Bachelor's Thesis

Exploring Software Architecture Using Data-Flow-Based Design Rule Spaces

Mohammed Saad

January 30, 2026

Advisor:
Sebastian Böhm Chair of Software Engineering

Examiners:
Prof. Dr. Sven Apel Chair of Software Engineering
Prof. Dr. Tomohiro Nagashima Department of Computer Science

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

UNIVERSITÄT
DES
SAARLANDES

Erklärung Statement

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne die Beteiligung dritter Personen verfasst habe, und dass ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus Veröffentlichungen oder aus anderweitigen fremden Äußerungen entnommen wurden, sind als solche kenntlich gemacht. Insbesondere bestätige ich hiermit, dass ich bei der Erstellung der nachfolgenden Arbeit mittels künstlicher Intelligenz betriebene Software (z. B. ChatGPT) ausschließlich zur Textüberarbeitung/-korrektur und nicht zur Bearbeitung der in der Arbeit aufgeworfenen Fragestellungen zu Hilfe genommen habe. Alle mittels künstlicher Intelligenz betriebenen Software (z. B. ChatGPT) generierten und/oder bearbeiteten Teile der Arbeit wurden kenntlich gemacht und als Hilfsmittel angegeben. Ich erkläre mich damit einverstanden, dass die Arbeit mittels eines Plagiatsprogrammes auf die Nutzung einer solchen Software überprüft wird. Mir ist bewusst, dass der Verstoß gegen diese Versicherung zum Nichtbestehen der Prüfung bis hin zum Verlust des Prüfungsanspruchs führen kann.

I hereby declare that I have written this thesis independently and without the involvement of third parties, and that I have used no sources or aids other than those indicated. All passages taken directly or indirectly from publications or other external sources have been identified as such. In particular, I confirm that I have used AI-based software (e.g., ChatGPT) exclusively for the following permitted sub-tasks: text rewriting/revision, and not to address or formulate the main research questions of the thesis. All parts of the thesis that were generated and/or edited using AI-based software (e.g., ChatGPT) have been disclosed and documented in accordance with the documentation requirements. I agree that the thesis may be checked using plagiarism detection software, including checks for the use of such software. I am aware that any violation of this declaration may result in failing the examination and lead to losing the right to be examined.

Saarbrücken, _____ _____
(Datum Date) (Unterschrift Signature)

Einverständniserklärung (optional) Declaration of Consent (optional)

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, _____ _____
(Datum Date) (Unterschrift Signature)

Abstract

Modern software systems are becoming increasingly complex, making architectural understanding essential for their effective maintenance and evolution. However, many existing architecture recovery methods fall short of capturing this complexity by assigning software elements a single architectural role, even though software elements often participate in multiple architectural concerns. Design Rule Spaces (DRSpaces) address this limitation by allowing software elements to assume multiple roles depending on the dependency relation under analysis, with each DRSpace providing a distinct architectural perspective. Existing DRSpaces research has focused primarily on static relations and evolutionary coupling, while data-flow has the potential to expose additional architectural properties.

This thesis explores the use of data-flow as a relation for constructing DRSpaces and evaluates whether they reveal architectural insights beyond those provided by static relations and evolutionary coupling. We compare data-flow-based DRSpaces with DRSpaces constructed from these other relations. Our evaluation combines quantitative and qualitative analyses to examine the structural properties of data-flow DRSpaces and to assess how other relations align with data-flow and what this alignment implies for the analyzed systems.

Our results show that data-flow-based DRSpaces are often structurally more complex than DRSpaces constructed from static relations, with tighter coupling between files, and they are frequently closer in structure to co-change DRSpaces. In addition, we find that using data-flow as a secondary relation within DRSpaces constructed from static relations provides a clear indication of a system's modularity with respect to data-flow. Generally, our findings demonstrate that data-flow offers a complementary architectural perspective for software architecture analysis.

Acknowledgments

I would like to express my deep gratitude for the support and encouragement I received throughout the writing of this thesis.

First, I would like to thank my advisor, Sebastian Böhm, for his guidance, continuous support, and for always being available to answer questions and provide feedback during this work. I would also like to thank Prof. Dr. Sven Apel, who has been an inspiration throughout my studies, for providing the opportunity to write this thesis at his chair and for examining this work.

Finally, I would like to thank my family and friends for their constant and incredible support throughout this journey.

Contents

List of Figures

List of Tables

Acronyms

DSM Design Structure Matrix

DRH Design Rule Hierarchy

DRSpace Design Rule Space

IL Implement relation

1 Introduction

Software architecture is one of the most important decisions in the design and development of large-scale and complex software systems. It defines the fundamental structures of a software system, including its elements, the relationships between them, and the properties that are essential to reason about the system [4]. These structures capture design decisions that shape the organization and evolution of the system.

Architectural decisions are difficult to change once the system is built. As a result, flawed architectural choices can be costly and often lead to workarounds, code duplication, and tight coupling between components. These issues are often referred to as technical debt, a metaphor used to describe software quality problems that arise from taking design shortcuts. Although technical debt can exist at many levels of a software system, studies have shown that architecture is the leading source of technical debt in source code [11].

To help analyze the architecture of large-scale systems, researchers have developed various approaches to recover architecture from source code based on structural, textual, and evolutionary criteria [9], [10], [15], [16], [19], [22]. For example, there is a search-based approach that groups software elements into modules by optimizing cohesion and coupling metrics [22]. These are two key principles of good modular design that enable the recovery of meaningful architectural information, especially in large or legacy systems where the original modularity may have eroded over time. Another technique is an approach that groups software elements based on lexical information, such as comments and method names [9]. Although these approaches provide different and valuable perspectives on the architecture of a system, they typically enforce a restrictive clustering model in which each software element is assigned to exactly one cluster. In practice, this assumption often fails to capture the complexity of real-world systems, where individual components may serve multiple roles and participate in multiple architectural concerns simultaneously.

Design Rule Spaces (DRSpaces) were introduced by Xiao et al. [25] to overcome this limitation. They are based on Baldwin and Clark's *Design Rule Theory* [3]. They define *design rules* as critical architectural decisions that decouple a system into independent modules that can be developed, modified, or replaced without affecting other parts of the system, thus increasing flexibility and long-term value.

DRSpaces operationalize this theory by grouping software elements around design rules using a specific dependency relation. Rather than producing a single architectural view, DRSpaces construct separate architectural spaces for different relations, allowing software elements to assume different roles depending on the relation under analysis. This results in multiple, potentially overlapping DRSpaces, each providing a distinct architectural perspective on the system. The relations used in DRSpaces typically capture static relations

such as inheritance, method calls, or module usage, each reflecting how software components interact or depend on one another.

In the context of DRSpaces, the software architecture is analyzed at the source-file level, where the architectural structure is defined by the relationships between files. Existing work on DRSpaces has focused primarily on static relations such as inheritance/realization, aggregation, and dependency, as well as evolutionary relations derived from revision histories. However, there remains an opportunity to explore the potential of DRSpaces using other types of dependencies. Specifically, data-flow dependencies could offer architectural insights that are not captured by static relations alone. Data-flow dependencies describe how data are propagated between components, and therefore encode relations that may not be visible in static relations. When used alongside static and evolutionary relations, data-flow dependencies have the potential to expose hidden coupling and modularity issues at the architectural level.

1.1 Goal of this Thesis

The goal of the thesis is to explore the use of data-flow dependencies as a relation in the context of Design Rule Spaces for software architecture analysis.

We investigate whether data-flow dependencies provide complementary architectural insights when used to construct DRSpaces. Specifically, we examine whether data-flow–based DRSpaces exhibit structural properties that differ from those obtained using static and evolutionary (co-change) relations. To this end, we construct DRSpaces for a set of software systems using data-flow dependencies, multiple static relations, and co-change and compare their resulting architectures both quantitatively and qualitatively.

In addition, we take advantage of the hierarchical nature of DRSpaces to analyze how different dependency relations align with the data-flow-based DRSpaces, and conversely, the extent to which data-flow dependencies align with the DRSpaces constructed from these relations. Through this analysis, we evaluate the potential of data-flow-based DRSpaces as a complementary perspective for software architecture analysis.

