEP). A recovered rule that exactly matches a conceptual rule. For example, if a conceptual rule is $X \dashrightarrow Y$, the recovered rule is the same.

Generalization (SUPER). A recovered rule that is a generalization (super-rule) of a conceptual rule. For example, if a conceptual rule is $X_C \dashrightarrow Y$, the recovered rule is $X \dashrightarrow Y$.

Specialization (SUB). A recovered rule that is a specialization (sub-rule) of a conceptual rule. For example, if a conceptual rule is $X \dashrightarrow Y$, a recovered rule can be $X_C \dashrightarrow Y$.

Intra-module (INTRA). A recovered rule that states allowed dependencies between non-represented sub-modules of a module in the conceptual architecture. An example is when a conceptual architecture shows X as a module with no sub-modules, and thus dependencies within X are allowed. An intra-module recovered rule is, for example, $X_{C_1} \dashrightarrow X_{C_2}$.

Mismatch (MIS). A recovered rule that is not in accordance with conceptual rules. For example, there is no conceptual rule that is $X \dashrightarrow Y$ (and no coarser-grained rule, if X or Y have parents), when there is such a recovered rule. This can be either a forbidden dependency (architecture violation) or an undocumented rule.

The classification of recovered rules according to these categories is shown in Table 5.7, which confirms our general analysis above, based on precision and recall. The precision of most of the systems (ArchStudio, AspectJ, Metrics and RecSys) is low due to the presence of a high amount of dependencies captured in recovered rules that are not allowed according to conceptual rules. More than a half of recovered rules fall into the MIS

Table 5.7: Number of Recovered Rules by Type (M5).

System	**CONCEP**		**SUPER**		**SUB**		**INTRA**		**MIS**	
	#	%	#	%	#	%	#	%	#	%
ArchStudio	2	0.6	1	0.3	39	12.0	79	24.3	203	62.7
AspectJ	7	2.9	5	2.1	11	4.6	56	23.4	160	66.9
EC	8	22.9	2	5.7	9	25.7	9	25.7	7	20.0
Metrics	2	10.5	3	15.7	3	15.7	0	0.0	11	57.9
OLIS	6	22.2	1	3.7	10	37.0	4	14.8	6	22.2
RecSys	8	6.8	0	0.0	7	6.0	28	23.9	74	63.2
AVG		11.0		4.9		16.8		18.7		48.8

category. For the systems in which a high precision is achieved, there is a higher number of rules that matches conceptual rules, but also sub-rules (i.e. specializations). The latter compromises recall, because they are more restrictive than the original rule. Intra-module rules are also responsible for lowering recall. These are rules not captured by conceptual rules, because dependencies between non-represented sub-modules are allowed. In contrast, recovered rules are able to identify how dependencies within modules occur, thus being more restrictive than conceptual rules from an intra-module perspective.

By analyzing the cases in which our method made generalizations (super-rules) and specializations (sub-rules) of conceptual rules, we observed that they are all plausible, and considered correct in our view, that is, the method has the expected behavior. For example, ArchStudio has one conceptual rule for each child of the module *Comp*, indicating that it can depend on the *Util* module. Our method, in contrast, derived a rule *Comp* $\rightarrow$ *Util*, which gives the notion that all children from *Comp* can depend on *Util*. Another scenario of generalization occurs when a parent module has a single child, the *Touchgraph* module that has *Graphlayout* as an only child in Metrics. As result, in our method, arrow weights are the same in this case—*Touchgraph* $\rightarrow$ *Core* and *Graphlayout* $\rightarrow$ *Core* have equal weights—and our method gives priority to the most generalized rule (i.e. *Touchgraph* $\rightarrow$ *Core*).

An example from OLIS illustrates a case of specialization. The conceptual rule *Business* $\rightarrow$ *Agent* allows dependencies between all *Business*'s sub-modules to all *Agent*'s sub-modules. However, the *Agent* module uses a façade pattern, so all dependencies occur from *Business*'s sub-modules to the *Agentlayerfacade* module, an *Agent*'s child. The rule recovered by our method is, in this case, *Business* $\rightarrow$ *Agentlayerfacade*. Although in all these cases the recovered rule is in accordance with the expected method behavior, both conceptual and recovered rules are adequate, and the final choice for the most suitable representation is a subjective developers' decision.

Concerning the effectiveness, our method achieves a reduction from module dependencies to recovered rules of 87.6%, on average, thus reducing most of the developers' effort. The comparison of recovered implemented rules with conceptual rules shows that they largely differ, leading to 37.1% and 37.8% of precision and recall, respectively, on average. A qualitative analysis of recovered rules shows that our method: (i) generalizes many rules associated with sub-modules of a module as a single super-rule; (ii) is able to capture rules that occur specifically between sub-modules, being often sub-rules of conceptual rules; (iii) identifies rules that govern dependencies within a module, which are typically not specified as conceptual rules; and (iv) leaves architectural violations as fine-grained rules, so that it is easier to distinguish them from other recovered rules using our dependency strength metric.

5.2.3 Threats to Validity

We identified two main tasks performed while elaborating our study design that may affect results. These consist of threats to the study validity, and we describe actions done to mitigate them as follows. The first is an external threat, which refers to the selection of our subject systems. Although in many studies a set of large-scale systems would bring more generalizable results, in our case this would not show how our method behaves with systems with different characteristics. Consequently, our selected systems vary in many aspects, such as development environments, adopted architecture patterns, domain, and size. The variation in our obtained rules reflect these different explored scenarios. The second threat is associated with how conceptual rules were manually recovered from subject systems. As previously mentioned, different developers and architects may have different views of a system's architecture. In order to mitigate this threat to construct validity, we consulted architects and developers directly involved in the development of each subject system, comparing their own views regarding the system's architecture with available documentation, following the guidelines introduced by Garcia, Ivkovic and Medvidovic (2013).

5.3 User Study

In the previous studies presented in this chapter, we analyzed the WGB method using the conceptual architecture rules as a baseline to evaluate it, assuming that those are the most appropriate representation of the architecture. Aiming to investigate whether the architecture rules extracted using the WGB method could improve the architecture documentation, we now present a user study. This user study enriches our previous analysis examining the reasoning of the developers about the architecture rules recovered using the WGB method. We detail the settings of our user study in Section 5.3.1, present and discuss its results in Section 5.3.2, and point the threats to validity of our study in Section 5.3.3.

5.3.1 Study Settings

The structure of the user study in this section is similar to that adopted in our previously described studies. We present in this section details of the user study design adopting the GQM paradigm. Then, we describe the procedure used to conduct this study and adopted materials. Finally, we present details of the subject systems and the study participants.

5.3.1.1 Goal and Research Questions

We designed our user study by formulating its goal, research questions to achieve it and the metrics that provide the necessary information to answer our research questions. Our goal, according to the GQM template, is the following.

> To *evaluate the proposed WGB method*, *analyze the architecture rules recovered by this method* from *the perspective of the developers* as they are extracted from the source code using the WGB method in an *commercial multi-project study*.

Based on our goal, we derive three research questions (RQs), which are described below, and the metrics associated with each research question. The metrics are measurements of the degree that the WGB method and its recovered rules satisfy a set of criteria. These criteria are listed in Table 5.8.

RQ1. How does the WGB method *improve* the abstraction of architecture rules?

RQ2. How does the WGB method *reveal* architectural misunderstandings and violations?

RQ3. How do developers *perceive the usefulness* of the WGB method to the system architecture?

By answering RQ1, we assess whether the level of abstraction of the rules recovered using the WGB method is appropriate to architecture documentation. With RQ2, we investigate how the rules recovered by the WGB method can identify violations, undocumented rules, and allowance of prohibited dependencies. The answers to R1 and R2 help understand the usefulness of each rule recovered by the WGB method. To analyze the architecture that results from the WGB retrieved rules, we collect 11 metrics to answer RQ3, which aims to assess the usefulness and the characteristics of this resulting architecture. All the metrics associated with the RQs are collected using questionnaires, which are further detailed, asking the developers of the systems about their opinions.

5.3.1.2 Procedure

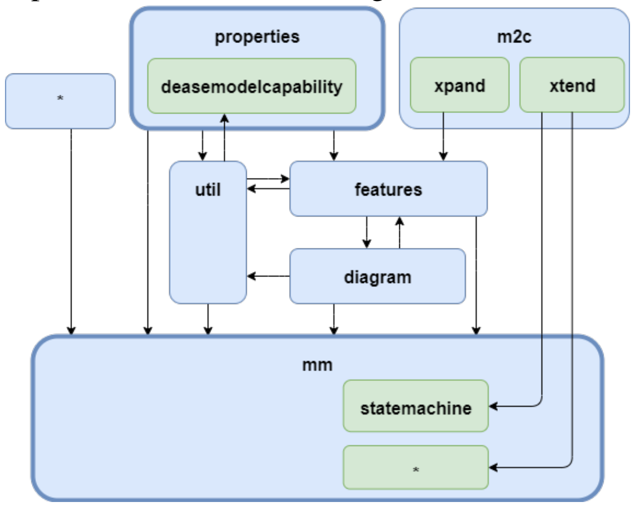
In this section, we describe the steps of the user study procedure. In short, we first apply the WGB method in the selected subject systems to extract their architecture rule. Then, we ask developers of the subject systems to manually recover the architecture of each subject system and its rules. To evaluate the architecture built based on the rules extracted by the WGB method, we instantiate questionnaires for each subject system based on a set of template questions that should be answered by the system developers. We provide further details of each of the steps of our study procedure as follows.

Recovery of Architecture Rules of the Subject Systems using the WGB Method. To obtain the architecture rules of the subject systems using the WGB method (WGB rules), we execute our method, as presented in Chapter 4 and using the same implementation presented in Section 5.2. Based on the WGB rules of each system, we provide a documentation that is similar to the architecture documentation of the case study described in Section 5.1. To exemplify the rule visualization, we present the documentation of MDD4ABMS in Figure 5.2. We use rectangles and arrows to represent modules and rules, respectively. Additionally, the documentation has a color to each granularity level, e.g. in Figure 5.2a, the modules `properties` and `m2c` are in blue because they are at the same level of granularity, and their

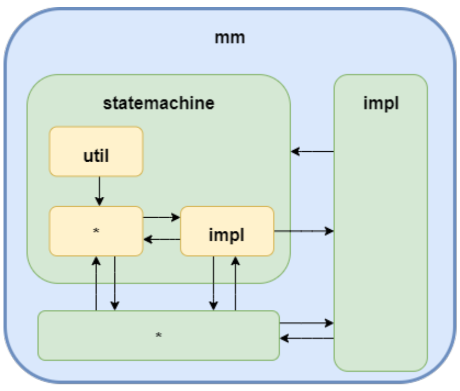
Table 5.8: Criteria Evaluated in the User Study.

Metric	Research Question	Description
Granularity	RQ1	The adequacy of the rules extracted using the WGB method as part of the architecture documentation considering the level of granularity.
Accuracy	RQ1, RQ2	The adequacy of the rules extracted using the WGB method as part of the architecture documentation considering the the accuracy.
	RQ3	The rules extracted using the WGB method are accurate.
Understanding	RQ1, RQ2	The adequacy of the rules extracted using the WGB method as part of the architecture documentation considering the usefulness for developers to understand the system.
Implementation	RQ1, RQ2	The adequacy of the rules extracted using the WGB method as the architecture documentation considering the usefulness for developers to implement the system.
Violations	RQ2	The rules extracted using the WGB method represent an architectural violation.
Violation Allowance	RQ2	The rules extracted using the WGB method allow dependencies that are prohibited in the architecture.
Usefulness	RQ3	The rules extracted using the WGB method provide useful information.
Organization	RQ3	It is easy to understand how the system is organized analyzing the rules extracted using the WGB method .
Dependencies	RQ3	It is easy to understand the system dependencies analyzing the rules extracted using the WGB method.
Allowed Dependencies	RQ3	The rules extracted using the WGB method show dependencies that are allowed in the system.
Forbidden Dependencies	RQ3	The rules extracted using the WGB method show dependencies that are forbidden in the system.
Coarse-granularity	RQ3	The rules extracted using the WGB method are an overly coarse-grained representation of the system.
Fine-granularity	RQ3	The rules extracted using the WGB method are an overly fine-grained representation of the system.
Conformance	RQ3	The rules extracted using the WGB method capture what is implemented in the system's source code.
Use Intent	RQ3	The documentation built based on the rules extracted using the WGB method could be used as an architecture model of the system.
Appropriateness	RQ3	The documentation built based on the rules extracted using the WGB method is more adequate for developers to understand and evolve the system than the documentation manually recovered.

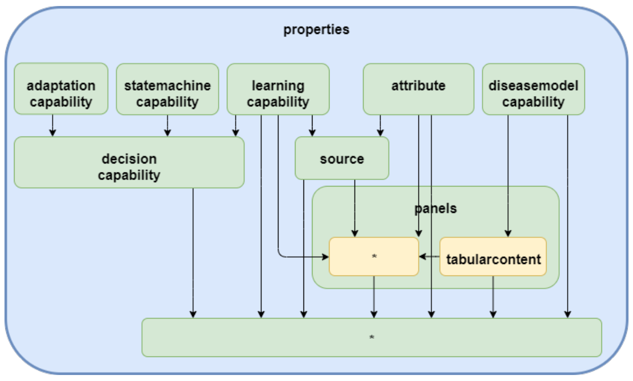
Figure 5.2: Example of Documentation using the WGB Rules of MDD4ABMS.



(a) Complete architecture of MDD4ABMS



(b) MM module of MDD4ABMS.



(c) Properties module of MDD4ABMS.

internal modules `deasemodelcapability`, `xpand` and `xtend` are in green because they are also at the same level of granularity. In the cases where the visualization of the rules has many modules and rules, we present the documentation of some modules in a separate document. For instance, the MDD4ABMS has two partial documentations presenting the internal modules and rules of module `MM` and `properties` in Figure 5.2b and 5.2c respectively. The modules that have a partial documentation has a thicker border in all documents, such as the modules `properties` and `mm` in Figure 5.2a. Using this representation of the architecture, we provide two documentations of each system: one that is built using the WGB rules, and another manually recovered by the developers.

Collection of the Demographic Information of the Participants. The first step involving the participants consists of asking their consent to participate in this study and collecting their demographic information using a questionnaire. We ask about personal information, their knowledge on software development, and their experience working on the systems.

Tutorial on Software Architecture. In order to have a common understanding of what a software architecture is and using a standard notation to represent modules and architecture rules, we provide a brief tutorial on software architecture to the participants. We introduce a software architecture as a set of design decisions that govern a software system. It includes the structuring of the system as a hierarchy of modules (each having a role) and rules that specify allowed dependencies. Modules are represented as squares and rules are arrows between modules.

Recovery of Architecture Rules of the Subject Systems by the Developers. To obtain the architecture rules from the developers' perspective, we ask the participants to document the software architecture of the subject systems. They perform this task as a group and are able to access any data available, such as the source code of the systems. During this task, one researcher is available to provide support to the participants in case of any problems or doubts. Moreover, there is no time limit to this task. The purpose of this task is that they discuss and decide which modules and rules should be included in the architecture. The developers can also choose how to document the architecture rules, i.e. there is no specific method or tool to document the architectures. For instance, they may use the blackboard or any CASE tool they prefer. As a result of this step, we have an architecture documentation with a set of rules based on the developers' perspective (*DEV rules*) for each subject system.

Questionnaires of Evaluation of the WGB Method. To evaluate the WGB rules and compare them with the DEV rules, we formulate a questionnaire to be answered by the participants. This questionnaire varies according to the quantity and the types of rules extracted using the WGB method and the rules recovered by the developers. Therefore, we use a questionnaire template with a set of questions and comparisons that is instantiated according to the rules of each system. We present details of the questionnaire template as follows. Given the instantiated questionnaire, we ask the participants to answer it individually. As a result of this step, we obtain the answers to the questionnaires evaluating the documentation built based on the WGB rules and each rule extracted using the WGB method individually.

5.3.1.3 Questionnaire Template

We formulate a questionnaire template to instantiate the questionnaires to collect the opinion of the participants about the rules recovered by WGB. We split the questionnaire template into three parts. The first and third parts of the template aim to collect data about how the developers perceive the usefulness of the documentation built using the WGB rules (RQ3), while the second part focuses on compare the WGB rules and the DEV rules (RQ1 and RQ2). To evaluate the effect of the inspection of the rules of the second part in the answers of the participants, the first and third part of the template have the same questions. Therefore, we can compare the answers and analyze whether and why participants change their opinion about the architecture built based on the WGB rules.

In the beginning of the first part, we provide the architecture documentation built using the WGB rules of the subject system under analysis. Both these parts have ten questions, which are associated with the metrics of RQ3. These questions are about the participant agreement with the phrase *"Analyzing WGB architecture model, please indicate how much you agree with the sentences below."* followed by the sentences associated with one of the RQ3's metrics, e.g. the sentence associated with accuracy is *"It is accurate"*. In the end of the third part, we ask the participants their agreement on the statement *"The WGB architecture model is more appropriate for developers to understand and evolve the system than the model recovered by the developers."*. This question is related to the appropriateness, which is also a metric associated with RQ3. These questions are objective and can be answered based on a 7 point Likert scale ranging from strongly agree to strongly disagree. Additionally to the objective questions, these parts have open-ended questions at the end asking the participant to justify his answers. To instantiate these parts of the template, the questions do not change, but the provided documentation changes based on the system under evaluation.

The second part of the questionnaire template focuses on comparing the WGB rules with the DEV rules in a case-by-case basis. We compare rules that refer to the same source code dependencies, but with different granularity, e.g. the developers recovered a rule allowing dependencies from module `A` to module `B` while the WGB method extracted a rule allowing dependencies from a module `A.1`, which is a sub-module of `A`, to module `B`. The questions in this part of the template are associated the criteria related to RQ1 and RQ2, listed in Table 5.8. We ask the participants about the correctness, preference, and existence of violations in each pair of rules presented. Given that we compare the WGB rules with the DEV rules, the number of comparisons of each system varies according to the recovered rules, and how they are related to each other. In the template, we have six types of comparisons, which are derived from the results of our previous studies. These types of comparisons are exemplified in Figure 5.3 and described below.

Specialization comparisons (Figure 5.3a) have the rules recovered by the developers with a higher granularity level than the rules extracted by the WGB method.

Generalization comparisons (Figure 5.3b) have the rules recovered by the developers with a lower granularity level than the rules extracted by the WGB method.

Explicit comparisons (Figure 5.3c) are those in which the developers explicitly document rules internal to a module, while WGB does not recover any rule (indicating that dependencies within the module are allowed).

Implicit comparisons (Figure 5.3d) are those that the developers do not document any rule (indicating that dependencies within the module are allowed), while WGB explicitly recovers rules internal to a module.