1.2 Overview

The remainder of this thesis is structured as follows.

Chapter 2 introduces the key terminology that is relevant to this research and necessary to understand the concepts used throughout the thesis. In Chapter 3, we present the research questions and describe the methodology used in this study, including the extraction and processing of the dependency data and the construction of DRSpaces for analysis. In Chapter 4, we apply the proposed methodology to a set of selected open source systems and evaluate the research questions based on the obtained results. We also discuss and interpret the observed patterns and differences across dependency relations. Afterwards, we outline the potential threats to the internal and external validity of our results. Chapter 5 reviews the existing literature in the areas of software architecture recovery, Design Rule

Spaces and data-flow analysis. Finally, Chapter 6 concludes this thesis with a summary of the main findings and a discussion of potential directions for future research.

2

Background

In this chapter, we introduce the core terminology and concepts upon which the architectural analysis presented in this thesis is built.

For illustrative purposes, this chapter uses examples derived from the *fmt* project, an open source C++ formatting library. The project was analyzed using the *DV8* [1] tool. These examples are intended to clarify the key concepts discussed throughout the chapter. A more detailed explanation of the tool and its configuration is provided in Section 3.3.2.

2.1 Design Rule Theory

The concept of *Design Rule Theory* was proposed by Baldwin and Clark [3]. According to this theory, a modular software system is structured around a set of top-level *design rules* that represent the most critical architectural decisions in the system. These design rules are intended to be stable and define constraints and interfaces that partition the rest of the system into independent modules. As a result, the independent modules can be developed, modified, or replaced without affecting other parts of the system. In this way, design rules enable and enforce modularity in software systems. In modern object-oriented systems, design rules are often implemented through interfaces, abstract classes, or architectural design patterns. For example, in the *Factory Method* design pattern, a factory interface defines a method for creating objects, while concrete factories implement this method to instantiate specific classes. This factory interface is considered the *design rule* of the system because it encapsulates a critical architectural decision, that is, how objects are created and decoupled from their usage. This allows concrete factories and products to form independent modules and new implementations can be added or modified without impacting other parts of the system, thus maintaining modularity.

2.2 Design Structure Matrix

A Design Structure Matrix (DSM) is an $N \times N$ matrix used to visually represent the dependencies between elements of a system. Each row and column of the DSM correspond to a system element in the same order. These elements could be classes, modules, packages, or any other design parameter of a system. The cells at the intersection of rows and columns

1 https://archdia.com/pages/dv8-user-guide

indicate whether there are dependencies from the elements represented by the rows to the elements represented by the columns.

DSM is used in this study to model the source files of a software project. Each row and column correspond to a source file within the project, which is indexed in the same order. The cells indicate dependencies between the files. Dependencies between files are marked by a symbol (e.g., an x) in the corresponding cell. For example, if file A (row) depends on file B (column), the cell at position (A, B) is marked. The diagonal cells represent self-dependencies and are annotated with the file's own index. The square blocks along the diagonal model sets of files that are grouped together.

In Figure 2.1 the structure of the *fmt* project is modeled using a DSM, which captures the dependencies between the source files. All dependencies represented in this DSM are static relations. Each dependency is indicated by an x in the corresponding cell of the matrix. For example, cell $(2, 5)$ shows that there is a dependency from the source file in the row 2 to the file in the column 5. Files belonging to the same package are grouped and marked with a square block around their rows and columns. In this example, files from the indexes $2 - 4$ are grouped because they belong to the `src` package.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1 test	(1)	x			x	x	x	x	x	x	x	x	x	x	x	x	x
2 src/os.cc		(2)			x			x				x	x				
3 src/fmt.cc			(3)			x	x		x	x	x		x	x	x	x	x
4 src/format.cc				(4)	x							x	x				
5 include/fmt/base.h					(5)												
6 include/fmt/args.h					x	(6)							x				
7 include/fmt/chrono.h	x				x		(7)						x				
8 include/fmt/os.h					x			(8)					x				
9 include/fmt/compile.h					x				(9)				x				
10 include/fmt/std.h	x				x		x			(10)			x			x	
11 include/fmt/color.h				x	x						(11)	x	x				
12 include/fmt/format-inl.h	x			x	x		x				x	(12)	x				
13 include/fmt/format.h	x				x						x	x	(13)				
14 include/fmt/ranges.h	x				x		x					x	x	(14)			
15 include/fmt/printf.h	x				x		x					x	x		(15)		
16 include/fmt/ostream.h				x	x		x				x	x	x			(16)	
17 include/fmt/xchar.h				x	x						x	x	x	x		x	(17)

Figure 2.1: A DSM representing the structure of the fmt project

2.3 Design Rule Hierarchy

Design Rule Hierarchy (DRH) is an algorithm proposed by Cai et al. [12], [7] to recover the architectural structure of a software system based on the relationships between its components. DRH follows the principles of *Design Rule Theory* by identifying *design rules* and *independent modules* and organizing them into a hierarchical, layered structure.

Given a set of components and the dependencies between them, DRH partitions the system into layers such that dependencies flow only upward: components in a layer may depend on components in the same or higher layers, but never on components in lower

layers. It follows that the top layer contains components that do not depend on components in any other layer and therefore the top layer contains the *design rules* of the system with respect to the relation used. The lower layers are considered the *independent modules* of the system. Each layer is further decomposed into modules, where a module consists of a set of components that share identical dependency patterns. The modules within a layer are also organized in the same dependency flow as the outer layers. This hierarchical decomposition is applied recursively, yielding a nested structure of layers and modules. Note that design-rule *independent modules* are a theoretical concept, while the DRH algorithm's modules are the recursively defined clusters within the top-level layers.

In this study, DRH is applied at the source-file level to cluster files based on a specific dependency relation. The resulting hierarchical structure is best visualized using a DSM because it can show the layered structure of the cluster, which is very difficult to express in other visual models.

Figure 2.2 shows the DRH clustering of the *fmt* project, modeled using a DSM, based on inheritance dependencies, specifically the static relations `Implement` and `Extend`. The DSM reveals four layers. They are marked with square blocks along the diagonal. Layer L_1 contains only one file (f_1), `base.h`, which is extended or implemented by other files but does not depend on any of them. With respect to the inheritance relation, this layer represents the design rule of the system. Layer L_2 contains one file (f_2), `format.h`, which depends only on L_1 and therefore forms a separate layer. Layer L_3 contains the files (f_3–f_{12}) and consists of multiple modules, such as (f_3–f_5) and (f_6–f_8), which are grouped because they share identical dependencies on the higher layers L_1 and L_2. They are marked by an additional square block inside the layer. Finally, layer L_4 contains the files (f_{13}–f_{16}), these are the files that neither depend on nor are depended upon by other files with respect to the inheritance relation and therefore form a leaf layer.

As shown in Figure 2.2, the files are organized by DRH so that dependencies occur only between files within the same layer or from files in lower layers to files in higher layers. This property holds both at the layer level and within the recursively defined modules.

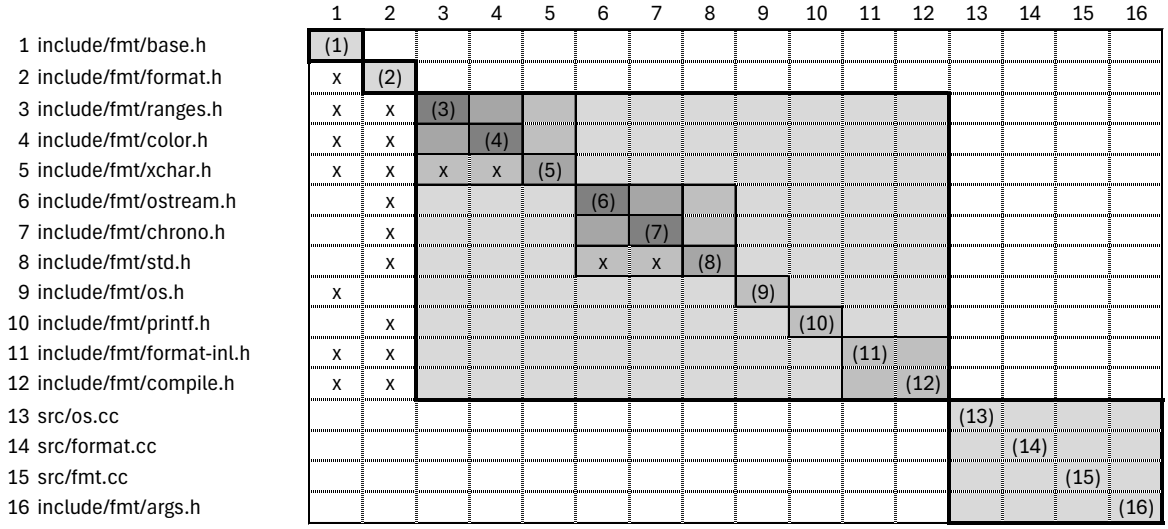

Figure 2.2: A DSM of the DRH clustering of the fmt project

2.4 Design Rule Spaces

Design Rule Spaces (DRSpaces) are a software architecture modeling approach based on *Design Rule Theory* introduced by Xiao et al. [25]. A DRSpace provides an architectural view of a software system constructed by clustering source files based on the relationships between them using the DRH algorithm. Each DRSpace captures the architectural structure induced by a single relation and represents it as a layered DSM. DRSpaces are defined over the full set of source files of a software project. However, only files participating in the selected dependency relation influence the resulting architectural structure and thus only this subset of files is contained in the respective DRSpace. A DRSpace is defined by three key characteristics:

(1) A DRSpace consists of a set of files and selected types of relations between them. DRSpaces can be constructed using various types of relations, such as static relations (e.g., inheritance, aggregation, dependency) or evolutionary relations derived from version control histories. In the case of evolutionary coupling, two files are considered related if they co-changed together in the same commit, and the number of such co-changes determines the weight of the relation.

For example, Figure 2.4 shows the value 12 in the cell $(5, 1)$, which is the number of co-changes between the files `ostream.h` and `base.h`.

(2) A DRSpace must contain one or more leading files, which correspond to the *design rules* of the space. We distinguish between leading files and design rules. Design rules refer to key architectural decisions that affect the entire system, while leading files are the most important files within the DRSpace, which may include only a subset of the system's files. If a DRSpace consists of multiple layers, we consider the files in the first layer to be the leading files of the DRSpace. If a DRSpace has only one layer, then all files in that layer are the leading files.

For example, in Figure 2.3 `base.h` is the only file in the first layer, therefore we say the inheritance DRSpace is led by the file `base.h`. This also means that `base.h` is the most important file within the inheritance DRSpace.

(3) A DRSpace must be clustered using the DRH algorithm, which automatically identifies the layered structure according to the selected relations. The relations used during clustering are called primary relations because they define the meaning of the DRSpace. All other relations that are used to analyze the DRSpace after it was clustered are called secondary relations.

For example, in Figure 2.3 the DRSpace is clustered using the static relations `Implement` and `Extend`. We refer to `Implement` and `Extend` as the primary relations of this DRSpace. These relations represent object-oriented inheritance, hence we refer to the resulting DRSpace as an inheritance DRSpace. Figure 2.4 presents the same DRSpace as in Figure 2.3 but the DRSpace has an additional relation, which is co-change between files. The numbers in the cells represent the number of co-changes between file pairs. Since this relation was not used for clustering, we refer to it as a secondary relation of the DRSpace.

A key feature of DRSpaces is that they support overlapping architectural views. Because each DRSpace is constructed from a different dependency relation, a file may have different roles in multiple DRSpaces. This reflects the reality that software components often partici-

	1	2	3	4	5	6	7	8	9	10	11	12
1 include/fmt/base.h	(1)											
2 include/fmt/format.h	Ext,Impl	(2)										
3 include/fmt/xchar.h	Impl	Impl	(3)	Impl								
4 include/fmt/ranges.h	Ext,Impl	Impl		(4)								
5 include/fmt/ostream.h		Impl			(5)							
6 include/fmt/chrono.h	Impl	Impl				(6)						
7 include/fmt/std.h		Ext,Impl			Ext	Impl	(7)					
8 include/fmt/os.h	Ext							(8)				
9 include/fmt/color.h	Ext,Impl	Impl							(9)			
10 include/fmt/format-inl.h	Ext,Impl	Impl								(10)		
11 include/fmt/compile.h	Impl	Impl									(11)	
12 include/fmt/printf.h		Ext										(12)

Figure 2.3: Inheritance DRSpace of the fmt project

	1	2	3	4	5	6	7	8	9	10	11	12
1 include/fmt/base.h	(1)	88										
2 include/fmt/format.h	Ext,Impl,88	(2)										
3 include/fmt/xchar.h	Imp,17	Impl,22	(3)	Impl,5								
4 include/fmt/ranges.h	Ext,Impl,15	Impl,20	5	(4)								
5 include/fmt/ostream.h	12	Impl,14	8	5	(5)	6	4					
6 include/fmt/chrono.h	Impl,20	Impl,24	8	6	6	(6)	9					
7 include/fmt/std.h	13	Ext,Impl,16	6	5	Ext,4	Impl,9	(7)					
8 include/fmt/os.h	Ext,9	15	6	5	8	5	3	(8)				
9 include/fmt/color.h	Ext,Impl,3	Impl,7	3	4	6	1		5	(9)		2	
10 include/fmt/format-inl.h	Ext,Impl,13	Impl,17	6	3	2	4	4	4		(10)	4	
11 include/fmt/compile.h	Impl,12	Impl,16	4	5	3	8	4	5	2	4	(11)	
12 include/fmt/printf.h	17	Ext,24	8	7	6	8	6	6	1	5	4	(12)

Figure 2.4: Inheritance DRSpace of the fmt project with Co-change as secondary relation

pate in multiple architectural concerns that cannot be captured by approaches that produce a single global clustering.

In prior work, DRSpaces have been used to identify error-prone regions, predict defects, and detect modularity violations [25] [8]. For example, modularity violations can be detected when two files have a high degree of co-changes despite not being structurally connected, indicating that they change together, even though they are not architecturally designed to do so.

2.5 Code Regions and Data-Flow Interactions

Data-flow analysis. Data-flow analysis is a static analysis technique that is used to compute dynamic properties of a program without executing it. It takes a representation of a program as input and uses data-dependencies and flow information to compute properties for each program component. This representation is usually a set of functions, each represented and analyzed as an intra-procedural control-flow graph (CFG).

A CFG represents a function as a directed graph, where nodes correspond to basic blocks—sequences of instructions with a single entry and single exit—and edges represent control flow, indicating how execution may proceed from one block to another.

Data-flow analysis computes abstract properties of a program—such as reaching definitions, live variables, or available expressions—at various program points, typically before

and after each instruction in the CFG. These properties represent the program's behavior and are encoded as elements in an abstract domain, often defined as a finite-height lattice.

To analyze a program, the analysis uses transfer functions to describe how each instruction affects the tracked properties and merge operations to combine information from multiple control-flow paths. The analysis starts with an initial state and propagates abstract values along the CFG edges by repeatedly applying these functions to each node. This process continues iteratively, updating the abstract state at each node, until a fix point is reached, that is, the properties stabilize and no further changes occur.

Code Regions. To perform data-flow analysis in practice, the source code is first lowered to an Intermediate Representation (IR). This step is typically performed by the compiler before translating the source code into machine code. IR abstracts away from the specifics of both the source and target languages while retaining enough structural information to enable precise program analysis. It is usually in Static Single Assignment (SSA) form, where each variable is assigned exactly once, and individual instructions closely map to underlying program operations.

Code regions are defined on top of the IR. They consist of groups of IR instructions that are grouped based on semantic tags. These tags describe the origin or logical role of the region (e.g., specific file, function, or loop), allowing related instructions to be treated as a coherent unit. They essentially abstract over low-level IR instruction by organizing them into higher-level program constructs, enabling more modular and reusable analyzes.