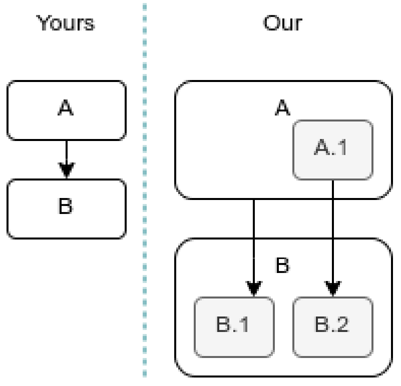
Non-conformance comparisons (Figure 5.3e) are those documented by the developers and have no corresponding rule recovered by WGB.

Missed comparisons (Figure 5.3f) are those that have rules recovered by WGB but no corresponding rule documented by the developers.

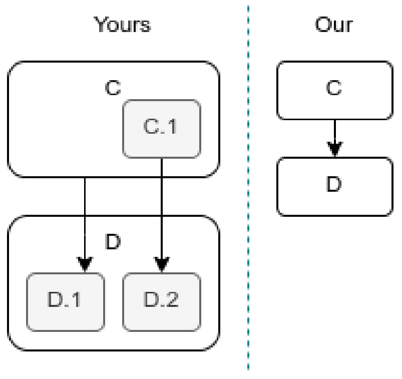
In Figure 5.3, we show the comparisons as they are presented in questionnaires to the participants. The rules recovered by the developers are labeled with *Yours* and placed on the left-hand side, and the rules extracted using the WGB method are labeled with *Our* and placed on the right-hand side.

For each of the comparisons of the second part of the questionnaire, we ask the participants about the granularity, accuracy, understanding, and implementation. The questions are about the appropriateness of the rules to be part of the system architecture. The participants have six objective answers to these metrics: (i) both rules are equally appropriate (*WGB = DEV*); (ii) both rules are inappropriate (*!WGB !DEV*); (iii) the WGB rule is appropriate while that documented by the developers is inappropriate (*WGB !DEV*); (iv) Both rules are appropriate, but that recovered by WGB is more appropriate (*WGB > DEV*); (v) the rule documented by the developers is appropriate while that recovered by WGB is inappropriate (*!WGB DEV*); and (vi) Both rules are appropriate, but

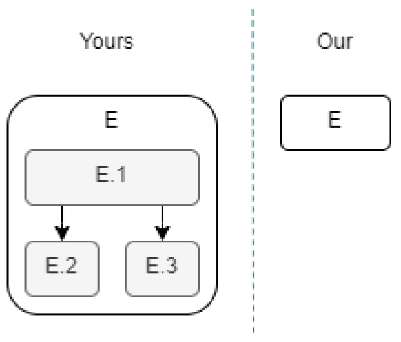
Figure 5.3: Types of Comparisons of the Questionnaire Template.



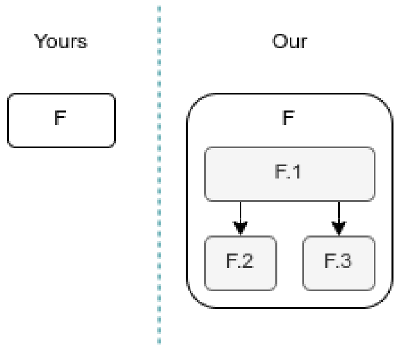
(a) Specialization.



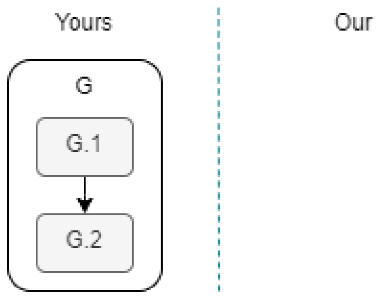
(b) Generalization.



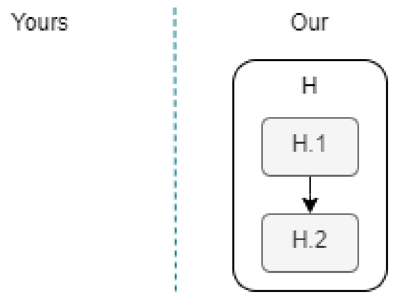
(c) Explicit.



(d) Implicit.



(e) Non-conformance.



(f) Missed.

that documented by the developers is more appropriate (*WGB* < *DEV*). With these objective answers, the participants show their opinion on the appropriateness and preference between both rules.

In addition, we also ask whether the rules consist of architectural violations (Violations) or allow prohibited dependencies (Violation Allowance), which are criteria associated with RQ2. Therefore, the participants answer about the presence of violations and the allowance of violations by the WGB rules, or by the DEV rules. They answer using a 5-point Likert scale ranging from definitely yes to definitely no. At the end of this part of the questionnaire, we ask them to justify their answers in an open-ended question.

All the questions of the questionnaire template can be seen in detail in Appendix B.

Table 5.9: Subject System Characteristics.

Name	**KLOC**	**Cl**	**Cl-Dep**	**Pkg**	**Pkg-Dep**
System A	30.3	494	1964	133	846
System B	57.1	835	3912	184	1920

Table 5.10: Age and Development Experience of the Study Participants.

	AVG	**SD**	**Min**	**Max**
Age	30.8	5.0	24	36
Development Experience	8.2	4.9	4	14
Experience in System A	3.2	1.6	1	5
Experience in System B	2.8	1.6	1	5

5.3.1.4 Subject Systems and Participants

In our user study, we investigate two commercial systems. In Table 5.9, we present the following information related to these systems: lines of code (KLOC); number of classes (Cl); the number of dependencies between classes (Cl-Dep); the number of packages (Pkg); and the number of package dependencies (Pkg-Dep). We perform this study analyzing these systems under a non-disclosure agreement, which does not allow sensitive or specific data of the systems to be revealed. Therefore, all data related to the subject systems are anonymized.

To have answers to our questionnaires, we ask the developers who currently evolve the systems to participate in our study. Their demographic information about software architecture, experience in these systems, and general information of the five participants of our study are presented in Tables 5.10 and 5.11. They have an average of 31 years old, all participants are male, four participants have an undergraduate degree, and one has a master's degree in information systems. The participants have at least four years of experience in software development and at least one year working on the subject systems. Their knowledge on software development, architecture, design, programming, and engineering is intermediate or advanced. In the questions about their knowledge about each system, they consider themselves intermediate or advanced in most of the topics.

Table 5.11: Knowledge of the Study Participants on Software Engineering and the Subject Systems

Topic	Knowledge	Basic		Intermediate		Advanced		Expert	
		%	#	%	#	%	#	%	#
Software Engineering	Development	0	0	20	1	60	3	20	1
	Architecture	0	0	60	3	40	2	0	0
	Design	0	0	20	1	80	4	0	0
	Programming	0	0	20	1	60	3	20	1
	Engineering	20	1	40	2	20	1	20	1
System A Architecture	Modules	0	0	40	2	60	3	0	0
	Rules	0	0	40	2	60	3	0	0
	Modules Implementation	20	1	40	2	40	2	0	0
	Most Familiar Modules	0	0	40	2	60	3	0	0
	Least Familiar Modules	20	1	80	4	0	0	0	0
System B Architecture	Modules	0	0	80	2	20	3	0	0
	Rules	0	0	60	3	40	2	0	0
	Modules Implementation	20	1	60	3	20	1	0	0
	Most Familiar Modules	0	0	60	3	40	2	0	0
	Least Familiar Modules	40	2	60	3	0	0	0	0

5.3.2 Results and Analysis

Given the study settings, we present the results of our study detailing the documentation of the subject systems, showing the answers to the questionnaires, and analyzing and discussing them.

5.3.2.1 Documentation and Questionnaires

Our study is based primarily on the analysis and comparison of the recovered WGB rules and those documented by the developers. The number of each retrieved set of rules is shown in Table 5.12. We present the number of rules recovered by the developers (DEV Rules), the number of rules extracted using the WGB method (WGB Rules), and the number of rules that are in both sets of rules (Equal Rules). The WGB method extracted seven rules recovered by the developers indicating that the WGB method can extract architecture rules that are similar to the rules recovered by the developers. However, similarly to the offline study results, the WGB method extracted more rules than the rules extracted by the developers.

Table 5.12: Number of Rules Documented by the Developers and Recovered by WGB

System	Dev Rules	WGB Rules	Equal Rules
System A	11	54	5
System B	16	252	2

Table 5.13: Types of Rule Comparison in the Questionnaire by Subject System

System	Specialization	Generalization	Implicit	Non-conformance	Missed
System A	1	1	4	0	3
System B	8	1	8	2	0

Based on the rules that are not in both sets of rules, we instantiate the questions of the second part of the questionnaire for each system. In Table 5.13, we present the number of comparisons by type for each subject system. We omitted the number of explicit comparisons because none occurred for both systems. Most of the comparisons are specializations or implicit because the rules extracted using the WGB method have lower level of granularity than the rules recovered by the developers.

5.3.2.2 RQ1: Improvement of the level of abstraction

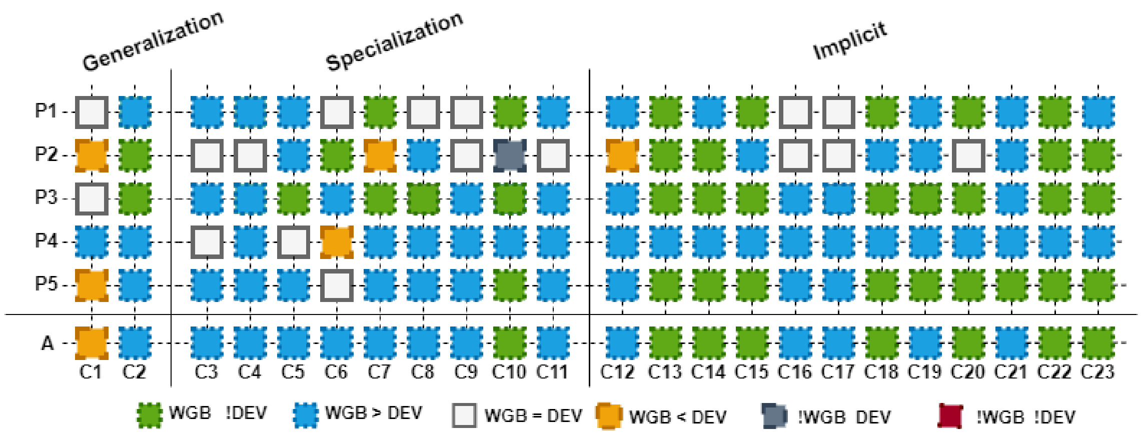
To understand the abstraction level of the WGB rules, we analyze the rules that are related to specializations (Figure 5.3a), generalizations (Figure 5.3b), and implicit (Figure 5.3d) comparisons. We investigate these types of comparisons because they show the differences in the level of abstraction of the WGB rules and the DEV rules. In Figures 5.4–5.7, we present the answers associated with RQ1 where:

- P1–P5 are the participants;
- A is the agreement between the participants;
- C1–C23 are the comparisons instantiated in the questionnaires; and
- The colored squares are the answers of the participants (lines) to the questions of the comparisons (columns).

An example of the information provided in Figure 5.4 is that, for comparison C3, the participant 1 (P1) answered that the WGB rule is more appropriate (WGB > DEV), which corresponds to the third blue square of the first line in the upper left corner.

Appropriateness of the WGB and the DEV rules. First, we analyze the answers to the four criteria associated with RQ1 regarding the appropriateness of the rules. Concern-

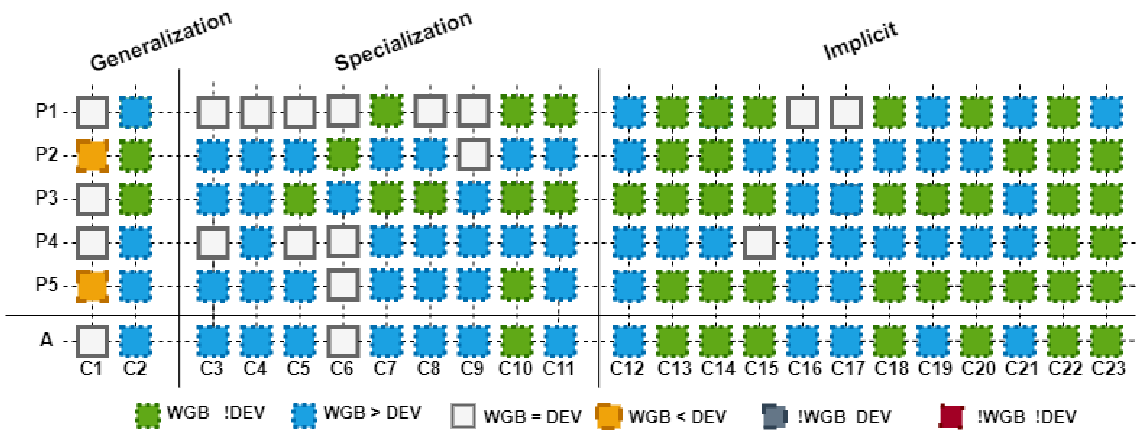
Figure 5.4: WGB and DEV Rule Comparison: Granularity



ing the granularity, 99.1% of answers report the WGB rules as appropriate. Moreover, 80.0% of the answers are that the WGB rules are better or the only appropriate rule in the comparison. Regarding the accuracy, the WGB rules are considered appropriate in all comparisons, and 83.5% of the answers report the WGB rules as better than the DEV rules or the only appropriate rule in the comparison. The implementation criteria has 96.5% of the answers indicating the WGB rules as appropriate, and 73.0% of the answers inform that the WGB rules are better than the DEV rules or the only appropriate rule in the comparison. Concerning the understanding, 98.3% of the answers report the WGB rules as appropriate, and 61.7% of the answers are that the WGB rules are better than the DEV rules or the only appropriate rule in the comparison. The results of these evaluation are similar and indicate that the WGB rules are appropriate in almost all comparisons (96.5%–100.0%). Furthermore, they are considered better representations than the DEV rules in many comparisons (61.7%–83.5%). Surprisingly, the DEV rules are considered inappropriate in 31.3%, 37.4%, 36.5%, and 22.6% regarding the granularity, accuracy, implementation, and understanding, respectively.

Observing the answers by types of the comparisons and by criteria, we notice that each type of comparison has a most common answer. In generalization comparisons, the accuracy, understanding and implementation have the both rules appropriate as most common answers (30.0%–40.0%), while the granularity most common answer is that the WGB rules are a better representation (40.0%). The most frequent answer to specialization comparisons is that both rules are appropriate but the WGB rule is a better representation of the architecture rule (44.4%–53.3%) independently of the criteria analyzed. In the implicit comparisons, the majority of the answers to accuracy and understanding is that the WGB rules are the only appropriate rules in the comparison (51.7% and 55.0%,

Figure 5.5: WGB and DEV Rule Comparison: Accuracy.



respectively). Concerning the granularity and implementation of implicit comparisons, the most common answer is that both rules are appropriate but the WGB rule is a better representation (47.7% and 36.7%, respectively).

Although the WGB rules are considered appropriate in most of the comparisons, there are eight comparisons with answers reporting the WGB rules as inappropriate or worst than the DEV rules. In C1, P2 and P5 do not agree with the restriction of dependencies represented in the WGB rule where P5 explained that *(DEV rules) presents more details of different possible implementations....* In C6, C7, C12, and C23, some participants prefer the DEV rules and argued that WGB rules had unnecessary details. With the same justification, P2 considers only the WGB rule inappropriate in C10. P2 reports both rules as inappropriate in C13 and C14, which is contradictory to his justifications. His textual answers indicate that the DEV rules is better and both rules are appropriate because he commented that *"Your diagram is way more useful to understand the module, but I do not think that our diagram is wrong..."* to C13 and that *"Again, a more detailed rule helps to understand the service, but the our diagram is only less detailed."*.

Analysis of the comparisons individually. Inspecting the answers to each comparison, we note that there is a variation in the answers to the same comparison. 33.7% of the comparisons has at least three different answers. For instance, C6 has four different answers regarding the granularity and implementation. In the answers to the open-ended questions, these differences are reinforced. In C6, P4 stated that *"(WGB rule) has unnecessary details."*, while P2 explained that *"(WGB rule) is interesting because it presents internal modules of the service..."*. Another example is the C12 where P2 reported that *"In this case, the three dependencies are not relevant..."*, while P4 argued that *"It has relevant details that could help."*. C6 and C12 are comparisons with opposite answers

Figure 5.6: WGB and DEV Rule Comparison: Implementation.

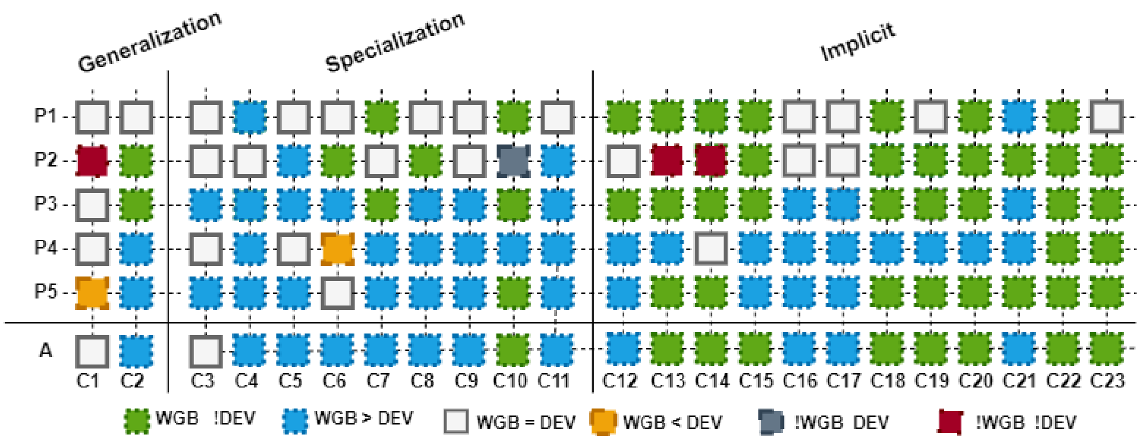
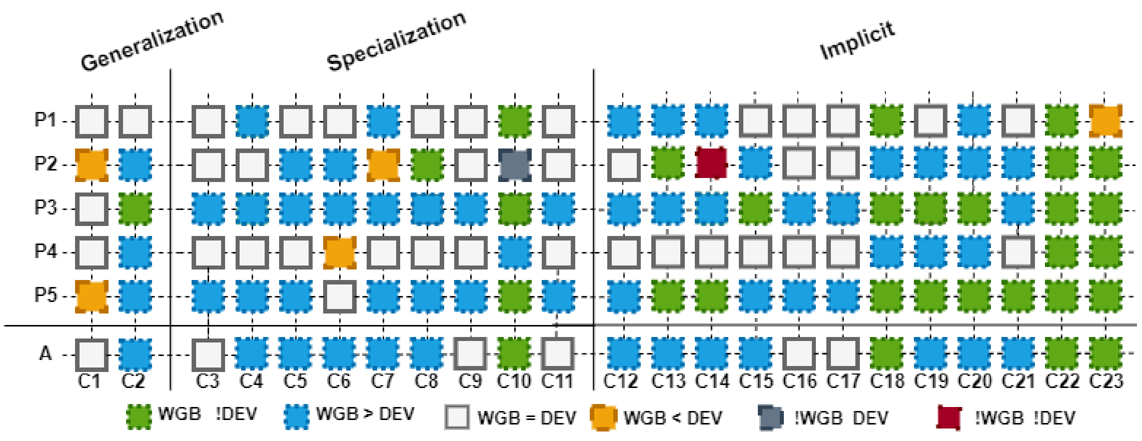


Figure 5.7: WGB and DEV Rule Comparison: Understanding.



about the WGB rules, but the most common differences are less contradictory about the WGB. As already presented, the majority of the answers reported the WGB rules as more appropriate than DEV rules or the only appropriate rule in the comparison. Thus, the divergences in the answers are about whether the DEV rules are appropriate.