Code regions are essential for our study, as they allow groups of IR instructions to be associated with higher-level entities such as files. This allows the data-flow analysis to be lifted from low-level instructions to the file level.

Data-flow interactions. An important aspect of understanding software architecture is analyzing which components within a software system interact with each other. These interactions reveal how elements are related in practice and help identify dependencies between components of the system (e.g. between files).

Data-flow interactions are interactions where data, produced by one part of the program, is actually used in another. To analyze these interactions, we can infer interactions between code parts by computing interactions between their code regions. If code region A produces data that are used by code region B, then B depends on A for those data. This relationship can be described as a data-flow dependency and is modeled as a data-flow interaction between code regions A and B.

To compute interactions between code regions, an important concept called interaction relations between code regions was introduced by Sattler et al. [23].
The data-flow interaction relation ($\rightsquigarrow$) between two code regions r_1 and r_2 is defined as:

Definition 1. The data-flow interaction relation $r_1 \rightsquigarrow r_2$ evaluates to true if the data produced by at least one instruction i that is part of r_1 flows as input to an instruction i' that is part of r_2.

$$r_1 \rightsquigarrow r_2 \;=\; \exists i \in r_1 \; \exists i' \in r_2 \; \mathrm{DF}(i, i')$$

This interaction relation serves as the basis for the interaction analysis framework developed by Sattler et al. [23].

VaRA. Their tool, VaRA, is built on top of LLVM and PhASAR and is capable of computing data-flow interactions across commit regions. In their novel approach SEAL, VaRA uses data-flow analysis to infer interactions between commits by identifying how changes in one commit region propagate to others through data-flow. This process is based on the interaction relation concept they defined, which is used to compute structural and data-flow interactions between commit regions.

Additionally, VaRA can be used to extract file-level data-flow dependencies. These dependencies are inferred by applying the interaction relation concept to code regions that are tagged by file. The resulting data-flow interactions between files form the foundation of our study and are essential to our analysis.

The description above is a simplification of the core concept of their method. Therefore, we recommend referring to the original publication for a complete and detailed explanation.

3

Methodology

In this chapter, we outline the methodology of our study. Firstly, we introduce the research questions that we established to investigate data-flow-based DRSpaces. Next, we detail the analysis approach. Finally, we describe how we select the projects, obtain and process the dependency data, and build DRSpaces.

3.1 Research Questions

The main goal of the thesis is to explore the use of DRSpaces with data-flow. Specifically, we want to assess whether data-flow reveals different architectural structures when used to construct DRSpaces compared to static relations and co-changes. To achieve this, we formulate three research questions.

> **RQ1** What are the structural characteristics of DRSpaces constructed with data-flow?

The purpose of this research question is to explore the structural properties of DRSpaces when constructed using data-flow as the primary relation. We want to identify recurring patterns in the hierarchical structure of these DRSpaces. This includes the number of leading files, the number, depth and composition of layers, the emergence and size of modules, and the overall complexity of the clustering structure. We also construct DRSpaces using static relations and co-changes, and compare the resulting structures both quantitatively and qualitatively to understand how data-flow-based DRSpaces differ from the others.

> **RQ2** To what extent do DRSpaces constructed using data-flow align with the static relations and co-changes?

To address this research question, we construct DRSpaces using data-flow as the primary relation. We then introduce static relations and co-changes as secondary relations to analyze their alignment. We consider two key aspects:

(1) We want to determine whether the compared dependencies violate the architectural boundaries established by the data-flow DRSpaces, particularly whether they introduce cross-layer or cross-module dependencies between files that break the hierarchical structure.

(2) We want to determine which relations align the most closely with data-flow-based DRSpaces and therefore complement them, and which dependencies diverge from them and therefore conflict with their structure.

> **RQ3** To what extent do DRSpaces constructed using the static relations and co-changes align with data-flow?

This is the converse of RQ2 and evaluates whether DRSpaces constructed with static relations and co-change as primary relation are supported or contradicted by data-flow when it is included as a secondary relation.

RQ3 also indicates whether components that are structurally decoupled avoid strong data-flow connections or whether hidden couplings exist that violate the intended design.

3.2 Operationalization

Relation Type	Source of the dependency → Target of the dependency
`Call`	Function call site → Function declaration site
`Cast`	Expression performing a cast → Cast type's declaration site
`Contain`	Struct/Type containing a field → Field's type declaration site
`Implementation Link`	Function call site → Function implementation site
`Implement`	Function implementation site → Function declaration site
`Import`	File inclusion site → Included header file
`Parameter`	Function using a parameter → Parameter type's declaration site
`Return`	Function return type → Return type's declaration site
`Use`	Symbol reference site → Symbol declaration site

Table 3.1: Static relations provided by the tool DV8. The *Relation Type* column indicates the semantic category of the relation, while the dependency column shows how DV8 defines the dependencies between files. The source of the dependency is the file where the symbol is referenced. The target of the dependency is the file where the symbol is declared. The source of the dependency creates the dependency and thus the file in the source is structurally dependent on the file in the target

RQ1

To answer RQ1, we analyze the structural characteristics of DRSpaces constructed using data-flow as the primary relation. Our goal is to identify unique patterns and structural properties that may emerge specifically from data-flow as a relation. The analysis is conducted in three steps:

(1) For each selected project, we construct a DRSpace using data-flow dependencies between files as the primary relation. In addition, we construct DRSpaces using each type of structural dependency between files depicted in Table 3.1 as well as co-change. This allows for a comparative analysis to determine whether any observed patterns in the data-flow DRSpace are unique or also present in the other DRSpaces. This step only requires primary relations, so no secondary relations are introduced.

(2) We compare the DRSpaces across several structural characteristics, including: the number of leading files (upper layer files), the number of layers, and the number of files in the DRSpace. In addition, we analyze the *maximum hierarchy depth* of each DRSpace, defined as the maximum number of recursively defined modules within any layer of the DRSpace. This metric captures the maximum depth of recursive nesting in a DRSpace. A depth of one indicates modules that are directly contained within a top-level layer, while higher depth values arise only when modules are further decomposed into nested sub-modules across multiple levels.

(3) Finally, we perform a qualitative analysis of the DRSpace structure. This includes examining the hiearchical structure of the the data-flow DRSpace compared to those in the other DRSpaces. We also interpret whether their structural differences reflect meaningful architectural insights.

RQ2 and RQ3

To address RQ2 and RQ3, we analyze DRSpaces constructed with different primary relations and evaluate them using secondary relations. Our goal is to determine the extent to which structural dependency types and co-changes align with data-flow in the structure of their DRSpaces and vice versa. Our evaluation consists of both quantitative and qualitative analyses, with quantitative analysis being the main focus.

For RQ2, we construct DRSpaces using data-flow as the primary relation. We then introduce each structural dependency type in Table 3.1, and co-change, individually as a secondary relation to assess how well these dependencies align with the hierarchical structure of the data-flow-based DRSpace.

For RQ3, we reverse the process: we construct DRSpaces using each type of dependency in Table 3.1, as well as co-change, as the primary relation. We then assess the alignment using data-flow as the secondary relation.

To assess the alignment between primary and secondary relations, we define the concept of a *hierarchy violation* as follows: a hierarchy violation occurs when the hierarchical structure of a DRSpace is violated.

This happens when a secondary relation introduces a dependency from a file in a higher layer to a file in a lower layer or between modules in the same layer that are not meant to be interdependent according to the hierarchical structure of the DRSpace. Since DRSpaces organize the files into a hierarchy such that dependencies are allowed only from the files in the lower layers to the files in the higher layers. The same holds also for the recursively defined modules within the layer itself.