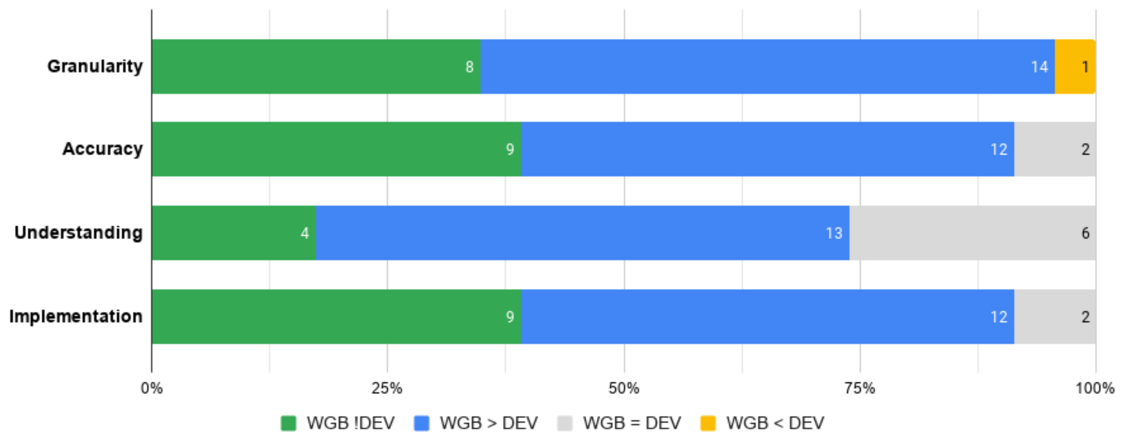
Perspectives of the participants. To understand these differences in the answers of the participants to the same comparison, we also analyze the answers of each participant to all comparisons. We observed that the participants frequently do not agree with each other, as can be noticed by the variation of color in the lines in Figures 5.4–5.7. The pair of participants with the most and the least similar answers are P3 and P5 with 67.4% equal answers, and P3 and P4 with 30.4% equal answers. Moreover, there are the divergences in the justifications, as C6 and C12 previously mentioned. These differences between participants may occur because they have different perspectives on what should be documented in the architecture. Therefore, establishing a unique view of a system architecture that is appropriate for all purposes and people a complex task.

Agreement of the answers of the participants. Given the lack of consensus in the answers to each comparison and the between the participants, we analyze the agreement (A) between the participants on each comparison. In this analysis, the agreement is given by the simple majority, i.e., three or more participants answering the same to a comparison. An example of the agreement by simple majority occurs in the answers to the granularity of C2, in Figure 5.4. In this comparison, three of the five participants answer that the WGB rule is better than the DEV rule. Therefore, the agreement to C2 regarding the granularity is that the WGB rule is more appropriate than the DEV rule. In the comparisons without a simple majority, the agreement is based on the appropriateness of the rules and the preferred rule. The appropriateness is given by the majority of the participants reporting the rule as appropriate, e.g., three participants reporting a rules as appropriate. The preferred rule is the rule with more participants reporting it as better or the only appropriate rule. For instance, the implementation of C6 have both rules as appropriate according to the simple majority because more than two participants report the WGB rule and the DEV rule as appropriate. Analyzing the preferred rule of C6, it has one participant preferring the DEV rule, and two participants preferring the WGB rule. Therefore, the agreement to C6 considering the implementation is that both rules are appropriate, but the WGB rule is best. This agreement is represented in the last line of the C6 in Figure 5.6.

We summarize the agreement results in Figure 5.8. The WGB rules are appropriate in 100.0% of the comparisons, and preferred in 73.9–95.7% of the comparisons analyzing the four criteria. There is one comparison that the DEV rule is preferred, which is the generalization comparison already discussed. These results reinforce that the participants prefer the WGB rules. They also suggest that the architecture rules should be more detailed because the participants prefer rules that are finer-grained instead of commonly adopted coarse-grained rules. This preference for fine-grained occurs even in the comparison where the DEV rule is better than the WGB rule.

In the answers to the comparisons, the WGB rules are considered appropriate in 96.5%–100.0%, better than the DEV rules in 61.7%–80.0%, and the only appropriate rule in 22.6%–36.5%. Analyzing these results, we notice a divergence in the answers to each comparison and between participants, which may occur because the developers have different perspectives of the architectures of the systems.

Figure 5.8: WGB and DEV Rule Comparison: Agreement.



Given these divergences, we consolidate the answers analyzing the agreement to each comparison. These results present the WGB rules as appropriate in all comparisons, and preferred in 73.9–95.7% of the comparisons concerning the four analyzed criteria. Thus, our method recovers appropriate architecture rules and, in most of the cases, even better rules than the manually recovered rules. Furthermore, the results indicate a preference for more detailed architecture rules than the commonly adopted high-level rules.

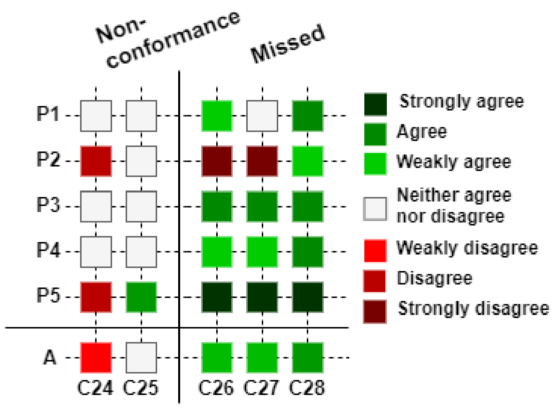
5.3.2.3 RQ2: Revealing Misunderstandings and Violations

In the previous section, we analyzed the comparisons of the WGB rules and the DEV rules. However, in both documentations, some rules do not have a corresponding representation in the other documentation. In this section, we analyze the comparisons associated with RQ2, aiming to investigate the rules related to non-conformance and missed comparisons.

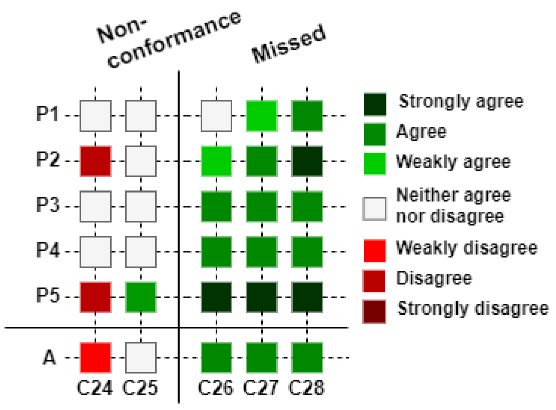
Nature of the misunderstandings. In Figure 5.9, we present the answers to the two non-conformance and three missed comparisons about the granularity, accuracy, understanding and implementation. We analyze these answers by their type considering that non-conformance comparisons are related to DEV rules, and missed comparisons are related to the WGB rules.

In C24 and C25, which are non-conformance comparisons, the participants are mostly neutral in their answers—70.0% neither agree nor disagree to all criteria. In the open-ended question of C24, P1 and P3 explained that they were not sure about these

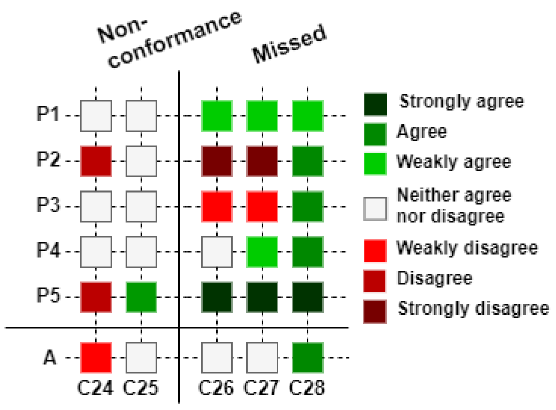
Figure 5.9: Non-conformance and Missed Comparisons.



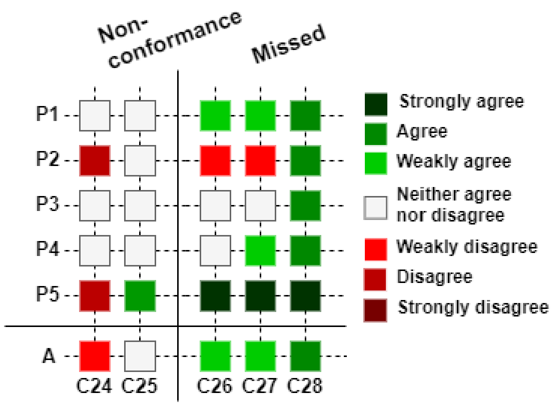
(a) Granularity



(b) Accuracy



(c) Understanding

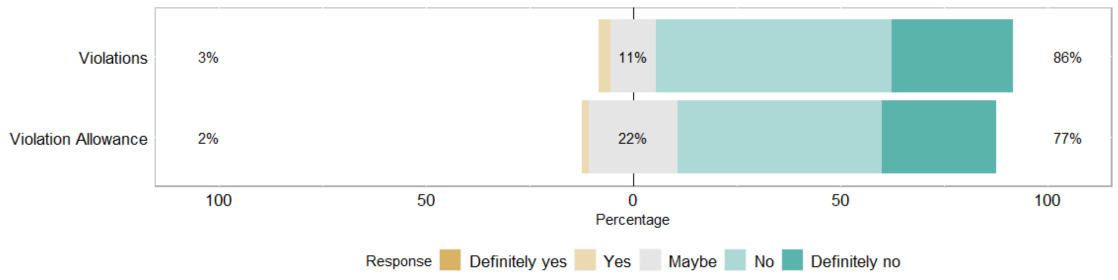


(d) Implementation

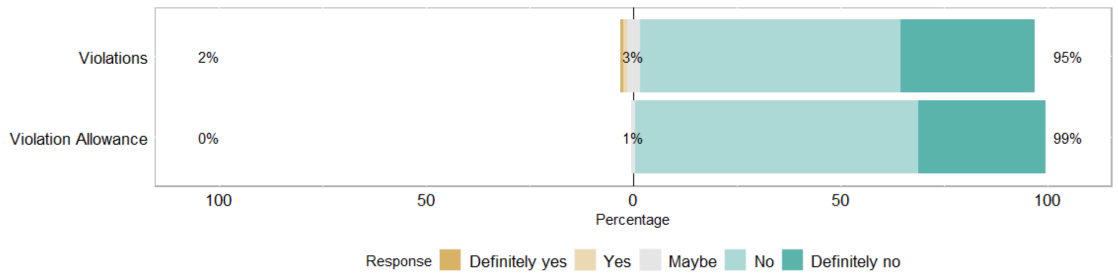
rules, P4 reported that *"we may misrepresent the dependencies..."*, and P5 recognized a problem and reported that *"We made a mistake..."*. In C25, P1–3 explained that they did not remember the reason to document this rule.

Differently from the non-conformance comparisons, in the missed comparisons (C26–C28), the participants vary more their answers to each criteria. 80.0%, 93.3%, 66.7%, and 66.7% of the answers to granularity, accuracy, understanding and implementation, respectively, at least weakly agree that the WGB rules are adequate. The understanding of C26 and C27 have divergences in the answers because they are related to the integration tests of the System B. This nature of the rule led P2 to disagree about the correctness of the granularity, understanding, and implementation. P2 argued that *"This module contains the integration tests and should not be documented..."* about C26 and C27. P3 do not agree to understanding and explained that *"I do not think that the it tests should be documented in the architectural level."*. In C28, P3 and P4 pointed out that *"this is an important part of the system..."*. Furthermore, P2 identify a mistake in the

Figure 5.10: WGB and DEV Rules: Violations and Violations Allowance.



(a) DEV rules.



(b) WGB rules.

DEV rule and reported that *"this module (presented in the WGB rule) should be documented."* in the justification of C28. These results present that the WGB rules support the identification of problems and reveal important parts of the system that were forgotten during the manual architecture recovery of the systems. There are also divergences in the answers due to the nature of the rules.

Given the divergences in the answers, we investigate the agreement to these comparisons. The agreement is given by the average of answers according to the Likert scale, where the answers have a variation of one and ranges from 3 (strongly agree) to -3 (strongly disagree). In the non-conformance comparisons, C24 is not adequate because it is a problem according to two participants, and C25 is neutral. In the missed comparisons, in most cases, the WGB rules are adequate. Only C26 and C27 in the understanding criteria are neutral because of the divergences already mentioned.

Presence and reasoning of violation. Additionally to the analysis of mistakes, we analyze the architectural violations that are explicitly documented (violation) and the violations that are not explicitly documented but may exist due to an overly coarse-grained representation of the rules (violation allowance). Therefore, we present the answers to questions of both sets of rules in Figure 5.10. We split our results presenting the answers to the DEV rules, in Figure 5.10a, and to the WGB rules in Figure 5.10b.

The answers to violations in the DEV rules show 3% of them presenting violations and 11% of them maybe presenting violations. Specifically, the DEV rules in C24 have violations according to P2 and P4. C24 is related to a missed rule that is a mistake in the documentation, as previously mentioned. We have four comparisons that the DEV rules may present violations: C2 is a generalization comparison that we did not find any specific reason to P3 report it as a violation; C7 is a specification comparison which P2 possibly noticed a violation comparing the DEV rule against the WGB rule because he answered that the WGB rule do not present violations; C24 is a missed comparison which P1 and P3 do not have information about as already discussed; and C25 is also a missed comparison that P1–3 report not having the knowledge to answer.

Analyzing the violation allowance in the DEV rules, there are more participants answering maybe in the allowance of violations than in violations. The comparisons that allow or maybe allow violations are the same four comparisons of violations (C2, C7, C24, and C25), and three more (C3, C4, and C11). C3 is a specialization comparison that P4 answer maybe to violation allowance but answered no to the presence of violations. This answer is possibly due to the lack of details in the DEV rule because he reported that *"It might help to declare the dependency relationship better"*. C4 is also a specialization comparison that P4 answer maybe to the violation allowance but no to the presence of violations. He reported that *"the WGB representation has useful details that actually could help understand the system"*. C11 is a specialization comparison that P1-4 answer maybe to the DEV rule and no to the WGB rule because of the fine-grained information provided by the WGB rule. The detailed information of the WGB rule made them sure about the dependencies related to these rules as P2 reported *"The developer diagram may represent violations given that a module X should not interact with components inside DAO."*.

Analyzing the answers to the violations in the WGB rules presented in Figure 5.10b, there are few rules presenting or allowing violations. The two answers related to the violations in the WGB rules are to C22 and C23 given by P2. He reported that *"It is clear the packaging problem..."* and *"the WGB diagram clarifies that this part of the system has much more problems and that this part of the system is really spaghetti."* to C22 and C23, respectively. Additionally, there are four rules that participants answer maybe to the presence of violations. One of them is C11 that is a comparison with a fine-grained representation but still may have violations, according to P2. C21 also may have violations and is a fine-grained rules that P2 reported as *"possibly a mistake in the modeling of the system..."*. C26 and C27 are reported by P1 as maybe due to their doubts about the importance of the integration test module to be documented.

According to the answers about the violations and allowing violations in the WGB and DEV rules, both sets of rules present few violations. Based on the previously discussed results, we expected these answers about violations in the WGB rules because they are reported as appropriate in RQ1. Thus, it is coherent that those same rules do not present violations. In DEV rules, we expected more violations because 22.6%–36.5% of them were reported as inappropriate in RQ1. These results indicate that the presence of violations is not the main reason that make participants reporting rules as inappropriate. In the results of RQ1, the implicit rules are the type of comparisons with most of the answers reporting the DEV rules as inappropriate. Thus, these results suggest that the rules are reported as inappropriate because of the lack of the details.

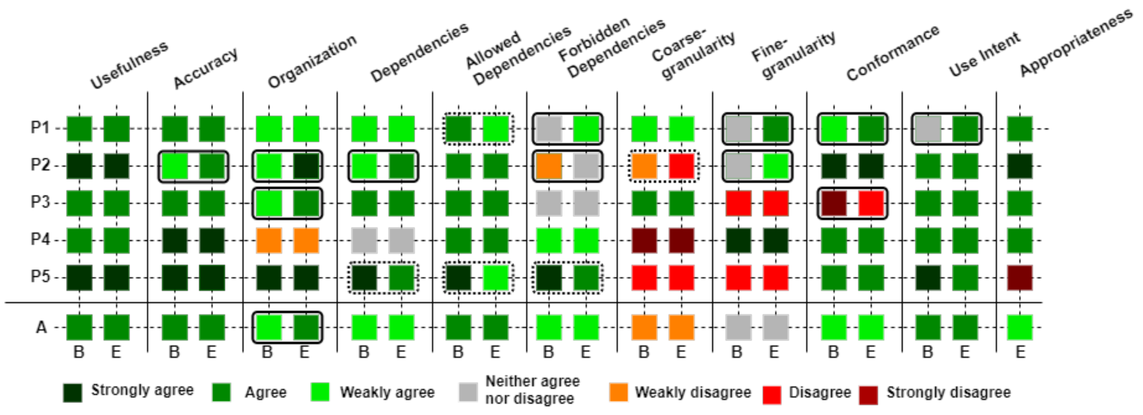
Analyzing the WGB rules, the participants identified violations and problems in the systems. These problems were related to the organization of the systems or the nature of the recovered rules. Moreover, the analysis of violations suggests the main reason to consider a rule as inappropriate is the level of abstraction and not the presence of violations.