To quantify hierarchy violations, we define two metrics:

Definition 2. Given a DRSpace d and a secondary relation r, the *violation count* is the number of files in d that have at least one hierarchy violation introduced by r. We denote this as:

$$vc(d, r)$$

Definition 3. Let m be the total number of files in a DRSpace d, and $vc(d, r)$ the violation count. The *violation ratio* is defined as:

$$vr(d, r) = \frac{vc(d, r)}{m}$$

For each secondary relation, we apply these two metrics to assess its degree of alignment with the DRSpace structure defined by the primary relation.

In addition to the quantitative evaluation, we perform a qualitative analysis in cases of extreme (dis)agreement. Specifically, we analyze cases where:
(1) The violation ratio is very high (indicating strong disagreement between primary and secondary relations), or
(2) The violation ratio is very low (indicating strong alignment between relations).
In such cases, we perform a qualitative analysis to understand the architectural or design-related reasons for these patterns.

3.3 Analysis Pipeline

Figure 3.1 depicts the analysis pipeline used in this study.

We first select the projects as explained in Section 3.3.1. Each project is then analyzed by the tools VaRA and DV8 to extract data-flow, structural, and co-change dependencies between files. The two tools use different conventions in their analysis of how they define the dependencies between files. Therefore, the data-flow output of VaRA must be normalized to follow the convention used by DV8 as explained in Section 3.3.2. From the combined dependency set, we construct analysis-ready DRSpaces as detailed in Section 3.3.3.

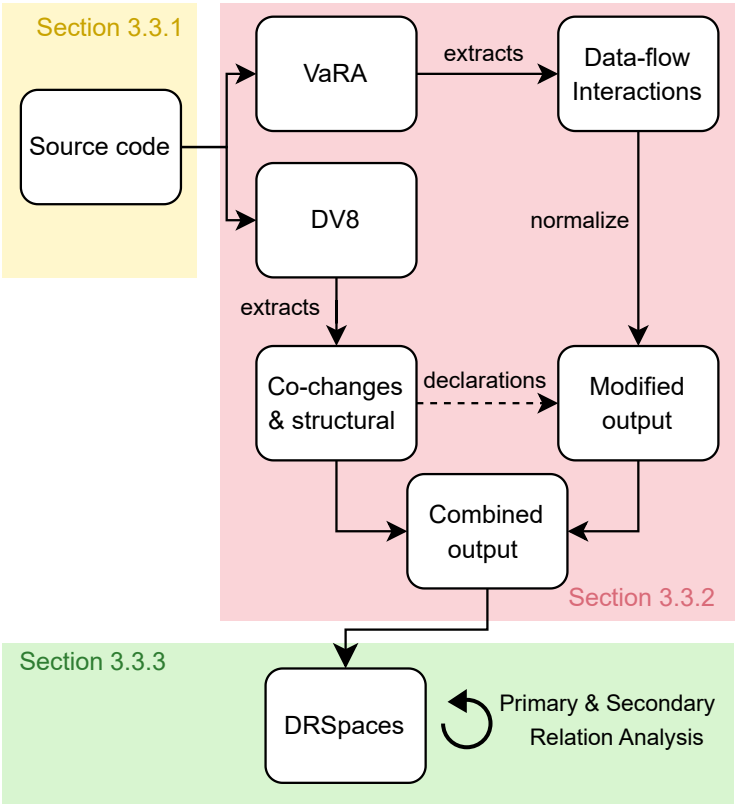

Figure 3.1: Overview of our analysis pipline

3.3.1 Project Selection

We analyze open-source software projects that are written in C/C++. For RQ1, we construct and analyze eleven DRSpaces for each project to cover the static relations, co-changes, and data-flow. The analysis requires qualitative inspection of the DRSpace structure to interpret architectural characteristics and identify recurring patterns. To keep this analysis manageable while still ensuring sufficient diversity among the analyzed systems, we limit the study to five projects from Table 3.2. For RQ2 and RQ3, all the projects listed in Table 3.2 will be analyzed.

To select the projects, we focused on later-stage versions rather than early development releases. This ensures that the systems provide sufficient revision history for the co-change analysis. In addition, we chose projects based on their file structure such that they do not have flat file structures to ensure that our file level analysis yields meaningful results. We also restricted our selection to medium-sized projects, ensuring that both preprocessing and analysis remain computationally manageable.

Project	Language	Category	Commit	SLOC	History
`htop`	C	UNIX utility	4102862	35.6 k	Mar 2006 – Oct 2024
`libvpx`	C	Video codec library	027bbee	315.6 k	May 2010 – Mar 2025
`opus`	C	Audio codec library	b5aad6a	89.1 k	Nov 2007 – Mar 2025
`toxcore`	C	Cryptographic library	1d4cc78	82.5 k	Jun 2013 – Mar 2025
`tig`	C	Git repository browser	e8e6756	39.7 k	Apr 2006 – Jun 2025
`lz4`	C	Compression library	2bc386d	24.4 k	Apr 2011 – Jun 2025
`libssh`	C	SSH library	ac6d2fa	102.7 k	Jul 2007 – Aug 2022

Table 3.2: List of projects analyzed in our study

3.3.2 Data Preparation

As part of this study, we perform data-flow analysis, co-change analysis, and static structural analysis on all projects. Together, these analyzes provide all the dependencies between files needed for constructing and comparing DRSpaces.

To extract data-flow dependencies, we use the VaRA tool developed by Sattler et al. [23], as introduced earlier in Section 2.5. For static dependencies and co-change analysis, we use the DV8 tool developed by Cai and Kazman [5]. DV8 supports DSM visualization, extraction of structural dependencies between files, co-change analysis from revision history, and DRH clustering based on a selected relation.

Step 1: Data Extraction

Data-flow. VaRA's architecture taint analysis produces a taint report that captures all data-flow interactions between files and functions. The report includes the number of flows

from each function to a target function of that data-flow, as well as the files in which these functions are implemented. These interactions are interpreted as data-flow dependencies between files. Files that receive data are considered to be the source of the dependency, while files that produce the data are considered to be the target of the dependency. This means that files that receive the data create the dependency, and thus are dependent on files that produce the data. These file-level dependencies form the basis for constructing the data-flow DRSpaces used in this study.

Co-changes and structural. DV8 performs both the co-change and structural analysis simultaneously on the provided repository. It outputs a dependency file that contains all structural dependencies between files, as well as the number of co-changes between file pairs. In this study, we use all types of structural dependency supported by DV8, as listed in Table 3.1.

Step 2: Data Normalization

The output of the tools used in this study differ in both format and dependency conventions. The first step is therefore to normalize the data into a single convention such that the two dependency outputs can be merged into one dependency file later.

The key difference is in how each tool defines the target of a dependency. DV8 always records a dependency from a symbol reference to the *declaration* site of the referenced symbol, not to its implementation.

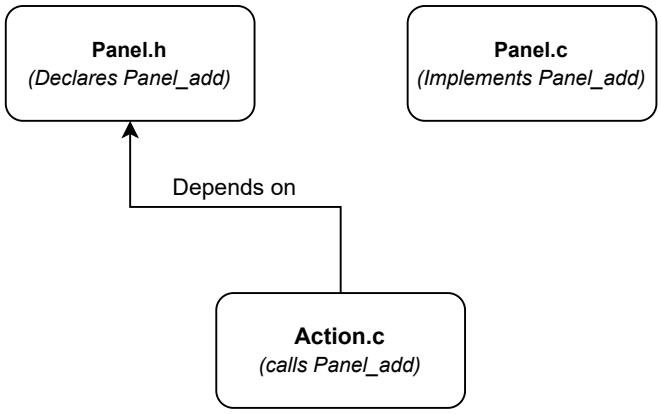

Figure 3.2: DV8 example of Call dependency from the htop project

As shown in Figure 3.2, a call from the file `Action.c` to the function `Panel_add` results in a dependency from the file `Action.c` to the file `Panel.h`, because the function is declared in `Panel.h`. DV8 treats the symbol declaration as the dependency target. And therefore the file in which the function was implemented, in this case `Panel.c`, is not included as the dependency target.

This behavior applies to all static relations listed in Table 3.1, except for the *implementation-link* relation. For implementation-link, the dependency target is the source file in which the function is implemented rather than where it is declared. The purpose of this relation is to show which concrete implementation is invoked when a function call occurs. As a

result, implementation-link dependencies point to the implementing source file instead of the declaration.

Figure 3.3 shows how the tool VaRA emits data-flow dependencies between files and

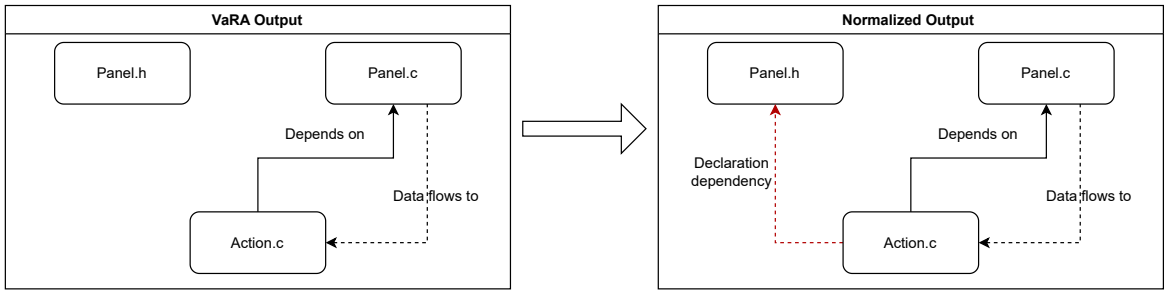

Figure 3.3: VaRA output before and after normalization

how these dependencies must be normalized. In VaRA's output, the target of a data-flow dependency is the file in which the referenced symbol is implemented or imported. Thus, when data flows from one file to another, the dependency always targets the source file rather than the corresponding header file. In contrast to the static dependency shown in Figure 3.2, this means that the file `Panel.c` is the target of the dependency instead of the file `Panel.h`, while the source of the dependency remains the file `Action.c`.

To normalize the data-flow output, we introduce an additional edge for each data-flow dependency. The original edge targeting the implementation file is preserved, and a second edge is added that targets the file in which the referenced symbol is declared. This aligns the data-flow output with the declaration-based convention used by DV8, and enables direct comparison between data-flow and structural dependencies, including implementation-link, at the file level.

With this normalization, a data-flow interaction between files through a symbol may induce dependencies to both the implementation file and the declaration file of that symbol. This applies to symbols that are declared in one file and implemented in another, which is common in C/C++ systems. However, some symbols do not have this property. For example, static and private functions are both declared and implemented in the same file and therefore result in only a single dependency edge to that file.

Step 2.1: Resolving Function Definitions

We first resolve the function references present in the data-flow output. To map data-flow dependencies to the corresponding declaration files, we use the Implement relation (IL) extracted by the tool DV8, as shown in Table 3.1, during structural analysis. This relation maps each function implementation to the file in which the function is declared. Using this mapping, we locate the declaration file associated with each function referenced in the data-flow output and introduce the corresponding edges.

To perform this step, we first build a lookup table by extracting all occurrences of IL, mapping each function implementation and its implementation file to the associated declaration file. We then add an additional declaration edge for every data-flow dependency

that targets a function implementation by looking up the corresponding declaration file in the table.

Although IL is extensive and captures most function declarations, it is not complete. Some function declarations are not detected by DV8's structural analysis. We handle these cases in the following steps.

Coverage. We tested the coverage of our approach on a set of projects. For this, we extracted all symbols that appear as data-flow dependency targets in the VaRA output and attempted to resolve them to their declarations using IL. Table 3.3 summarizes the results, listing each project we tested.

Project	#Symbol	#Resolved	#Unresolved	#Unresolved Files
`htop`	120	45	75	24
`libvpx`	54	22	32	5
`opus`	110	76	34	14
`toxcore`	869	496	373	66

Table 3.3: Coverage of symbol resolution in the data-flow output after Step 2.1. *Symbol* denotes the total number of distinct symbols appearing as data-flow dependency targets. *Resolved* and *Unresolved* indicate how many of these symbols could or could not be mapped to declaration files using the Implement relation, while *Unresolved File* reports the number of distinct files containing unresolved symbols.

Step 2.2: Resolving Static and Private Functions

Static and private functions within the unresolved symbol list require special handling. These functions have both their implementation and declaration inside the same source file that appears in the data-flow output. Therefore, each such function should have only one dependency edge to that file.

To correctly classify static and private functions, we perform a regex-based search and build a lookup table for all static and private functions in each file that appears in the unresolved file list in Table 3.3. We then look up all unresolved symbols and assign an edge to a file if the symbol is identified in the lookup table.

Additionally, we construct a lookup table for all functions implemented in these files that are not static or private, also using a regex-based search. This table allows us to capture functions whose declarations exist in a different file but were missed by the structural analysis of DV8 and thus were not resolved in Step 2.1.

Updated Coverage. Table 3.4 presents the updated coverage results after handling static and private functions. Compared to Table 3.3, this table distinguishes the number of unresolved functions whose declarations exist but were not captured by IL. These cases are reported in parentheses in the *Unresolved* column. The results show that resolving static and private functions significantly reduces the number of unresolved symbols and the number of distinct files.

Project	#Symbol	#Resolved	#Unresolved (Function)	#Unresolved File
htop	120	79	41 (27)	8
libvpx	54	41	13 (4)	1
opus	110	97	13 (2)	8
toxcore	869	864	5 (5)	1

Table 3.4: Coverage after handling static and private functions

Step 2.3: Manual Resolution of Remaining Definitions

The remaining unresolved symbols consist mainly of static inline functions, macros, or global variables imported into a source file, as well as functions whose declarations were not captured by IL. All of these cases require a second edge to the file in which the symbol is actually declared.

We resolve these symbols manually by inspecting the source code and identifying their declaration files. This is manageable because Step 2.1 and Step 2.2 significantly reduce both the number of unresolved symbols and the number of files in which they appear.

Step 3: Data Merging

After normalizing the data-flow dependencies, we merge the normalized output of the tool VaRA with the output of the structural and co-change analysis of the tool DV8 into a single combined dependency file that conforms to the DV8 dependency input format.

3.3.3 Constructing Design Rule Spaces

After extracting, normalizing, and merging the dependencies, we construct the DRSpaces for the projects using the tool DV8.

To build a DRSpace, DV8 first analyzes the combined dependency file of a project. The project is then visualized in a DSM showing all files and the dependencies between them. We then select a dependency type to serve as the primary relation and apply the DRH clustering algorithm provided by DV8. This produces a layered DRSpace that reflects the hierarchical structure imposed by the chosen dependency type.

After the DRSpace is constructed, we can introduce additional dependency types as secondary relations. These secondary relations are then used to analyze structural alignment with the hierarchical structure, modularity, and cross-layer violations.

4

Evaluation

In this chapter, we present our results and answer the research questions.

We first present the results for RQ1, followed by the results for RQ2 and RQ3. We then discuss the findings in the context of our research questions and conclude with the threats to validity associated with our evaluation.

4.1 Results

Before presenting the results, we clarify two aspects of how the DRSpaces are interpreted in this chapter.

First, we refer to the recursive layers within each top-level layer produced by the DRH clustering algorithm as *modules*. In our analysis, we consider only modules at depth one, that is, modules directly below each top-level layer of the DRSpace. This follows after inspecting the structure of 77 DRSpaces. We observed that most DRSpaces have a well-structured hierarchy with recursive modules in the depth one. We therefore restrict our analysis to modules at this depth to ensure the consistency of our results, specifically in RQ2 and RQ3, where we use the hierarchal structure to analyze the alignment of relations.

Second, for co-change data, we consider only file pairs with at least five co-changes. Because we analyze the full revision history of each project, co-changes below five are treated as noise. We selected this threshold by experimenting with different thresholds and observing their effect on the co-change DRSpace of multiple projects. A threshold of five did not affect the structural layout for most of the projects, therefore, values below five can be considered unimportant.