5.3.2.4 RQ3: Usefulness of the WGB Method

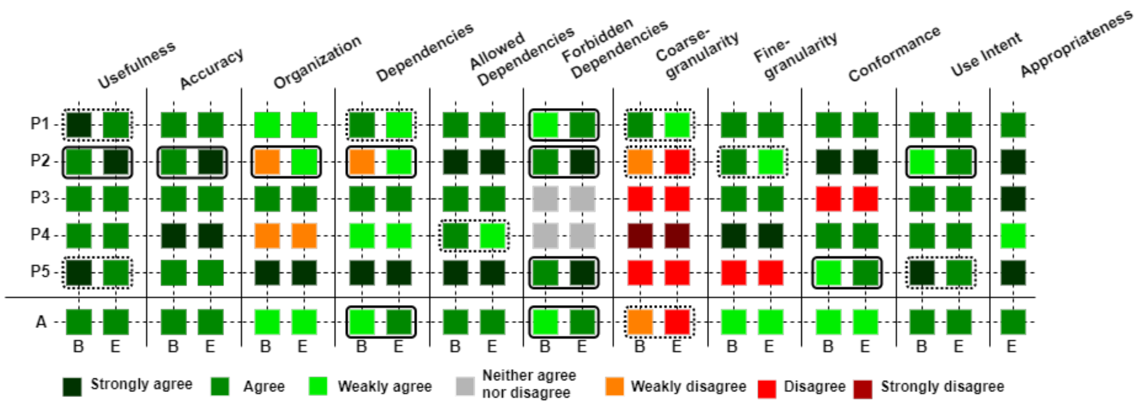
We investigated the rules individually or compared the WGB rules and the DEV rules in the previous analysis. To evaluate the entire documentation recovered built based on the WGB rules, we analyze the opinion of the participants about it to answer RQ3. Therefore, we collect the eleven criteria associated with RQ3 based on the answers to the first and third parts of the questionnaires. These answers are presented in Figure 5.11, where the 20 first columns are associated with the ten questions asked at the beginning (B) and the end (E) of the questionnaires, and the last column is the appropriateness that was asked only at the end. To facilitate the visualization of the answers that changed after the inspection of the WGB rules, we highlighted these changes. A box with a solid border means that a participant agrees more in the last answer. A box with a dotted border means that a participant agrees less in the last answer.

Usefulness of the documentation extracted using the WGB method. To evaluate the usefulness of the documentation built based on the WGB rules, we depict the criteria based on the overall usefulness, dependencies, granularity and practical usage of the documentation.

Figure 5.11: Evaluation of the Documentation Built Based on the WGB Rules Answers.



(a) System A.



(b) System B.

The information of the usefulness, accuracy, organization, and conformance, which are criteria related to the overall usefulness of the documentation, have most of the answers agreeing in both systems. These results indicate that the documentation built based on the WGB rules provides useful information. However, there are two participants disagreeing on these criteria. P2, in the first question to System B, and P4, in all answers, weakly disagrees on the organization because of the overly fine-grained representation in some parts of the documentation.

Concerning the dependencies, allowed dependencies, and forbidden dependencies, the results also show most of the participants agreeing. Although most of the participants agree to these criteria, there are few participants not agreeing to these criteria. These criteria are directly related to the conformance of the implementation with the architecture. Given that the WGB method extracts the rules from the source code, the participants agree that the WGB method extracted the architecture rules properly with these answers. The forbidden dependencies criteria depends on the violations of the systems. In the results of RQ2, the participants identified few violations and problems in the architecture, which may make them agree with the presence of forbidden dependencies in the documentation.

To evaluate the level of abstraction of the documentation, we evaluate the granularity of the documentation based on the answers to fine-granularity and coarse-granularity criteria. Analyzing these criteria, the documentations are not considered coarse-grained to both systems. Moreover, the documentation of System B is considered fine-grained. As in the comparisons of the WGB and DEV rules previously discussed, these answers indicate that the documentations of System A and B are more fine-grained than coarse-grained.

Independently of the level of abstraction, we evaluate whether the participants would use the documentation built based on WGB rules based on the use intent and appropriateness criteria. All participants agree to both criteria excepting P1 and P5. P1 is neutral about the use intent before inspecting the rules of System A. P5 strongly disagrees on the appropriateness of System B. The disagreement of P5 may be due to a mistake because his answer is the opposite of what he reported in the open-ended question. He reported that *"The architecture of WGB is richer than the architecture made by us."*. These results corroborate with the individual rule evaluation in which the WGB rules were considered appropriate and can be used as architecture documentation of the systems despite of their level of abstraction.

In their textual answer about the documentation, the comments of the participants are related to the usefulness of the information provided and the context that the documentation should be adopted. P3 reported that *"The diagram is very detailed. I think that it helped in finding some problems, but they would be detected with a more high-level diagram and the few detailed diagrams."*. Moreover, P2 reported that *"The documentation has too much details... The dependencies of the internal components could be represented only in the detailed diagrams."*. P1 and P4 consider the documentation proper for developers and reported that *"For developers, the documentation provides fundamental information... and facilitates the conformance."* and *"It depends to whom it would be presented... However to an experienced developer or someone with a deep understanding of the product, a more detailed representation could be helpful indeed."*, respectively.

In most of the evaluations of the documentation, there are no contradictory answers, i.e., the evaluations have only agreeing or neutral answers. However, 43.2% of the evaluations have contradictory answers. For instance, the organization of System A has one participant disagreeing and the others agreeing. To obtain a consolidated answer to these 19 evaluations, we analyze the agreement (A) of these evaluations. In organization, dependencies, forbidden dependencies, conformance, and appropriateness of both systems, the majority of the participants agrees and only one or two participants disagree. Thus, these agreements are at least weakly agree. The opposite occurs in the coarse-granularity, where P1 and P3 agree and the others disagree. Therefore, the agreement of the coarse-granularity is weakly disagree or disagree. In fine-granularity in System A, the agreement is neutral and, in System B, is weakly agree. Considering all the evaluations, the consolidated results are that the participants agree with the statements of the criteria. Only the evaluations related to the granularity that have neutral and disagreeing consolidated answers. Considering the agreement of fine-granularity and coarse-granularity criteria, there is a difference in the answers according to the systems. The difference in the granularity of System A and B may be due to their size and architectural differences. Given that System B is larger than System A, it is expected that System B has more modules and rules than System A.

Changes in the answers after the inspection of the rules. In the previous analysis, we evaluated the documentation based on the answers to the first part and the third part of the questionnaire not considering the effect of inspecting the WGB rules in the second part. To evaluate this effect on the answers, we compare the answers from the first and the third part of the questionnaire.

The inspection of individual rules affected perception of the participants in 32% of the answers. Only the usefulness of the documentation built using the WGB method of System A has no changes. In all the other criteria, at least one participant changed his opinion. P2 is the participant that most changed his opinion—changed his opinion in 70% of the evaluations. The inspection of individual rules made some participants agree more to questions. The accuracy improved in both systems due to P2 changes. The organization had four opinions changed and the conformance had three changes. These criteria are related to information about the systems suggesting that the inspection of the architecture rules improved the knowledge of the participants about the systems.

Another two criteria had three changes to agree less after the participants inspecting the rules: allowed dependencies and coarse-granularity criteria. These criteria are related to the information and the abstraction of the architecture rules. Based on the individual rule analysis, these changes are expected because most of the justifications reported that the WGB rules are appropriate and have a detailed representation of the architecture.

The documentation built based on the WGB rules is adequate concerning the quality and practical usage criteria according to the answers of the participants. Although it is considered fine-grained, the participants prefer the documentation built based on the WGB rules instead of high-level documentation commonly adopted. Therefore, the documentation provided by the WGB method is more detailed but also more useful. Additionally, the inspection of the rules individually had little influence on the answers of the participants about the documentation.

5.3.3 Threats to Validity

In this section, we identify threats to validity in our evaluation and detail the actions adopted to mitigate them.

A threat to construct validity identified is that some steps of our evaluation require the involvement and knowledge of the developers about software architecture and the subject systems. The lack of knowledge and experience could affect the recovered architectures. It is important to note that we asked the developers that work in the systems daily, and, consequently, they are the most expert developers in these systems to participate in this experiment. Moreover, we noticed in the demographic information that all of them have at least one year of working on the systems and have formal degrees in information systems. Additionally, we provided a brief tutorial on software architecture that ensured that the concepts adopted in this study were presented to all participants. We also were available during the recovery of software architectures to answer any doubts they may have. Another threat to construct validity is that our evaluation relied on five developers to answer questions about the comparisons of documentations of the subject systems. Thus, our results might be affected by some degree of subjectivity. We analyzed the agreement between the developers to provide a more consensual perspective between the participants mitigating the subjectivity in the answers.

An internal validity threat of our study is related to the involvement of the participants in time-consuming activities. To reduce the tiredness and boredom impact in these activities, we did not set a time-limit to finish any of the tasks. Moreover, they were free to take breaks whenever they want. Another threat to internal validity is related to the diagrams provided to the participants to analyze the rules, which may difficult the activity.

The diagrams were used to facilitate the understanding with different color, presenting the modules and rules in a better organization possible. Furthermore, we provided the documentation to the participants to let them inspect the documentation freely.

An external threat, which refers to the selected subject systems, is related to the generalization of the results. Given that the selected systems were developed by the same company, i.e. both had the same domain, similar development methods, almost the same developers, and are not large systems, there is few variability on these aspects of the subject systems. In this evaluation, we focused on the quality of the extracted rules to provide complementary investigation of the results presented in Section 5.2, which was a trade-off in this evaluation.

5.4 Final Remarks

In this chapter, we evaluated the WGB method according to its efficiency, effectiveness, and usefulness. Considering the efficiency, our method takes 45.5s to execute with our largest subject system. With respect to effectiveness, our method achieves a reduction from module dependencies to recovered rules of 87.6%, on average, thus reducing most of the developers' effort. The comparison of recovered implemented rules with conceptual rules shows that they largely differ, leading to 37.1% and 37.8% of precision and recall, respectively, on average. A qualitative analysis of recovered rules shows that our method: (i) generalizes many rules associated with sub-modules of a module as a single super-rule; (ii) is able to capture rules that occur specifically between sub-modules, being often sub-rules of conceptual rules; (iii) identifies rules that govern dependencies within a module, which are typically not specified as conceptual rules; and (iv) leaves architectural violations as fine-grained rules, so that it is easier to distinguish them from other recovered rules using our dependency strength metric. The evaluation of usefulness reinforces that it extracts fine-grained rules. However, based on the opinion of the developers, the WGB method extracts rules that are more appropriate than the rules recovered by them in most of the comparisons and could be used as documentation. Additionally, the user study revealed the disagreement between developers about the proper granularity of the rules, which according to our results, should be fine-grained to be more useful during the development, maintenance, and evolution of the system.

6 CONCLUSION

The lack of understanding of architecture rules that are implemented in the code is a key barrier to a healthy software development and evolution, leading to many maintainability problems. This thesis addressed the problems related to the recovery of architecture rules aiming to mitigate this lack of understanding. Then, in this chapter, we summarize the contributions of this thesis in Section 6.1 and present future work in Section 6.2.

6.1 Contributions

As the result of this thesis, a number of contributions can be enumerated. These contributions present the reasoning, the development and the evaluation of the Weighted-graph-based method to recover architecture rules and are listed below.

Architecture Recovery Review. We provide a review of the research related to architecture recovery presenting them based on their input, output, analysis, and purpose (Chapter 2).

Empirical Analysis of Architecture Conformance. We presented an exploratory study that assesses and investigates the gap between conceptual architecture rules and dependencies among modules implemented in the source code (ZAPALOWSKI; NUNES; NUNES, 2018). This study shows how complex the task of keeping the architecture documentation in conformance with the source code is because the abstraction commonly used to document architectures is not easily mapped to the source code. (Chapter 3).

WGB Method to Recover Architecture Rules. We proposed the *Weighted-graph-based* (WGB) method to recover architecture rules (ZAPALOWSKI; NUNES; NUNES, 2018), which is automatic and domain-independent. Our method does not require the specification of any threshold, or system-specific customizations. Our method includes the calculation of a proposed metric, module dependency strength (MDS), between pairs of modules and the resolution of an optimization problem. MDS not only takes into account dependencies within a module, but also its context, i.e. its surrounding modules. Furthermore, our method relies on a optimization problem proposed by us to choose the best set of rules using the results of MDS (Chapter 4).

Offline Evaluation. We evaluated the performance and effectiveness of the WGB method (ZAPALOWSKI; NUNES; NUNES, 2018). The results show that our method produces results in a timely fashion. Considering the effectiveness, our method reduces the gap between module dependencies and recovered rules in 87.6%, on average. Furthermore, the comparison of recovered implemented rules with conceptual rules shows that they differ due to the inevitable gap between an intended architecture and an implemented architecture. The qualitative analysis of recovered rules presented many specialization and intra-module rules, which allowed rules but more specific than commonly adopted conceptual rules. It also shows the architectural violations as fine-grained rules facilitating the identification of them from other recovered rules using our dependency strength metric (Chapter 5).

User Evaluation. We evaluated our method from the perspective of developers to understand the usefulness of the rules extracted using the WGB method. We analyzed the architectures of two commercial systems comparing the architecture manually recovered by developers and the architecture built based on the rules extracted using the WGB method. This study indicates that the WGB method extracts architecture rules that are more appropriate than the architecture rules recovered by the developers of the systems in most of the cases. The results also indicate that the developers would use the WGB rules as architecture documentation. Furthermore, these results reinforce that the architecture rules should be finer-grained because of the preference of the developers for the WGB rules mainly concerning more detailed rules—specialization and implicit comparisons (Chapter 5.3).

6.2 Future Work

In addition to the discussions presented in the previous chapters, we present in this section further insights that emerged from our proposal and its evaluations.

Undocumented Rules *vs.* Architectural Violations. Our offline study, presented in Chapter 5, resulted in many recovered implemented rules that are inconsistent with the conceptual rules. Rules in this category occur due to two causes: (i) an undocumented rule, which should be added as a conceptual rule; or (ii) an architectural violation, which should be fixed in the code. Based on our qualitative analysis of recovered rules, we identified many undocumented rules. Architectural violations, in turn, should not be abstracted as a general rule, so that they can be identified as a violation. Our method addresses this assuming there are few occurrences of the same violation type, causing our metric to have a low value. Thus, the MDS metric (i.e. the weight) associated with the rules recovered using our method can serve as an indicator to distinguish undocumented rules from architectural violations. Further work should be done to do this automatically.

Alternative Architecture Views. As a result of the pairwise clusterization of dependencies of the WGB method, presented in Chapter 4, there are many arrows in the graph that are candidates to be recovered implemented rules. However, a subset of arrows should be selected to provide rules that are not redundant, e.g. a rule is a sub-rule of another. We denote redundant rules as conflicting, and they both cannot be in the solution that gives recovered rules.

Our method considers as an optimal solution that with the highest dependency strengths. Nevertheless, there are many alternative solutions with non-conflicting rules. Such alternative solutions consist of possible alternative recovered implemented rules, providing different representations of the same architecture. Furthermore, the idea of having alternative views of a system architecture is reinforced by the results of our user study that presented many divergences between the participants' answers. The participants have different opinions about the rules and modules that should be presented in the architecture. This can be potentially further explored to investigate the suitability of these alternative sets of rules to represent implemented rules.

Tool Support. We developed a tool (SCHMITZ et al., 2017) that provided support to the investigation presented in Chapter 3. However, this tool does not have the WGB method implemented. In our evaluations of the WGB method, we used the IBM ILOG CPLEX Optimization Studio as the optimization problem solver, which is one of the best optimization problem solvers available. However, it is a multipurpose solver, which is large and complex to use, and cannot be distributed as part of the tool due to legal rights. To integrate the WGB method into a tool, this solver cannot be used. Thus, the implementation of the WGB method in a tool using a different optimization problem solver would promote the adoption of the WGB method and facilitate the recovery of architecture rules.

In summary, this thesis advances research on recovery of the software architecture. Clearly there is still much to be done to have a method that fully automate the process of architecture recovery, but our work consists of a relevant step towards the reduction of the effort needed to have reliable and up-to-date architecture documentation.

REFERENCES

ALLEN, E. B.; KHOSHGOFTAAR, T. M.; CHEN, Y. Measuring coupling and cohesion of software modules: An information-theory approach. In: INTERNATIONAL SOFTWARE METRICS SYMPOSIUM. **Proceedings...** 2001. p. 124–134. ISBN 0-7695-1043-4. Available from Internet: <http://doi.org/10.1109/METRIC.2001.915521>.

BASS, L.; CLEMENTS, P.; KAZMAN, R. Architectural tatics and patterns. In: ____. **Software Architecture in Practice**. Pearson Education, 2012. (SEI Series in Software Engineering). ISBN 9780132942782. Available from Internet: <http://books.google.com.br/books?id=-II73rBDXCYC>.

BASS, L.; CLEMENTS, P.; KAZMAN, R. What is a software architecture? In: ____. **Software Architecture in Practice**. Pearson Education, 2012. (SEI Series in Software Engineering). ISBN 9780132942782. Available from Internet: <http://books.google.com.br/books?id=-II73rBDXCYC>.

BELLE, A. B.; BOUSSAIDI, G. E.; KPODJEDO, S. Combining lexical and structural information to reconstruct software layers. **Information and Software Technology**, v. 74, p. 1–16, 2016. Available from Internet: <https://doi.org/10.1016/j.infsof.2016.01.008>.

BISCHOFBERGER, W.; KÜHL, J.; LÖFFLER, S. Sotograph – a pragmatic approach to source code architecture conformance checking. In: EUROPEAN WORKSHOP SOFTWARE ARCHITECTURE. **Proceedings...** 2004. p. 1–9. ISBN 978-3-540-24769-2. Available from Internet: <https://doi.org/10.1007/978-3-540-24769-2_1>.

BITTENCOURT, R. A. et al. Improving Automated Mapping in Reflexion Models Using Information Retrieval Techniques. In: WORKING CONFERENCE ON REVERSE ENGINEERING. **Proceedings...** IEEE, 2010. p. 163–172. ISBN 978-1-4244-8911-4. Available from Internet: <http://doi.org/10.1109/WCRE.2010.26>.

BOURQUIN, F.; KELLER, R. High-impact refactoring based on architecture violations. In: EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING. **Proceedings...** 2007. p. 149–158. Available from Internet: <http://doi.org/10.1109/CSMR.2007.25>.

BROWN, W. J. et al. **AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis**. New York, NY, USA: John Wiley & Sons, Inc., 1998. ISBN 0-471-19713-0.

BRUNET, J. a. et al. On the Evolutionary Nature of Architectural Violations. In: WORKING CONFERENCE ON REVERSE ENGINEERING. **Proceedings...** 2012. p. 257–266. Available from Internet: <https://doi.org./10.1109/WCRE.2012.35>.

CHRISTL, a.; KOSCHKE, R.; STOREY, M.-a. Equipping the reflexion method with automated clustering. In: WORKING CONFERENCE ON REVERSE ENGINEERING. **Proceedings...** IEEE, 2005. p. 89–98. ISBN 0-7695-2474-5. Available from Internet: <http://doi.org/10.1109/WCRE.2005.17>.

CHRISTL, A.; KOSCHKE, R.; STOREY, M.-A. Automated clustering to support the reflexion method. **Information and Software Technology**, v. 49, n. 3, p. 255–274, 2007. ISSN 0950-5849. Available from Internet: <https://doi.org/10.1016/j.infsof.2006.10.015>.

CONSTANTINOU, E.; KAKARONTZAS, G.; STAMELOS, I. Towards Open Source Software System Architecture Recovery Using Design Metrics. In: PANHELLENIC CONFERENCE ON INFORMATICS. **Proceedings...** 2011. p. 166–170. Available from Internet: <http://doi.org/10.1109/PCI.2011.36>.