4.1.1 Structural Characteristics (RQ1)

With this research question, we examine whether data-flow DRSpaces exhibit structural properties that differ from DRSpaces constructed using other relation types. To do this, we construct DRSpaces for each static relation and for co-change, and compare them with the data-flow DRSpaces. We also qualitatively analyze the structure of the data-flow DRSpaces and draw conclusions about their implications.

Across all analyzed projects, DRSpaces constructed from static relations exhibit largely similar structural characteristics. These DRSpaces consistently show a well-structured hierarchy with clearly separated layers and relatively similar leading files. In contrast, the data-flow DRSpaces showed more variation between projects.

To explore these variations in the structure of DRSpaces, we analyze the *maximum hierarchy depth* of each DRSpace for each project, as defined in Section 3.2.

Through inspection of all analyzed DRSpaces, we observed that DRSpaces with a well-defined hierarchical structure consistently have low maximum hierarchy depth. Whereas DRSpaces that lack a clear hierarchy and show more complex structures have higher depth values. Based on this observation, we use the maximum hierarchy depth as an indicator of how well-defined or complex the hierarchical structure of a DRSpace is.

We also compute the *dependency density* of each DRSpace following the definition used in [25]. Let d be a DRSpace. Let E_d denote the set of dependencies between files in d, and let $|d|$ be the number of files in d. We define the dependency density of d as

$$\text{dens}(d) \;=\; \frac{|E_d|}{|d|^2}. \tag{4.1}$$

The higher the density, the more tightly coupled the files within the DRSpace.

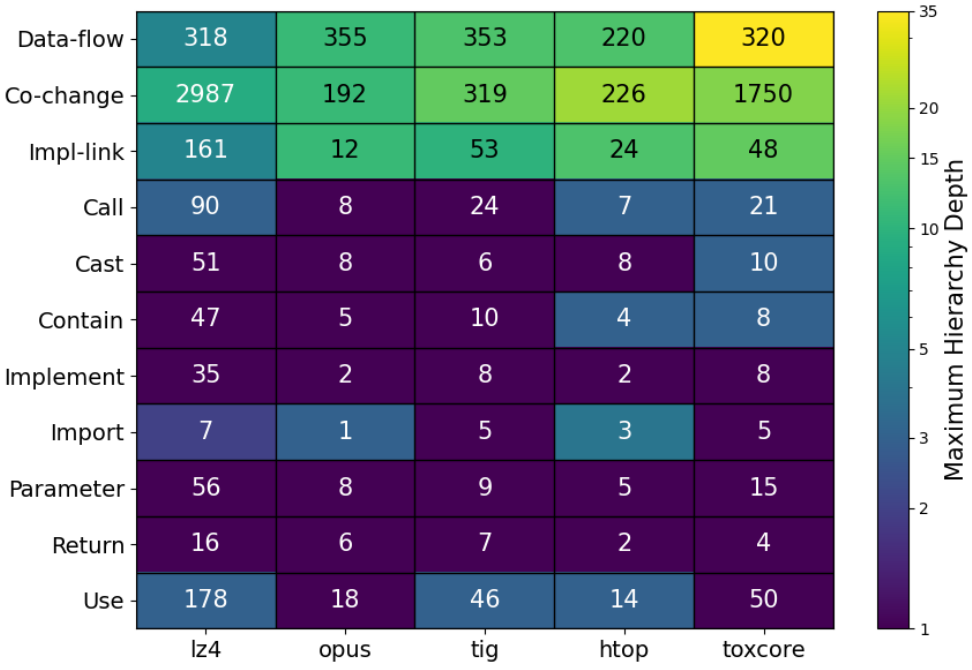

Figure 4.1: Structural complexity of DRSpaces across projects and relations. Rows correspond to dependency relations and columns to projects. Each cell represents the DRSpace d constructed for the respective relation–project pair. Cell color encodes maximum hierarchy depth, as indicated by the color scale. The numeric value inside each cell denotes the dependency density $dens(d)$ expressed as a percentage.

Figure 4.1 shows that data-flow DRSpaces generally exhibit higher structural complexity than DRSpaces constructed from static relations. Across most projects, data-flow DRSpaces have higher maximum hierarchy depth than static-relation DRSpaces. The only exception is the project `lz4`, where the data-flow DRSpace has a maximum hierarchy depth of 4, which is comparable to that of static-relation DRSpaces. The data-flow DRSpaces of `tig`, `htop`, and `toxcore` have high maximum hierarchy depths, and also appear visually more

complex with a poor hierarchy structure. This is most visible in the project `toxcore`, where the data-flow DRSpace shows the highest depth among all analyzed DRSpaces, reflecting a highly entangled structure with no hierarchy. In contrast, although the data-flow DRSpace of `opus` also shows a relatively high maximum hierarchy depth, its structure remains largely well-defined. Only a single module in the second layer of its data-flow DRSpace exhibits deeper recursive nesting, while the remainder of the DRSpace maintains a clear hierarchical organization.

In comparison, DRSpaces constructed from static relations consistently show well-defined hierarchies and modular structures in all projects. In the majority of the DRSpaces, static relations have a maximum hierarchy depth of one. Some deviations occur, for example, in the *Call* DRSpace of the projects `lz4`, `htop`, and `toxcore`, but even in these cases the resulting hierarchies remain well structured. Among static relations, *Implementation-link* stands out by having higher maximum depth values and more complex structures that are more comparable to the data-flow DRSpaces. Among all DRSpaces, the co-change DRSpaces have the highest maximum hierarchy depth values overall, with the exception of the project `toxcore`, where data-flow exceeds co-change in maximum hierarchy depth.

Similar to data-flow, the DRSpaces of implementation-link and co-change have a modular structure in `lz4` and `opus`, and a complex structure with no clear hierarchy in `tig`, `htop`, and `toxcore`.

The dependency density further highlights these differences. Data-flow DRSpaces consistently show higher dependency density than static relations, ranging between 220% and 355%, with `htop` exhibiting the lowest and `opus` the highest value. The DRSpaces constructed from implementation-link also have higher dependency density compared to other static relations, particularly in the projects `lz4`, `tig` and `htop`. In contrast, the remaining static relations generally have much lower dependency densities, mostly below 25% with some exceptions—most notably in `lz4`, where static relations exhibit higher density values. Co-change DRSpaces show the highest dependency densities overall, reaching 1750% and 2987% in `toxcore` and `lz4` respectively.

Overall, Figure 4.1 shows that DRSpaces with a high maximum hierarchy depth, such as data-flow and co-change DRSpaces, often coincide with a high dependency density, although there are exceptions. For example, the implementation-link DRSpace of `lz4` has a high dependency density while still exhibiting a well-structured hierarchy. Similarly, the data-flow DRSpace of `opus` has a high dependency density but its structure is modular. These exceptions indicate that the dependency density alone does not fully explain the structural complexity of DRSpaces.

To further explore the characteristics of data-flow DRSpaces, we analyze their structural properties.

Table 4.1 highlights clear structural differences among the data-flow DRSpaces of the analyzed projects. The data-flow DRSpaces of projects such as `lz4` and `opus`, which exhibit well-defined hierarchical structures, also show a larger number of top-level layers and very few leading files. This indicates that the DRH algorithm was able to identify clear architectural layers and a small set of dominant files based on the data-flow dependencies between files in these systems.

In contrast, the data-flow DRSpaces of `tig`, `htop`, and `toxcore` consist of only two top-level layers. In each case, most files appear in the first layer and are therefore classified as

Project	#Layers	#Leading Files	#DRSpace Files	#Total Files
`lz4`	4	1	17	26
`opus`	3	5	67	225
`tig`	2	42	44	88
`htop`	2	55	57	180
`toxcore`	2	119	126	150

Table 4.1: Structural properties of Data-flow DRSpaces. *Layers* denotes the number of top-level layers in the DRSpace. *Leading Files* denotes the number of files in the uppermost layer. *DRSpace Files* corresponds to the number of files included in the DRSpace. *Total Files* refers to the number of source-code files in the project after excluding external library code (including vendored sources) and all test files.

leading files, while the second layer is very small (2–7 files). Consequently, the clustering yields little layer separation and a shallow hierarchy for these projects.