CORAZZA, A. et al. Investigating the Use of Lexical Information for Software System Clustering. In: EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING. **Proceedings...** 2011. p. 35–44. ISBN 978-1-61284-259-2. ISSN 15345351. Available from Internet: <https://doi.org/10.1109/CSMR.2011.8>.

DUCASSE, S.; POLLET, D. Software architecture reconstruction: A process-oriented taxonomy. **IEEE Transactions on Software Engineering**, v. 35, n. 4, p. 573–591, 2009. ISSN 00985589. Available from Internet: <https://doi.org/10.1109/TSE.2009.19>.

GARCIA, J.; IVKOVIC, I.; MEDVIDOVIC, N. A comparative analysis of software architecture recovery techniques. In: INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING. **Proceedings...** 2013. p. 486–496. Available from Internet: <https://doi.org/10.1109/ASE.2013.6693106>.

GARCIA, J. et al. Obtaining ground-truth software architectures. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING. **Proceedings...** 2013. p. 901–910. ISSN 0270-5257. Available from Internet: <https://doi.org/10.1109/ICSE.2013.6606639>.

GARCIA, J. et al. Toward a catalogue of architectural bad smells. In: **Architectures for Adaptive Software Systems**. Springer Berlin Heidelberg, 2009, (Lecture Notes in Computer Science, v. 5581). p. 146–162. ISBN 978-3-642-02351-4. Available from Internet: <https://doi.org/10.1007/978-3-642-02351-4_10>.

GARCIA, J. et al. Enhancing architectural recovery using concerns. In: INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING. **Proceedings...** 2011. p. 552–555. ISBN 978-1-4577-1638-6. Available from Internet: <http://doi.org/10.1109/ASE.2011.6100123>.

HENNEY, K.; SCHMIDT, D.; BUSCHMANN, F. **Pattern Oriented Software Architecture: On Patterns and Pattern Languages**. John Wiley & Sons, 2007. (Wiley Series in Software Design Patterns). ISBN 9780471486480. Available from Internet: <http://books.google.com.au/books?id=wzplRf3uh-EC>.

HORA, A. et al. Mining system specific rules from change patterns. In: WORKING CONFERENCE ON REVERSE ENGINEERING. **Proceedings...** IEEE, 2013. p. 331–340. ISBN 978-1-4799-2931-3. Available from Internet: <https://doi.org/10.1109/WCRE.2013.6671308>.

HUYNH, S. et al. Automatic modularity conformance checking. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING. **Proceedings...** 2008. p. 411–420. Available from Internet: <https://doi.org/10.1145/1368088.1368144>.

JANSEN, A.; BOSCH, J. Software architecture as a set of architectural design decisions. In: WORKING IEEE/IFIP CONFERENCE ON SOFTWARE ARCHITECTURE. **Proceedings...** 2005. p. 109–120. Available from Internet: <http://doi.org/10.1109/WICSA.2005.61>.

KNODEL, J.; POPESCU, D. A comparison of static architecture compliance checking approaches. In: WORKING IEEE/IFIP CONFERENCE ON SOFTWARE ARCHITECTURE. **Proceedings...** 2007. p. 12–22. Available from Internet: <http://doi.org/10.1109/WICSA.2007.1>.

Kong, X. et al. Directory-based dependency processing for software architecture recovery. **IEEE Access**, v. 6, p. 52321–52335, 2018. Available from Internet: <http://doi.org//10.1109/ACCESS.2018.2870118>.

LI, Z.; ZHOU, Y. Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code. **ACM SIGSOFT Software Engineering Notes**, v. 30, n. 5, p. 306–315, sep. 2005. ISSN 0163-5948. Available from Internet: <https://doi.org/10.1145/1081706.1081755>.

LUTELLIER, T. et al. Comparing Software Architecture Recovery Techniques Using Accurate Dependencies. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING. **Proceedings..** 2015. v. 2, p. 69–78. ISBN 978-1-4799-1934-5. ISSN 02705257. Available from Internet: <https://doi.org/10.1109/ICSE.2015.136>.

MACIA, I. et al. On the relevance of code anomalies for identifying architecture degradation symptoms. In: EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING. **Proceedings...** 2012. p. 277–286. Available from Internet: <http://doi.org/10.1109/CSMR.2012.35>.

MACIA, I. et al. Are automatically-detected code anomalies relevant to architectural modularity? In: INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT. **Proceedings...** ACM Press, 2012. p. 167–178. ISBN 9781450310925. Available from Internet: <https://doi.org/10.1145/2162049.2162069>.

MAFFORT, C. et al. Heuristics for discovering architectural violations. In: WORKING CONFERENCE ON REVERSE ENGINEERING. **Proceedings...** IEEE, 2013. p. 222–231. ISBN 978-1-4799-2931-3. Available from Internet: <https://doi.org/10.1109/WCRE.2013.6671297>.

MAFFORT, C. et al. Mining architectural patterns using association rules. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING. **Proceedings...** 2013. p. 375–380. Available from Internet: <http://www.dcc.ufmg.br/~mtov/pub/2013_seke_archlint.pdf>.

MAFFORT, C. et al. Mining architectural violations from version history. **Empirical Software Engineering**, jan. 2015. ISSN 1382-3256. Available from Internet: <http://doi.org/10.1007/s10664-014-9348-2>.

MANCORIDIS, S. et al. Using automatic clustering to produce high-level system organizations of source code. In: INTERNATIONAL WORKSHOP ON PROGRAM COMPREHENSION. **Proceedings...** 1998. p. 45–52. ISSN 1092-8138. Available from Internet: <https://doi.org/10.1109/WPC.1998.693283>.

MEDVIDOVIC, N.; TAYLOR, R. N. Software architecture: Foundations, theory, and practice. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING. **Proceedings...** 2010. p. 471–472. ISBN 978-1-60558-719-6. Available from Internet: <http://doi.org/10.1145/1810295.1810435>.

MO, R. et al. Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells. In: WORKING IEEE/IFIP CONFERENCE ON SOFTWARE ARCHITECTURE. **Proceedings...** IEEE, 2015. p. 51–60. ISBN 978-1-4799-1922-2. Available from Internet: <https://doi.org/10.1109/WICSA.2015.12>.

MO, R. et al. Architecture anti-patterns: Automatically detectable violations of design principles. **IEEE Transactions on Software Engineering**, p. 1–21, 2019. Available from Internet: <https://doi.org/10.1109/TSE.2019.2910856>.

MURPHY, G.; NOTKIN, D.; SULLIVAN, K. Software reflexion models: bridging the gap between design and implementation. **IEEE Transactions on Software Engineering**, v. 27, n. 4, p. 364–380, apr 2001. ISSN 00985589. Available from Internet: <https://doi.org/10.1109/32.917525>.

MURPHY, G. C.; NOTKIN, D.; SULLIVAN, K. Software reflexion models. **ACM SIGSOFT Software Engineering Notes**, ACM, New York, NY, USA, v. 20, n. 4, p. 18–28, oct. 1995. ISSN 01635948. Available from Internet: <http://doi.org/10.1145/222132.222136>.

NISTOR, E. C. et al. Archevol: Versioning architectural-implementation relationships. In: INTERNATIONAL WORKSHOP ON SOFTWARE CONFIGURATION MANAGEMENT. **Proceedings...** 2005. p. 99–111. ISBN 1-59593-310-7. Available from Internet: <https://doi.org/10.1145/1109128.1109136>.

OFFUTT, A. J.; HARROLD, M. J.; KOLTE, P. A software metric system for module coupling. **Journal of Systems and Software**, v. 20, n. 3, p. 295–308, mar. 1993. ISSN 0164-1212. Available from Internet: <https://doi.org/10.1016/0164-1212(93)90072-6>.

PAIVA, R. et al. Exploring the combination of software visualization and data clustering in the software architecture recovery process. In: ACM SYMPOSIUM ON APPLIED COMPUTING. **Proceedings...** 2016. p. 1309–1314. ISBN 9781450337397. Available from Internet: <https://doi.org/10.1145/2851613.2851765>.

PARNAS, D. Software aging. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING. **Proceedings...** 1994. p. 279–287. ISSN 0270-5257. Available from Internet: <https://doi.org/10.1109/ICSE.1994.296790>.

PERRY, D. E.; WOLF, A. L. Foundations for the study of software architecture. **SIGSOFT Software Engineering Notes**, v. 17, n. 4, p. 40–52, oct. 1992. ISSN 0163-5948. Available from Internet: <https://doi.org/10.1145/141874.141884>.

SANGAL, N. et al. Using dependency models to manage complex software architecture. In: ACM SIGPLAN CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS. **Proceedings...** 2005. p. 167. ISBN 1595930310. Available from Internet: <http://doi.org/10.1145/1094811.1094824>.

SARKAR, S. et al. Modularization of a large-scale business application: A case study. **IEEE Software**, v. 26, n. 2, p. 28–35, March 2009. ISSN 0740-7459. Available from Internet: <https://doi.org/10.1109/MS.2009.42>.

SCHMITZ, C. et al. Extracting implemented module dependencies with the arr tool. In: CONGRESSO BRASILEIRO DE SOFTWARE: TEORIA E PRáTICA - SESSãO DE FERRAMENTAS. **Proceedings...** 2017. p. 89–96. Available from Internet: <https://www.lia.ufc.br/~cbsoft2017/proceedings_files/AnaisSessaoFerramentas_CBSoft2017.pdf>.

SCHRöDER, S.; RIEBISCH, M. Architecture conformance checking with description logics. In: EUROPEAN CONFERENCE ON SOFTWARE ARCHITECTURE. **Proceedings...** 2017. p. 166–172. ISBN 9781450352178. Available from Internet: <https://doi.org/10.1145/3129790.3129812>.

SHAW, M.; CLEMENTS, P. The golden age of software architecture. **IEEE Software**, v. 23, n. 2, p. 31–39, mar. 2006. ISSN 0740-7459. Available from Internet: <http://doi.org/10.1109/MS.2006.58>.

SILVA, L. D.; BALASUBRAMANIAM, D. Controlling software architecture erosion: A survey. **Journal of Systems and Software**, v. 85, n. 1, p. 132–151, jan 2012. ISSN 01641212. Available from Internet: <https://doi.org/10.1016/j.jss.2011.07.036>.

SOLINGEN, R. van et al. Goal question metric (gqm) approach. In: ____. **Encyclopedia of Software Engineering**. [S.l.]: John Wiley & Sons, Inc., 2002. ISBN 9780471028956.

TAN, P.-N.; STEINBACH, M.; KUMAR, V. Introduction to data mining. In: ____. [S.l.]: Addison-Wesley, 2005. chp. Association Analysis: Basic Concepts and Algorithms.

TERRA, R.; VALENTE, M. T. A dependency constraint language to manage object-oriented software architectures. **Software: Practice and Experience**, v. 39, n. 12, p. 1073–1094, 2009. ISSN 1097-024X. Available from Internet: <http://doi.org/10.1002/spe.931>.

TZERPOS, V.; HOLT, R. C. Acdc: an algorithm for comprehension-driven clustering. In: WORKING CONFERENCE ON REVERSE ENGINEERING. **Proceedings...** 2000. p. 258–267. ISSN 1095-1350. Available from Internet: <https://doi.org/10.1109/WCRE.2000.891477>.

WONG, S. et al. Detecting software modularity violations. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING. **Proceeding...** 2011. p. 411–420. ISBN 9781450304450. Available from Internet: <https://doi.org/10.1145/1985793.1985850>.

XIAO, C.; TZERPOS, V. Software clustering based on dynamic dependencies. In: EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING. **Proceedings..** 2005. p. 124–133. ISSN 1534-5351. Available from Internet: <https://doi.org/10.1109/CSMR.2005.49>.

ZAPALOWSKI, V.; NUNES, D. J.; NUNES, I. Understanding architecture non-conformance: Why is there a gap between conceptual architectural rules and source code dependencies? In: BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING.

Proceedings... 2018. p. 22–31. ISBN 9781450365031. Available from Internet: <https://doi.org/10.1145/3266237.3266261>.

ZAPALOWSKI, V.; NUNES, I.; NUNES, D. J. Revealing the relationship between architectural elements and source code characteristics. In: INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION. **Proceedings...** 2014. p. 14–25. ISBN 9781450328791. Available from Internet: <https://doi.org/10.1145/2597008.2597156>.

ZAPALOWSKI, V.; NUNES, I.; NUNES, D. J. The wgb method to recover implemented architectural rules. **Information and Software Technology**, v. 103, p. 125 – 137, 2018. ISSN 0950-5849. Available from Internet: <https://doi.org/10.1016/j.infsof.2018.06.012>.

APPENDIX A — DETAILS OF THE SUBJECT SYSTEMS

In this appendix, we detail each subject systems used in the exploratory study presented in Chapter 3, and in the offline evaluation of the WGB method in Section 5.2.

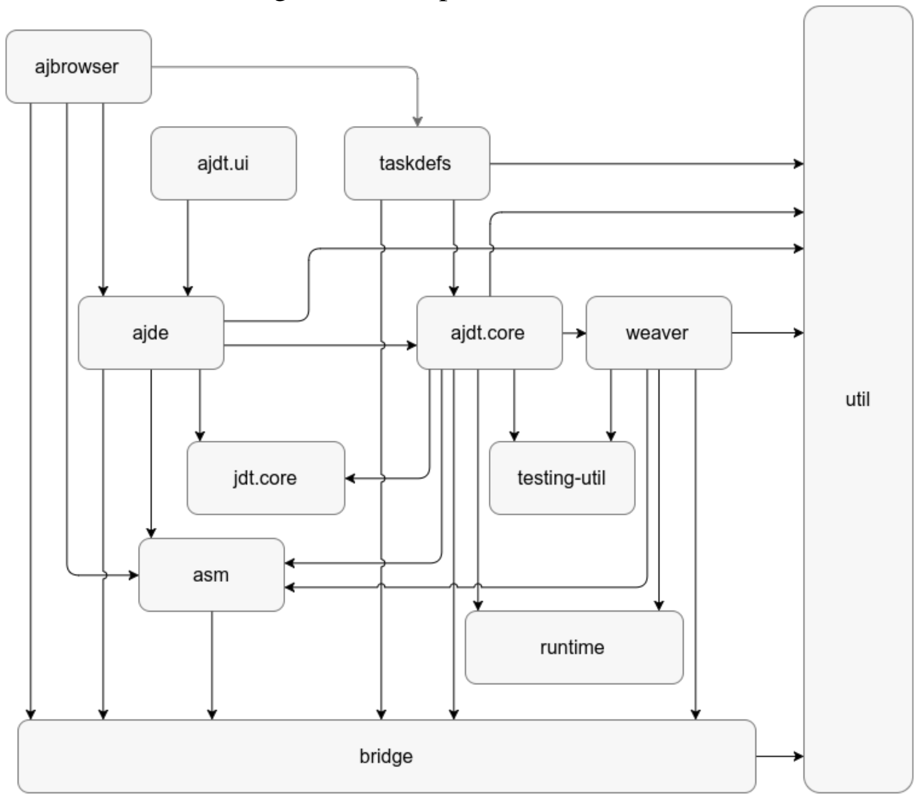
AspectJ is an extension of the Java language that provides support to aspect oriented programming. The conceptual architecture recovered from AspectJ is illustrated in Figure A.1. This architecture was recovered mainly based on the documentation available in the AspectJ official website. This architecture has specific modules. This variation in the module roles lead us to classify it as heterogeneous architecture. The diagram presented in Figure A.1 was used to extract the implemented architecture of AspectJ, which was extracted from the system source code based on its package structure.

ArchStudio is an architecture development environment. It was developed based on the idea of supporting the task of document software architecture. ArchStudio has built-in for modeling the hierarchical structure of complex systems, the types of various components, connectors, and interfaces, product-lines of systems that are related by a common base. ArchStudio is also a architecture meta-modeling environment, which adopts xADL 2.0 architecture description language to model the architectures. The architecture of ArchStudio could not be classified in one architecture pattern due to their size and complexity. Another studies investigated its architecture and represented it as a heterogeneous architecture with specific dependencies between architecture modules. The architecture of ArchStudio was mostly recovered based on the previous analysis of its source code. Garcia, Ivkovic and Medvidovic (2013) provided a module diagram and a mapping between ArchStudio architecture modules and source code package. The recovered information provided by Garcia, Ivkovic and Medvidovic (2013) is available in their study website[1]. Another source of documentation was the analysis provided by Lutellier et al. (2015). They also provided diagrams of the source code of ArchStudio and documented the modules dependencies in their study website[2].

EC is a web application developed to support conference organization. Its features cover paper submission, reviewing processes and notifications of the users involved. It

[1]Availabe at: <https://softarch.usc.edu/wiki/doku.php?id=recoveries:archstudio_4>
[2]Available at: <http://asset.uwaterloo.ca/ArchRecovery/>

Figure A.1: AspectJ Architecture



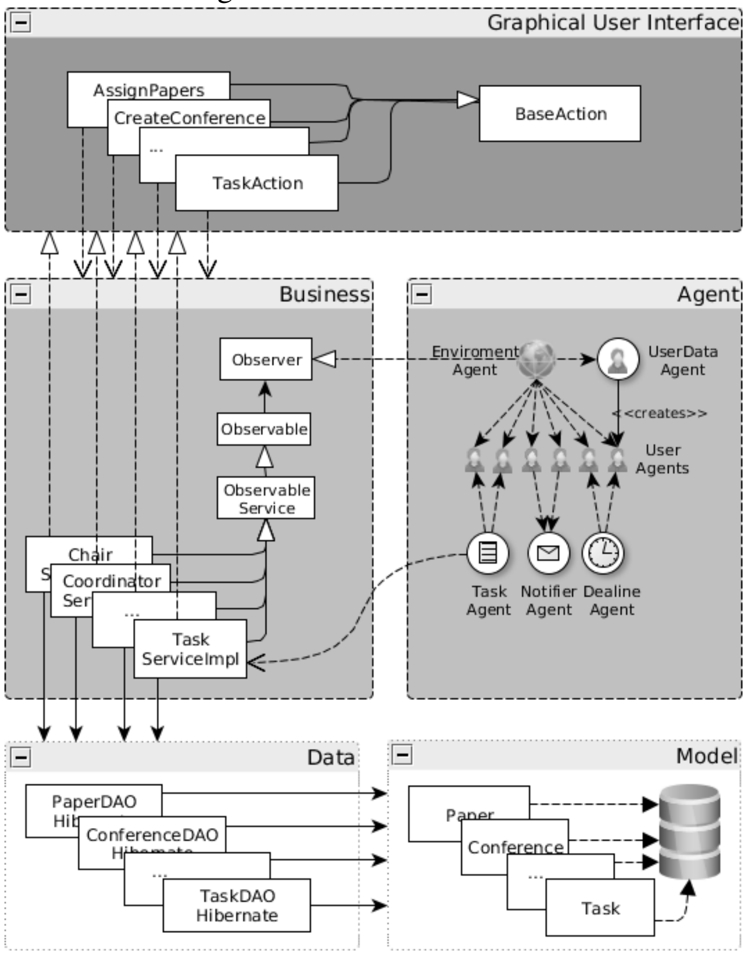
is written in Java uses agent technology. The conceptual architecture recovered from Expert Committee is illustrated in Figure A.2. This architecture was validated by Expert Committee developers and verified by us during the experiment. As can be seen in Figure A.2, this architecture follows the layered pattern with an agent module. From the six architecture rules are explicitly presented in Figure 1, we mapped these rules to package structure of the project. This mapping of rules generated 19 architecture rules analyzed during our experiment. We used the information presented in EC official page [3] and the source code of the system used during our experiment[4].

Metrics is a plugin for Eclipse IDE, which extracts a suite of metrics related to object-oriented good practices from projects developed using Eclipse. It automatically extracts and exports the design metrics – proposed in two books: "OO Design Quality Metrics, An Analysis of Dependencies" and "Object-Oriented Metrics, measures of Complexity" – and also plot a dependency graph of the project. It is conceptual architecture follows an Model-View-Controller pattern with two modules of

[3]Available at:<https://www.inf.ufrgs.br/~ingridnunes/maspl/index.php?base=casestudies&page=expertcommittee>

[4]Available at: <https://www.inf.ufrgs.br/~ingridnunes/maspl/casestudies/ec_work_products.zip>

Figure A.2: EC Architecture



view. One view (Eclipse View) has the responsibility of handle user interaction with the Eclipse and the other (Exporting View) provide the different format of export the projects information. Figure A.3 presents the architecture diagram of Metrics architecture. Metrics' implemented architecture was extracted using its package structure as (sub-)modules to recover the implemented rules. Investigating Metrics' implemented architecture, we noticed a strong dependency from Eclipse model classes since most of the model classes used are from Eclipse API. We provide the link to the official home page[5] and the source code of the system[6] used during our experiment.

OLIS is a web application that provides several personal services to users, developed using a reactive approach. It is composed mainly by two services: the Events Announcement and the Calendar Services. However, the OLIS was designed in such a way that it can be evolved to incorporate new services without interfering

[5]Available at: <https://sourceforge.net/p/metrics2/>
[6]Available at: <https://sourceforge.net/p/metrics2/code/162/tree/>

Figure A.3: Metrics Architecture

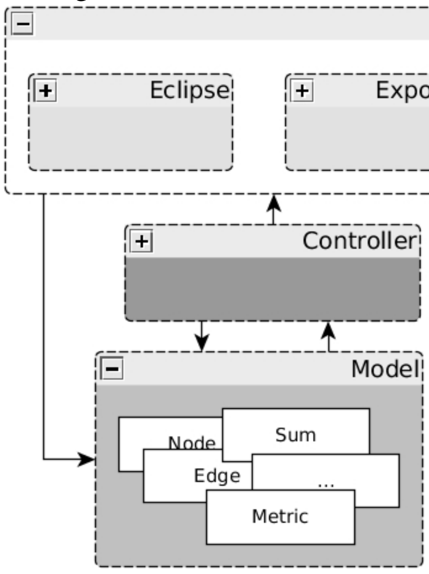
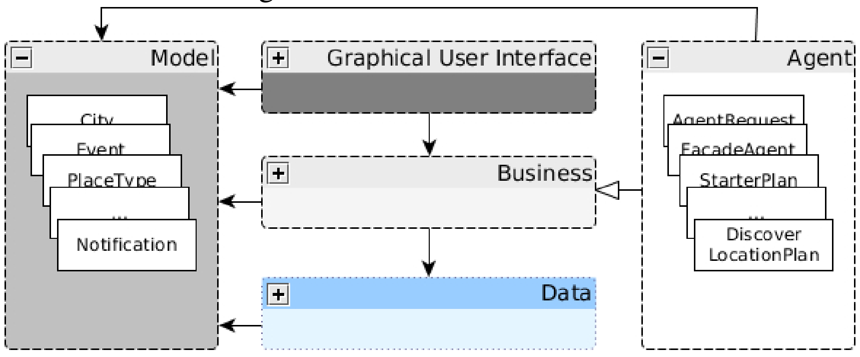


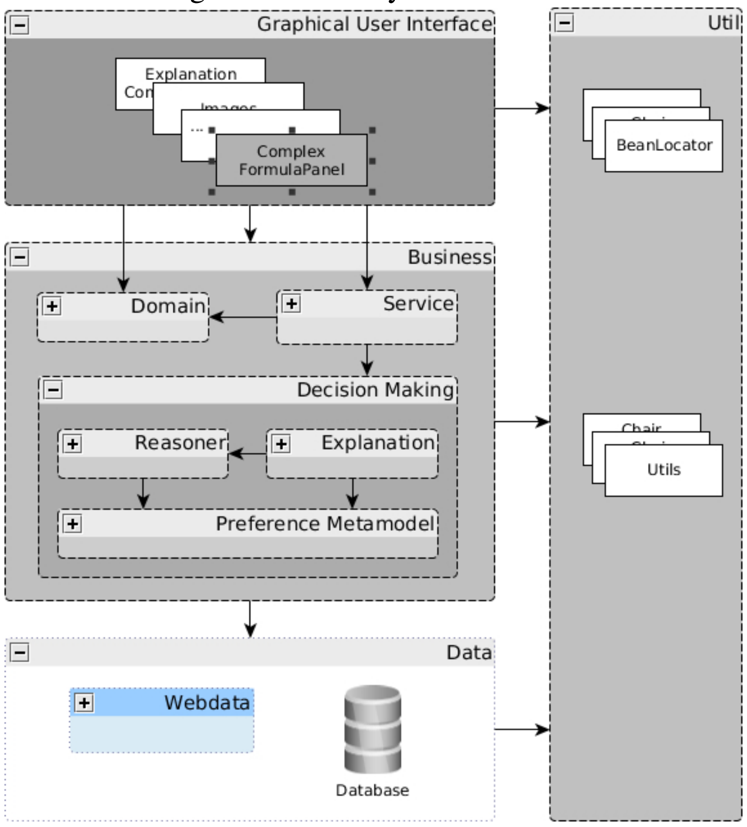
Figure A.4: OLIS Architecture



the existing ones. It is also is written in Java. OLIS conceptual architecture is illustrated in Figure A.4. We presented this diagram to OLIS developers to ensure that this layered architecture represents their conceptual ideas about the system architecture. Furthermore, during the mapping of conceptual rules to implemented rules, we inspected its source again to be sure of the correctness of the 13 conceptual rules stated. The implemented architecture of OLIS was recovered based on its source code. Analyzing the conceptual architecture (Figure A.4), we can notice that the dependency between $Data \rightarrow Agent$ and $Data \rightarrow GUI$ are not allowed. During our inspection of the implemented rules, we noticed two unexpected rules that are clearly not allowed considering Figure A.4. The unexpected rules are $persistence(Data) \rightarrow ui(GUI)$ and $persistence(Data) \rightarrow agent(Agent)$. We also used the information presented in OLIS official page [7] and the source code of the system used during our experiment[8].

[7]Available at:<https://www.inf.ufrgs.br/~ingridnunes/maspl/index.php?base=casestudies&page=olis>
[8]Available at: <https://www.inf.ufrgs.br/~ingridnunes/maspl/casestudies/olis_work_products.zip>

Figure A.5: RecSys Architecture



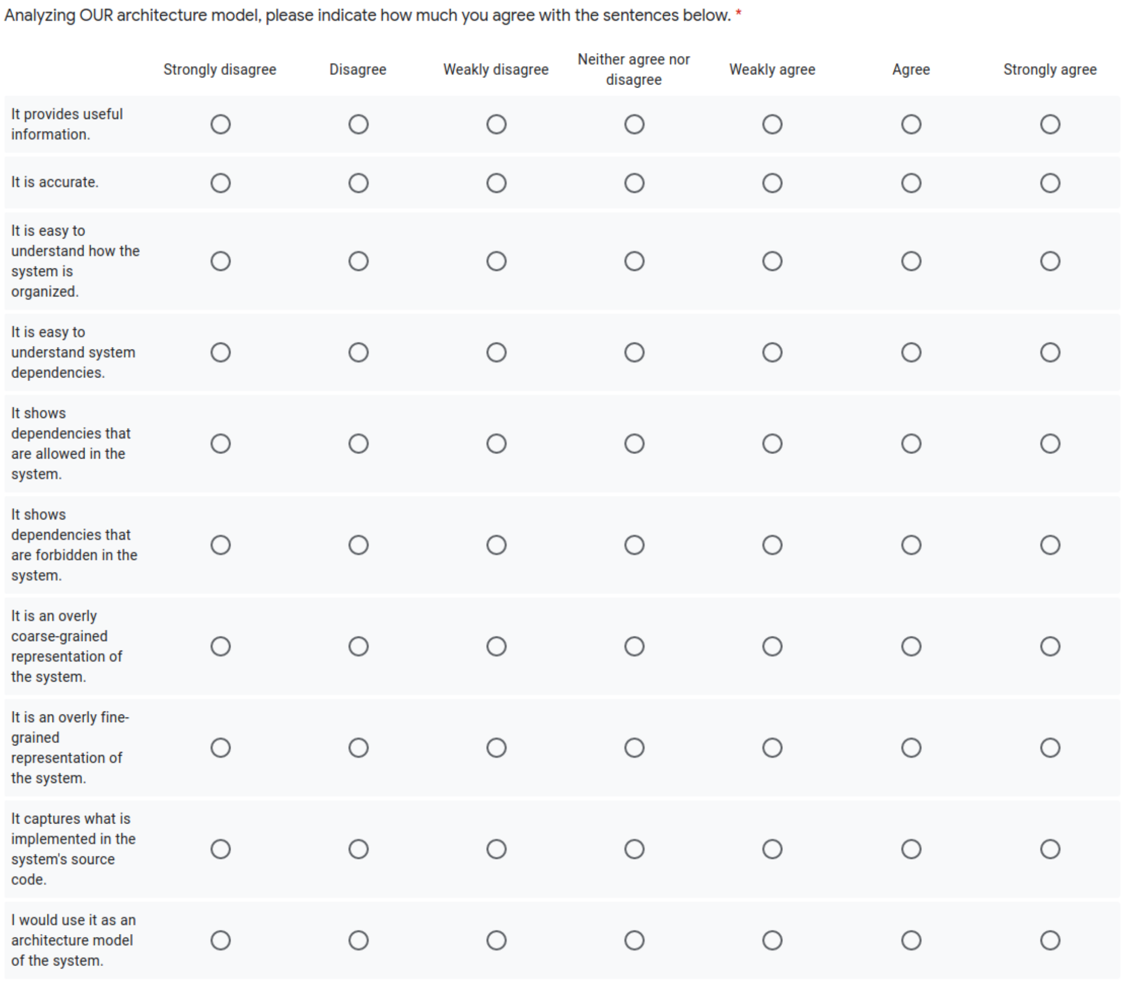
RecSys is a Java desktop-based recommender system that aims to recommend items based on their properties and user-defined preferences. RecSys recommends three types of items: hotels, laptops, and cellphones. The conceptual architecture of RecSys is mainly formed by its business tasks, as can be seen in Figure A.5. Its conceptual architecture follows a layered pattern to organize its $Data$ and $Presentation$ modules. However, the majority of RecSys source code handle business task. This concentration of complexity in one module created a necessity of sub-divide the $Business$ module in other modules to properly organize its architecture.

APPENDIX B — QUESTIONNAIRE TEMPLATE

In this appendix, we present the question of the questionnaire template as they were presented to the participants of the study detailed in Section 5.3. In Figure B.1, we show the 10 questions of first and third part of the questionnaire associated with the evaluation of the documentation built based on the WGB rules. Next, we present the questions related to the second part of the questionnaire, which compares pairs of rules from the WGB rules and the developers' rules. In Figure B.2, we present the questions of generalization and specialization comparisons. In Figures B.3-B.5, the question asked in the implicit, missed, and non-conformance comparisons respectively. Finally, in Figure B.6, we present the last question which is related to the evaluation of the WGB documentation. Additionally, we provide a complete questionnaire online[1] with a made-up architecture to present an example of the questions, comparisons, and documentations of the questionnaires used in our user study.

[1]Available at: <https://forms.gle/TpwGgkuhipZrdLYC8>

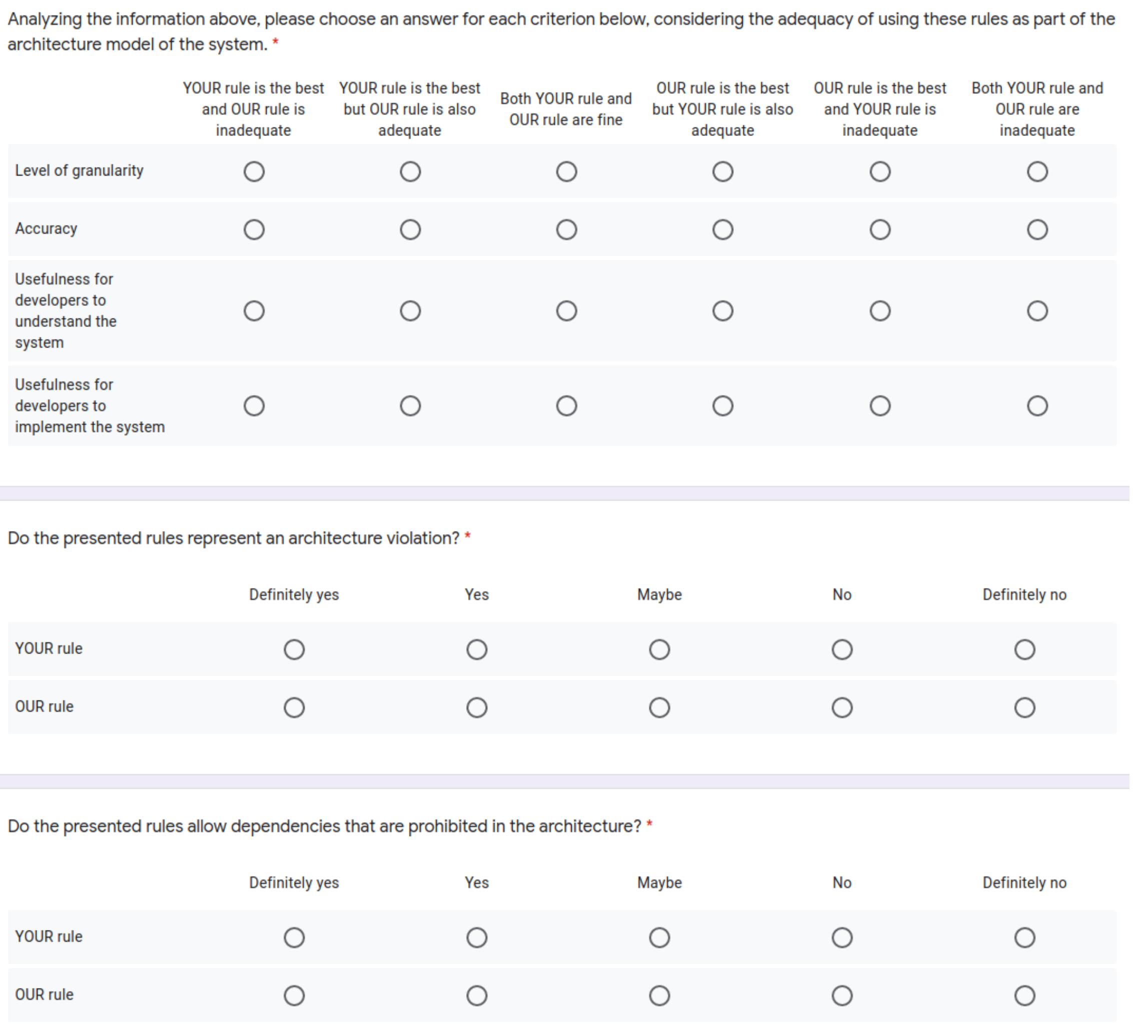
Figure B.1: Questions Asked in the First and Third Part of the Questionnaire.



Analyzing OUR architecture model, please indicate how much you agree with the sentences below. *

	Strongly disagree	Disagree	Weakly disagree	Neither agree nor disagree	Weakly agree	Agree	Strongly agree
It provides useful information.	○	○	○	○	○	○	○
It is accurate.	○	○	○	○	○	○	○
It is easy to understand how the system is organized.	○	○	○	○	○	○	○
It is easy to understand system dependencies.	○	○	○	○	○	○	○
It shows dependencies that are allowed in the system.	○	○	○	○	○	○	○
It shows dependencies that are forbidden in the system.	○	○	○	○	○	○	○
It is an overly coarse-grained representation of the system.	○	○	○	○	○	○	○
It is an overly fine-grained representation of the system.	○	○	○	○	○	○	○
It captures what is implemented in the system's source code.	○	○	○	○	○	○	○
I would use it as an architecture model of the system.	○	○	○	○	○	○	○

Figure B.2: Questions Asked in the Generalization and Specialization Comparisons.



Analyzing the information above, please choose an answer for each criterion below, considering the adequacy of using these rules as part of the architecture model of the system. *

	YOUR rule is the best and OUR rule is inadequate	YOUR rule is the best but OUR rule is also adequate	Both YOUR rule and OUR rule are fine	OUR rule is the best but YOUR rule is also adequate	OUR rule is the best and YOUR rule is inadequate	Both YOUR rule and OUR rule are inadequate
Level of granularity	○	○	○	○	○	○
Accuracy	○	○	○	○	○	○
Usefulness for developers to understand the system	○	○	○	○	○	○
Usefulness for developers to implement the system	○	○	○	○	○	○

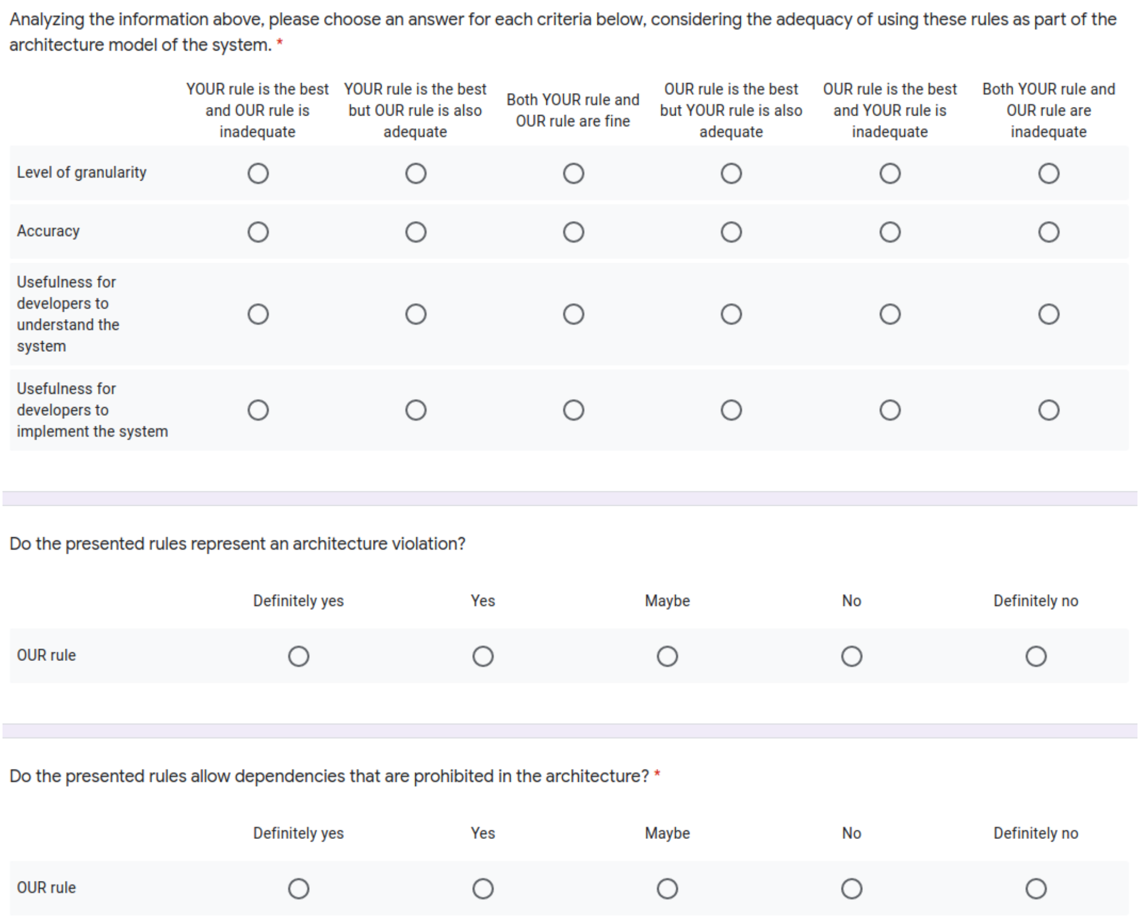
Do the presented rules represent an architecture violation? *

	Definitely yes	Yes	Maybe	No	Definitely no
YOUR rule	○	○	○	○	○
OUR rule	○	○	○	○	○

Do the presented rules allow dependencies that are prohibited in the architecture? *

	Definitely yes	Yes	Maybe	No	Definitely no
YOUR rule	○	○	○	○	○
OUR rule	○	○	○	○	○

Figure B.3: Questions Asked in the Implicit Comparisons.



Analyzing the information above, please choose an answer for each criteria below, considering the adequacy of using these rules as part of the architecture model of the system. *

	YOUR rule is the best and OUR rule is inadequate	YOUR rule is the best but OUR rule is also adequate	Both YOUR rule and OUR rule are fine	OUR rule is the best but YOUR rule is also adequate	OUR rule is the best and YOUR rule is inadequate	Both YOUR rule and OUR rule are inadequate
Level of granularity	○	○	○	○	○	○
Accuracy	○	○	○	○	○	○
Usefulness for developers to understand the system	○	○	○	○	○	○
Usefulness for developers to implement the system	○	○	○	○	○	○

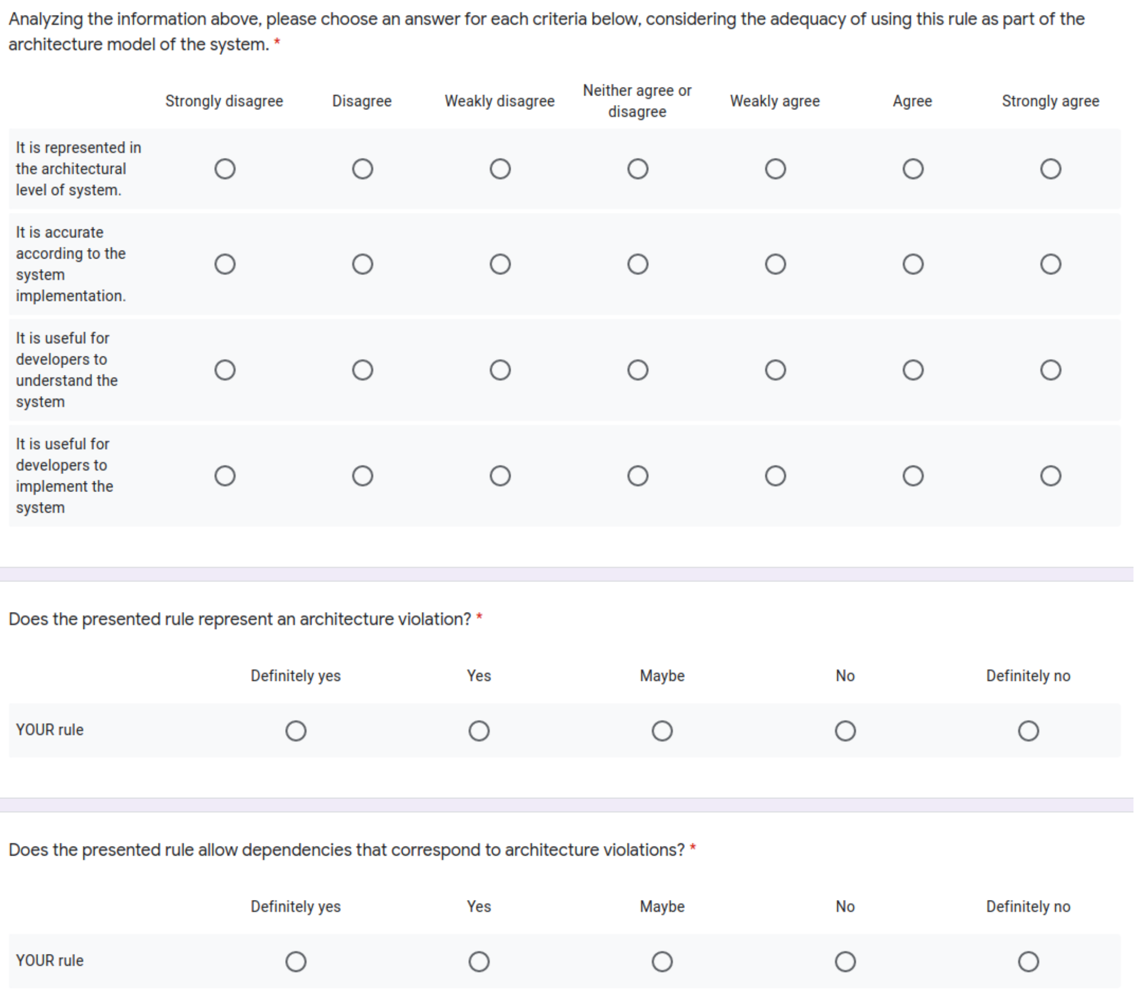
Do the presented rules represent an architecture violation?

	Definitely yes	Yes	Maybe	No	Definitely no
OUR rule	○	○	○	○	○

Do the presented rules allow dependencies that are prohibited in the architecture? *

	Definitely yes	Yes	Maybe	No	Definitely no
OUR rule	○	○	○	○	○

Figure B.4: Questions Asked in the Missed Comparisons.


Analyzing the information above, please choose an answer for each criteria below, considering the adequacy of using this rule as part of the architecture model of the system. *

	Strongly disagree	Disagree	Weakly disagree	Neither agree or disagree	Weakly agree	Agree	Strongly agree
It is represented in the architectural level of system.	○	○	○	○	○	○	○
It is accurate according to the system implementation.	○	○	○	○	○	○	○
It is useful for developers to understand the system	○	○	○	○	○	○	○
It is useful for developers to implement the system	○	○	○	○	○	○	○

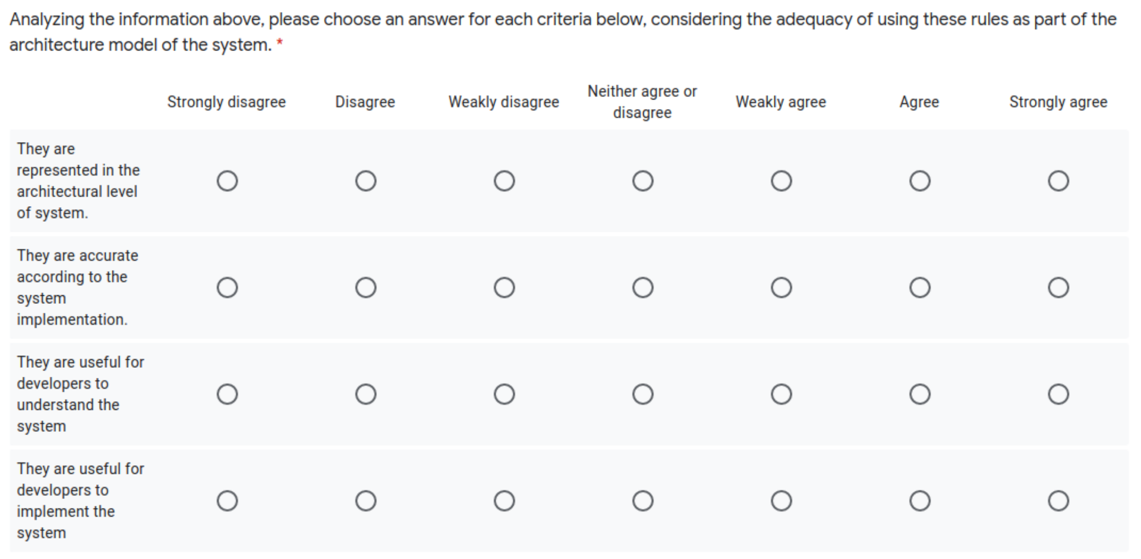
Does the presented rule represent an architecture violation? *

	Definitely yes	Yes	Maybe	No	Definitely no
YOUR rule	○	○	○	○	○

Does the presented rule allow dependencies that correspond to architecture violations? *

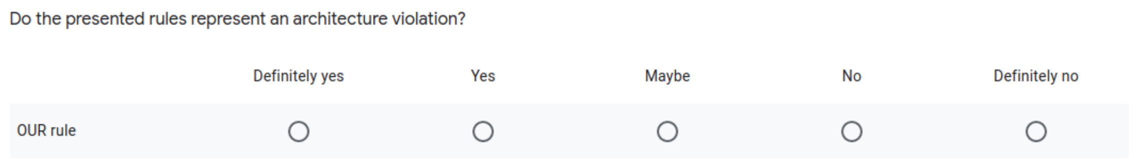
	Definitely yes	Yes	Maybe	No	Definitely no
YOUR rule	○	○	○	○	○

Figure B.5: Questions Asked in the Non-conformance Comparisons.



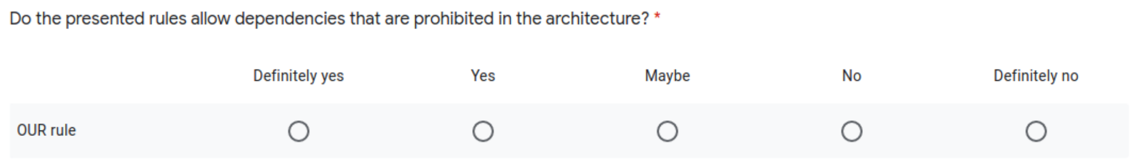
Analyzing the information above, please choose an answer for each criteria below, considering the adequacy of using these rules as part of the architecture model of the system. *

	Strongly disagree	Disagree	Weakly disagree	Neither agree or disagree	Weakly agree	Agree	Strongly agree
They are represented in the architectural level of system.	○	○	○	○	○	○	○
They are accurate according to the system implementation.	○	○	○	○	○	○	○
They are useful for developers to understand the system	○	○	○	○	○	○	○
They are useful for developers to implement the system	○	○	○	○	○	○	○



Do the presented rules represent an architecture violation?

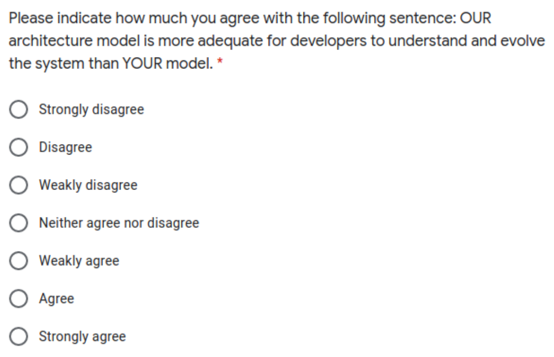
	Definitely yes	Yes	Maybe	No	Definitely no
OUR rule	○	○	○	○	○



Do the presented rules allow dependencies that are prohibited in the architecture? *

	Definitely yes	Yes	Maybe	No	Definitely no
OUR rule	○	○	○	○	○

Figure B.6: Final Question About the WGB Method Appropriateness.



Please indicate how much you agree with the following sentence: OUR architecture model is more adequate for developers to understand and evolve the system than YOUR model. *

- ○ Strongly disagree
- ○ Disagree
- ○ Weakly disagree
- ○ Neither agree nor disagree
- ○ Weakly agree
- ○ Agree
- ○ Strongly agree

APPENDIX C — RESUMO ESTENDIDO

Compreendendo e Recuperando Regras de Arquitetura

Todos os sistemas tem uma arquitetura que estrutura o sistema em módulos que são divididos de acordo com alguns critérios e com regras arquiteturais que especificam dependências permitidas entre os módulos. Os módulos e regras arquiteturais definem os principais conceitos que devem ser seguidos para que a implementação esteja de acordo com a arquitetura especificada. Isso faz com que o software tenha um desenvolvimento e evolução saudáveis e controlados. Idealmente, os módulos devem ser explicitados no código-fonte, por exemplo, por meio de pacotes, que contêm elementos menores de software (classes) que dependem apenas de elementos aos quais são permitidos, de acordo com as regras definidas na arquitetura. Apesar da importância da arquitetura de software, muitos sistemas possuem sua documentação arquitetural desatualizadas ou nem possuem documentação. Isso acontece porque a tarefa de documentar a arquitetura deve ser feita constantemente, consome bastante tempo e precisa ser feita por uma pessoa com conhecimento sobre o software.

A recuperação de arquitetura de software tem sido estudada para reduzir o trabalho necessário para os sistemas que não possuem sua arquitetura documentada ou possuem uma documentação desatualizada terem sua arquitetura documentada adequadamente. Ter a arquitetura do software corretamente documentada e em conformidade com o código-fonte ajuda a entender a estrutura do software, bem como as regras de dependências entre seus módulos que estão implementada. A falta de conhecimento sobre a arquitetura dificulta um desenvolvimento de forma organizada, o que leva a problemas de manutenção que poderiam ser evitados ou minimizados com uma documentação arquitetural bem documentada. Por exemplo, a introdução de violações à arquitetura do software faz com que problemas como o desvio da arquitetura conceitual (*architectural drift*) e erosão arquitetural aconteçam pela falta de conhecimento de como o software deve ser implementado. Além disso, a análise das regras arquiteturais que estão implementadas no código-fonte pode revelar regras não planejadas introduzidas pelos desenvolvedores, que normalmente permanecem não documentadas. Consequentemente, recuperar esse tipo de informação arquitetural minimiza o problema de vaporização do conhecimento (*knowledge vaporization*).

Dado que a arquitetura de software é importante para o desenvolvimento e que existem dificuldades para manter a arquitetura em conformidade com o código-fonte, a questão de pesquisa desta tese é: *Como extrair regras arquiteturais adotadas na implementação de um sistema com base no código-fonte considerando a falta de documentação adequada?*

Para responder a questão de pesquisa é preciso endereçar três problemas importantes: (i) como recuperar regras arquiteturais usando somente dependências de código-fonte; (ii) como distinguir quais dependências de código-fonte são mais adequadas para serem representadas na arquitetura; e (iii) como auxiliar na identificação de violação das regras arquiteturais. Então, nosso objetivo é melhorar a confiabilidade e a conformidade da documentação arquitetural através da análise das dependências de código para recuperação de regras arquiteturais. Com o resultado das análises, nós propomos um método para reduzir o esforço necessário para obter as regras arquiteturais de um sistema. Nosso método é independente de domínio e produz resultados específicos de acordo com as características do sistema. Nosso método identifica regras de arquitetura com base em dependências frequentes e relevantes implementadas. A principal vantagem do nosso método é que ele não precisa de informações adicionais como, por exemplo, uma arquitetura inicial ou uma lista de definições de regras.

Com base nessa ideia, apresentamos a seguir nossa hipótese de pesquisa. *A análise e adoção de uma técnica independente de domínio é efetiva para extrair regras arquiteturais do código-fonte que são específica para cada sistema.*

Existem muitos estudos que investigaram os problemas relacionados a questão de pesquisa desta tese, mas a maioria deles produzem resultados que não são efetivos, i.e. eles produzem uma baixa precisão sobre os módulos e regras recuperados. Isso acontece porque o código-fonte e a arquitetura são abstrações muito diferentes que representam o mesmo sistema. Essa diferença de abstração dificulta a tarefa de manter uma documentação arquitetural atualizada e confiável.

Para entender por que existe uma (grande) diferença entre as regras conceituais de arquitetura e dependências implementadas entre os módulos, nós avalamos e caracterizamos suas divergências do ponto de vista dos pesquisadores no contexto de um estudo multi-projeto. Nesse estudo buscamos entender o quão distante são as regras arquiteturais das dependências de código analisando a conformidade das regras arquiteturais com o código-fonte desses sistemas. Detalhes dos sistemas usados nessa análise são apresentados nas Tabelas 3.2 e 3.3.

Para esse primeiro estudo, nós formulamos três questões de pesquisa: *Q1.* Qual é a diferença entre as regras conceituais de arquitetura e as dependências implementadas entre módulos? *Q2.* Como as dependências implementadas entre módulos podem ser categorizadas em relação às regras conceituais da arquitetura? *Q3.* É possível distinguir as dependências implementadas entre módulos considerando sua categorização?

Para responder essas questões de pesquisa, nós extraímos as dependências de código-fonte considerando módulos e seus elementos dois seis sistemas analisados. As dependências entre módulos são consideras através das estruturas hierárquicas do código, pacotes para a linguagem Java, e as dependência entre elementos são os arquivos ou, por exemplo, classes em Java. Para analisar as informações das dependências implementadas, foi utilizado o algoritmo de regras de associação que mede o suporte de cada uma das dependências. Além disso, nós recuperamos manualmente as arquiteturas dos seis sistemas, que são detalhas no Apêndice A.

Com base nas dependências implementadas e nas regras conceituais da arquiteturas, nós apresentamos os dados referentes à conformidade das dependências permitidas pelas regras da arquitetura conceitual e as dependências de implementadas no código-fonte na Tabela 3.4. Analisando os resultados, fica claro uma grande diferença entre as dependências implementadas e as regras conceituais da arquitetura, com divergências variando de 5,9% a 73,9%. Consequentemente, o número de violações arquiteturais é alto. Além disso, observamos que a quantidade de dependências permitidas que nunca ocorrer no código é ainda maior, variando de 60,3% a 98,2%. Isso indica que as regras da arquitetura devem ser mais detalhadas e restritivas do que são atualmente para serem mais úteis.

Analisando a diferença entre as regras conceituais e as dependências implementadas, nós derivamos quatro categorias de dependências implementas com base nas regras conceituais: *Conceituais* que são exatamente iguais as regras conceituais quando mapeadas para o código; *Sub-conceituais* que são dependências que respeitam as regras conceituais, mas com um nível de granularidade mais baixo; *Intra-módulo* que são dependências permitidas que não existem explicitamente na arquitetura por serem de comunicação interna dos módulos; e *Inesperadas* que são dependências que não se encaixam em nenhum elemento da arquitetura conceitual. Usando essa categorização de dependências, nós classificamos cada uma das dependências extraídas do código-fonte. A distribuição de dependências por categoria é apresentada na Figura 3.2. Desses resultados, um ponto interessante é o grande número de dependências sub-conceituais ($AVG = 29,0\%$) e intra-módulo ($AVG = 21,5\%$) indicando que há necessidade de regras arquiteturais mais detalhadas. Isso por que essas categorias de dependências estão em conformidade com a arquitetura conceitual, mas não estão explicitamente documentadas.

Visando diferenciar as dependências implementadas entre as categorias propostas, nós analisamos o suporte de cada uma das dependências implementadas. Na Tabela 3.5, os valores de suporte de cada categoria com seus respectivos médias, desvios padrão, mediana, mínimo e máximo. Além disso, o gráfico de caixa é apresentado para cada uma das categorias em cada um dos sistemas na Figura 3.3. Analisando esses resultados, o suporte de dependências conceituais é geralmente inferior ao suporte de dependências sub-conceituais (exceto para sistemas com uma hierarquia de módulo plana e pequena), confirmando que as regras de arquiteturais devem ser mais detalhadas. O suporte de dependências inesperadas é significativamente menor do que o suporte de outras categorias de dependência, indicando que essa métrica pode ser usada para identificar grupos de dependências que correspondem a violações de arquitetura ou regras não documentadas.

Dadas as limitações encontradas e as diferenças entre a arquitetura conceitual e o código-fonte encontrados no nosso primeiro estudo, nós propomos um método para recuperação de regras arquiteturais chamado *Weighted-graph-based* (WGB)—método baseado em grafos ponderados. O método WGB escolhe *um conjunto de regras de arquitetura para representar uma arquitetura implementada de software usando como entrada uma determinada estrutura de módulos (por exemplo, estrutura de pacote) e dependências entre elementos de módulo* (por exemplo, classes). Essas regras recuperadas são uma representação de granularidade maior das dependências implementadas, que é uma visão arquitetural do sistema. Nosso método é composto de três etapas sequenciais: (i) cálculo de uma métrica que quantifica a força das dependências entre dois módulos, considerando as dependências entre os elementos; (ii) clusterização em pares de dependências com base na nossa métrica, considerando níveis de módulos de mesmo nível e níveis acima ou abaixo; e (iii) seleção de um conjunto de regras que maximiza as dependências implementadas de acordo com seus valores calculados por nossa métrica visando remover redundâncias nas dependências.

Essas etapas, as entradas e as saídas associadas a cada uma das etapas são ilustradas na Figura 4.1. Com base nas dependências do código-fonte, extraímos métricas (intensidade e distribuição), que são combinadas em uma única métrica, a *Module Strength Dependency* (MDS)—Força de Dependência entre Módulos. O valor de MDS da dependências implementadas é calculado e então usado para remover dependências menos representativas. Por fim, a partir das regras implementadas que são candidatas, selecionamos um conjunto de regras sem informações redundantes e com o maior valor de MDS.

A métrica de intensidade (int) avalia as dependências implementadas entre módulos considerando os elementos internos de cada módulo nos dois sentidos da dependência, i.e. ela avalia as dependências do módulo de fonte e do módulo alvo. Assim a definição da métrica é dada por duas equações, uma para o módulo que é fonte da dependência (Equação 4.2), outra para o módulo que é alvo da dependência (Equação 4.3). Dessa forma a métrica contribui com uma avaliação mais completa da dependência do que as métricas comumente utilizadas para medir dependência que só consideram um sentido da dependência. O cálculo da métrica de intensidade é exemplificado na Figura 4.3a.

Complementando a métrica de intensidade, a métrica de distribuição (dst) mensura as dependências entre módulos analisando dois níveis por vez da estrutura hierárquica do código-fonte e também diferencia os módulos fonte e alvo. Com isso, a análise é feita da perspectiva do módulo pai e seus submódulos filhos diretos em ambas as direções da dependência. A definição dessa métrica é dada também em duas equações, uma equação para a os módulos fonte (Equação 4.6), e outra para os módulos alvos (Equação 4.7). Para o cálculo da métrica de distribuição são considerados os módulos filhos e irmãos para medir quão representativa é a dependência dentro do contexto de níveis. Ou seja, não são considerados os elementos internos de cada módulos. O cálculo da métrica de distribuição é exemplificado na Figura 4.3b.

Essas duas métricas combinadas compõem a MDS que considera as dependências dos elementos dos módulos, dois níveis de granularidade diferentes e os dois sentidos da dependência implementada, como pode ser visto no cálculo de um exemplo de calculo das métricas relacionadas a MDS na Figura 4.4. A MDS é definida na Equação 4.8, onde as distribuições são pesos normalizados para os valores de intensidade. Um exemplo completo de cálculo de cada uma das partes da MDS é apresentando na Tabela 4.2. Esse valores utilizados são com base no exemplo apresentado na Figura 4.2.

O cálculo da MDS nos da valores para as dependência entre pares de módulos considerando módulos filhos e pais. Então, existem quatro formas de representar essas dependências na arquitetura para que não existam conflitos e nem repetições: (i) módulo pai para módulo pai; (ii) módulo pai para módulos filhos; (iii) módulos filhos para módulo pai; e (iv) módulos filhos para módulos filhos. Para decidir qual dessas dependências implementas é melhor, nós fazemos a clusterização das dependência calculadas usando a

MDS fazendo a média entre cada nível delas. Assim, consideramos a média da MDS por nível para selecionar as regras das dependências como regras candidatas. Um exemplo dessa seleção é apresentado na Tabela 4.1, onde é selecionada a dependência de módulo pai para módulo pai por ser a que tem a maior média ($AVG = 0.40$).

As regras candidatas entre os pares de módulos do sistema consideram somente dois níveis de granularidade diferentes. Isso faz com que existam regras redundantes, como no exemplo da Figura 4.5, porque a árvore da estrutura do código-fonte poder ter vários níveis e, logo, várias regras candidatas. Para resolver esse problema, nós modelamos o cenário de regras candidatas como um problema de otimização, formalizado na Equação 4.9, no qual nosso objetivo é selecionar um conjunto de regras candidatas que maximize a MDS geral, eliminando regras candidatas redundantes. Na formalização do problema, nós definimos um peso para regras de mais alto nível que representam várias eliminam várias outras regras quando selecionadas. Definimos dessa forma para que regras de mais alto nível tenham um valor apropriado de acordo com a quantidade de regras que ela elimina caso seja selecionada.

Apresentado o nosso método, nós avaliamos ele com um estudo de caso que mostra detalhes da aplicação em um sistema, um estudo *offline* que analisa a aplicação em seis sistemas mostrando a eficiência e eficácia, e um estudo com desenvolvedores que visa entender a utilidade do resultado gerado.

O estudo de caso é baseado na aplicação do nosso método no sistema MDD4ABMS. Esse sistema é de médio porte tendo 41.3 KLOC, 335 classes e 40 pacotes. Nesse estudo nós recuperamos a arquitetura do MDD4AMBS manualmente e aplicamos o nosso método para comparar as duas documentações da arquitetura. O resultado é o diagrama mostrado na Figura 5.1. Nessa figura são mostradas as regras extraídas utilizando o WGB e a classificação das regras de acordo com a definição da arquitetura recuperada manualmente. Das 44 regras recuperadas, 26 (59%) foram selecionadas para compor a documentação da arquitetura do sistema, que são as regras iguais as definidas na recuperação manual e de granularidade fina comparadas as recuperadas manualmente. As regras de granularidade fina referem-se a submódulos dos módulos referidos nas regras recuperadas manualmente. Por exemplo, existe uma regra conceitual `m2c` $\rightarrow$`mm` que é implementada usando a biblioteca *xtend* que fez com que o WGB recuperasse a regra `m2c.xtend` $\rightarrow$ `mm.statemachine`. Além disso, nós perguntamos ao desenvolvedor mais experiente que avaliasse de maneira geral a documentação gerada por nosso método, usando uma escala Likert de 7 pontos. Ele concordou (fortemente) que o diagrama gerado fornece informações úteis, fornece informações precisas, reflete o que está implementado no código, que será usado como documentação de arquitetura e que facilita a compreensão da organização do sistema.

Para obter uma avaliação mais completa do nosso método, nós analisamos de maneira quantitativa o método WGB em um estudo *offline*. Nosso objetivo com esse estudo é *avaliar nosso método WGB com base nas regras de arquitetura recuperadas por ele que estão implementadas no código-fonte sob a perspectiva dos pesquisadores aplicando nosso método em seis sistemas.* Para atingir esse objetivo, nós aplicamos nosso método nos seis sistemas já mencionados anteriormente, detalhados nas Tabelas 3.2 e 3.3, e analisamos a eficiência e a efetividade dos resultados.

Para avaliar a eficiência, nós medimos o tempo que nosso método leva para executar nos seis sistemas considerando os três passos do método. A Tabela 5.2 detalha essa medição. Como podemos ver, mesmo para o maior sistema, nosso método executa em um tempo aceitável para ser executado frequentemente.

No que diz respeito à eficácia, nós comparamos as regras extraídas pelo WGB com as dependências entre módulos do código-fonte e as regras da arquitetura conceitual recuperada pelos desenvolvedores. Os resultados da comparação entre as regras extraídas pelo WGB e as dependências entre módulos são apresentados na Tabela 5.4. Nessas comparações vemos uma diminuição no número de regras de 87,6%, em média, reduzindo assim a maior parte do esforço dos desenvolvedores na verificação de dependências. A comparação das regras implementadas recuperadas com as regras conceituais mostra que elas diferem amplamente, apresentado na Tabela 5.5, levando a 37,1% e 37,8% de precisão e revocação, respectivamente, em média. Para uma análise qualitativa da regras recuperadas, nós classificamos cada uma delas de acordo com as regras recuperadas conceituais. Essa classificação é apresentada na Tabela 5.7. Essa análise qualitativa das regras recuperadas indica que nosso método: (i) generaliza muitas regras associadas aos submódulos de um módulo como uma única super-regra; (ii) é capaz de capturar regras que ocorrem especificamente entre submódulos, sendo muitas vezes sub-regras de regras conceituais; (iii) identifica regras que governam as dependências dentro de um módulo, que normalmente não são especificadas como regras conceituais; e (iv) deixa as violações de arquitetura como regras refinadas, de modo que seja mais fácil distingui-las de outras regras recuperadas usando nossa métrica de força de dependência.

Nos estudos anteriores apresentados, nós analisamos o WGB usando as regras da arquitetura conceitual como parâmetro para avaliá-lo, supondo que as regras conceituais são a representação mais adequada da arquitetura. Com o objetivo de *investigar se as regras da arquiteturas extraídas usando o método WGB podem melhorar a documentação das arquiteturas.* Esse estudo conta com a participação de desenvolvedores de dois sistemas comerciais que são usados para avaliar a utilidade das regras extraídas pelo WGB. Esse estudo com usuários complementa nossa análise anterior investigando se os desenvolvedores concordam que as regras extraídas pelo WGB são adequadas para serem usadas com documentação dos seus projetos. Detalhes desses projetos são apresentados na Tabela 5.9.

O estudo com usuário foi feito comparando as arquiteturas recuperadas pelos desenvolvedores com as arquiteturas construídas com base nas regras extraídas pelo WGB. Para isso, cada um dos participantes respondeu dois questionários—criados questionários com base em um modelo de questionário—, sendo um para cada sistema. O modelo de questionário tem perguntas que visam entender três pontos principais sobre as regras extraídas pelo WGB: (i) a melhoria na abstração; (ii) os erros ou violações mostrados; e (iii) a utilidade prática. Cada um desses pontos tem uma série de perguntas que são detalhadas na Tabela 5.8. Nos dois primeiros pontos, nós comparamos as regras extraídas pelo WGB com as regras recuperadas pelos desenvolvedores. Já para o terceiro ponto, nós construímos o diagrama de arquitetura com as regras do WGB para que os participantes avaliassem sua utilidade.

As respostas sobre a melhoria na abstração da regras são apresentadas nas Figuras 5.4–5.7. Analisando as respostas das melhorias na abstração das regras, as regras extraídas pelo WGB são consideradas adequadas em 96,5%–100,0%, melhores do que as regras recuperadas em 61,7%–80,0%, e a única regra apropriada em 22,6%–36,5%. Analisando os resultados, nós notamos uma divergência nas respostas para cada comparação entre os participantes, o que pode ocorrer pelas diferentes perspectivas diferentes da arquitetura de acordo com o participante. Diante dessas divergências, consolidamos as respostas analisando a concordância de cada comparação. Esses resultados apresentam as regras extraídas pelo WGB como apropriadas em todas as comparações, e preferidas em 73,9%–95,7% das comparações considerando os quatro critérios analisados. Assim, nosso método recupera regras de arquitetura adequadas e, na maioria dos casos, regras ainda melhores do que as regras recuperadas manualmente. Além disso, os resultados indicam uma preferência por regras de arquitetura mais detalhadas do que as regras de alto nível comumente adotadas.

Dado que as regras extraídas pelo WGB provem melhoria na abstração das regras arquiteturais, nós analisamos os casos em que existem erros. Para analisar os erros foram feitas as mesmas perguntas que para melhoria na abstração para o conjunto de regras que aparecem em somente um dos dois conjuntos de regras. Foram encontrados cinco erros de documentação considerando ambos os conjuntos de regras e ambos os sistemas. As respostas para esse grupo de regras são apresentados na Figura 5.9. Esses erros aconteceram por engano dos desenvolvedores no momento da recuperação manual da arquitetura, no caso das regras recuperadas, e por não serem módulos relevantes, no caso das regras extraídas pelo WGB.

Para entendermos se as regras extraídas pelo WGB ajudam na identificação de violações, nós perguntamos aos participantes sobre a presença de violação e a permissão de violações nas regras apresentadas. As respostas são apresentadas através de gráficos de caixa na Figura 5.10b. Os participantes identificaram poucas violações em ambos conjuntos de regras. Na maioria dos casos em que existem violações é decorrente de um problema de implementação. Além disso, a análise das violações sugere que o principal motivo para considerar uma regra como inadequada é o nível de abstração e não a presença de violações. Chegamos a essa conclusão porque muitas regras recuperadas pelos desenvolvedores foram reportadas como inadequadas, mas poucas foram reportadas tendo violações.

Dado que individualmente as regras extraídas pelo WGB são adequadas, é preciso analisar se elas em conjunto provem uma arquitetura apropriada. Isso é investigado na primeira e última parte dos questionários respondidos pelos participantes, onde são feitas as mesmas perguntas para os participantes. Isso possibilita compararmos as respostas antes e depois de eles analisarem as regras individualmente. As respostas dos participantes sobre a documentação construída com as regras extraídas pelo WGB é apresentada na Figura 5.11. De acordo com as respostas, a documentação construída com base nas regras do WGB é adequada considerando os critérios de qualidade e utilidade. Embora seja considerada muito detalhada em grande parte dos casos, os participantes preferem a documentação construída com base nas regras extraídas pelo WGB ao invés da documentação de alto nível comumente adotada. Portanto, a documentação fornecida pelo método WGB é mais detalhada, mas também mais útil. Além disso, a inspeção individual das regras teve pouca influência nas respostas dos participantes sobre a documentação visto que houveram poucas mudanças nas respostas.

Como resultado desta tese, uma série de contribuições podem ser enumeradas. Essas contribuições apresentam a motivação, o desenvolvimento e a avaliação do método de recuperação de regras arquiteturais WGB. As principais contribuições são listadas a seguir.

Revisão dos Estudos de Recuperação de Arquitetura. Fornecemos uma revisão da pesquisa relacionada à recuperação da arquitetura, apresentando-as com base em sua entrada, saída, análise e propósito.

Análise Empírica de Conformidade Arquitetural. Apresentamos um estudo exploratório que avalia e investiga a diferença na abstração entre as regras da arquitetura conceitual e as dependências entre os módulos implementados no código-fonte (ZAPALOWSKI; NUNES; NUNES, 2018). Esse estudo o quão complexa é a tarefa de manter a documentação da arquitetura em conformidade com o código-fonte porque a abstração comumente usada para documentar arquiteturas não é facilmente mapeada para o código-fonte. Com base nos resultados deste estudo, derivamos quatro categorias de relacionamentos entre regras conceituais e dependências de código-fonte que foram analisadas.

Método WGB para Recuperar Regras de Arquitetura. Propusemos o método *Weighted-graph-based* (baseado em grafo ponderado) para recuperar regras de arquitetura (ZAPALOWSKI; NUNES; NUNES, 2018), que é automático, independente de domínio e específico de sistemas. Nosso método não requer a especificação de nenhum limite ou personalizações específicas do sistema. Além disso, nosso método inclui o cálculo de uma nova métrica, nomeada como MDS, para medir a força da dependência entre módulos e a modelagem do problema de recuperação de arquitetura como um problema de otimização.

Avaliação *Offline* do Método WGB. Avaliamos o desempenho e a eficácia do método WGB (ZAPALOWSKI; NUNES; NUNES, 2018). Os resultados mostram que nosso método produz resultados em tempo hábil. Com relação à eficácia, nosso método atinge uma redução das dependências do módulo para regras recuperadas. Além disso, a comparação das regras implementadas recuperadas com as regras conceituais mostra que elas diferem amplamente. Uma análise qualitativa das regras recuperadas mostra que nosso método recupera da arquitetura dentro das quatro categorias identificadas e principalmente regras que provem mais detalhes sobre a arquitetura implementada. Nosso método também recupera violações arquiteturais, caso elas existam, como regras refinadas, de modo a facilitar distingui-las de outras regras recuperadas através do valor atribuído a essa regra.

Avaliação com Usuários do Método WGB. Avaliamos nosso método a partir da perspectiva dos desenvolvedores para entender a utilidade do nosso método. Esse estudo indica que o método WGB extrai regras de arquitetura mais adequadas do que as regras de arquitetura extraídas pelos desenvolvedores dos sistemas na maioria dos casos. Eles também indicaram que usariam as regras WGB como documentação de arquitetura. Além disso, os resultados reforçam que as regras arquiteturais devem ser mais detalhadas devido à preferência dos desenvolvedores pelas regras WGB, principalmente em relação a regras mais detalhadas—comparações de especialização e implícitas.

Em resumo, esta tese avança na pesquisa de recuperação de regras de arquiteturais de software. É evidente que ainda há muito a se fazer para ter um método que automatize completamente a recuperação de arquitetura, mas nosso trabalho consiste em um passo para reduzir o esforço necessário para se ter um documentação de arquitetura confiável e atualizada.