In addition, the results show that in `opus`, `tig`, and `htop`, data-flow dependencies are concentrated in a small subset of files. In these projects, the data-flow DRSpaces capture approximately 30%, 50%, and 32% of all files in the system, respectively. This indicates that only a portion of the system's files participate in data-flow interactions.
For comparison, the structural characteristics of DRSpaces constructed from static relations and co-changes are reported separately in Appendix A.

Finally, the comparatively high dependency densities observed for `lz4`, despite its DRSpaces having well-structured hierarchies, are explained by the small size of the project and its DRSpaces. As shown in Figure 4.1, the DRSpaces of `lz4` have higher densities across static relations and co-change compared to the other projects. However, the corresponding DRSpaces contain only a small number of files (e.g., 17 files for co-changes and 12 files for implementation-link). Because the dependency density is normalized by $|d|^2$ (Equation 4.1), small DRSpaces can yield high density values even when their hierarchical structure is well-defined.

4.1.2 Relations Alignment (RQ2 and RQ3)

With these research questions, we explore the use of secondary relations in the analysis of DRSpaces. For RQ2, we analyze all relations as secondary relations within the data-flow DRSpace. For RQ3, we reverse this by analyzing data-flow as a secondary relation within DRSpaces constructed from the other relations. The goal is to explore what additional insights this bidirectional analysis provides and how the relations align.

To analyze the alignment, we use the *violation count vc* and the *violation ratio vr* as defined in Definitions 2 and 3 in Section 3.2, respectively. The violation count is the number of files in a DRSpace that have at least one hierarchy violation after introducing a secondary relation. A hierarchy violation occurs when a dependency contradicts the hierarchical structure imposed by the primary relation, for example, when a file in an upper layer depends on a

file in a lower layer. The violation ratio normalizes the violation count by the size of the DRSpace, allowing comparison of alignment across relations and projects.

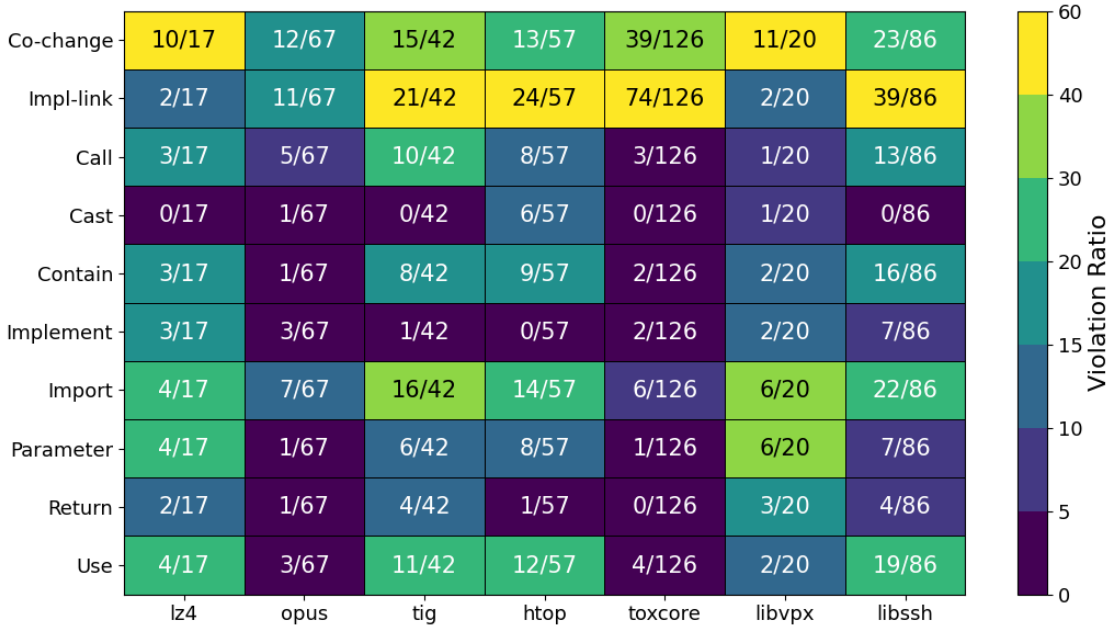

Figure 4.2: Alignment of secondary relations with data-flow-based DRSpaces. Each row corresponds to a secondary relation r and each column to a project. Each cell represents the data-flow DRSpace d constructed for the respective project. Values in the cells report the violation count $vc(d,r)$ over the size of the DRSpace $|d|$. The cell color encodes the violation ratio $vr(d,r) = \frac{vc(d,r)}{|d|}$ using discrete bins, as shown in the color scale.

Figure 4.2 shows the alignment between secondary relations and data-flow DRSpaces, measured using the violation count and the violation ratio. Based on our inspection of the data-flow DRSpaces, we distinguish projects with DRSpaces that have a clear hierarchal structure (e.g., `lz4`, `opus` and `libvpx`), from projects whose data-flow DRSpaces are structurally complex and with no clear hierarchy (e.g., `tig`, `htop`, `toxcore` and `libssh`). This structural difference affects how informative the violations are in this direction of the analysis.

A key example of this effect is provided by the projects `opus` and `toxcore`. Similar to `opus`, most secondary relations show very low violation ratios with respect to the `toxcore` data-flow DRSpace (often close to 0-5%), while implementation-link and co-changes remain high (e.g., $vr(d, \textit{impl-link}) \approx 59\%$, $vr(d, \textit{co-change}) \approx 31\%$). However, unlike `opus`, where low violation ratios reflect true alignment within a well-structured data-flow DRSpace, the low ratios observed for many relations in `toxcore` do not necessarily indicate true alignment. This is because we only count violations between top-level layers and modules of depth one. In highly entangled DRSpaces with no hierarchical structure such as the data-flow DRSpace of `toxcore`, violations may be hidden by the complexity of the structure. Therefore, the low violation ratios in `toxcore` are largely a result of structural complexity rather than evidence of alignment.

We also observe that the project `lz4` has a relatively low violation count across static relations, mostly between 2 and 4 violating files, whereas the violation count is much higher for the co-change relation. Despite these low counts, the violation ratios in `lz4` are comparatively high. This is mainly an effect of the DRSpace size. The violation ratio *vr* normalizes the violation count by the number of files in the DRSpace. As a result, even a small number of violations can produce a large ratio when the DRSpace is small. For example, $vr(d, Import)$ is similar for `lz4` and `libssh`, but the violation counts differ: 4 violating files in `lz4` compared to 22 in `libssh`. This indicates that for small DRSpaces, the violation count provides an important context to interpret the violation ratio.

Across projects, the relation *Implementation-link* stands out among the static relations as having the strongest misalignment with the data-flow DRSpaces. This is most notable in projects with structurally complex data-flow DRSpaces, namely `tig`, `htop`, `toxcore`, and `libssh`. Co-change also shows high misalignment with data-flow compared to the remaining static relations. It consistently yields higher violation ratios, with implementation-link being the only relation that exceeds it in some cases. In contrast, implementation-link aligns comparatively well with data-flow in `lz4` and `libvpx`, where its violation ratios remain low.

In addition, the static relation *Cast* shows very low violation ratios with respect to data-flow, which can appear as high alignment. However, this result is largely explained by the sparsity of the cast dependencies. Cast is a specific static relation that introduces dependencies only when explicit type casting occurs. In projects where type casting is rarely used, very few cast dependencies are present, which leads to low violation ratios regardless of the underlying data-flow structure. In these cases, the low violation ratios reflect low relation coverage rather than a true alignment between cast and data-flow. This effect is observed in all projects except `htop`, where casting is used more frequently.

Overall, in this analysis direction, projects whose data-flow DRSpaces exhibit a clear hierarchical structure (e.g., `opus` and `libvpx`) tend to show mostly stable violation ratios across secondary relations. In these cases, low violation ratios are more likely to reflect true alignment between the relations and data-flow. In contrast, in structurally complex data-flow DRSpaces with a weak hierarchical structure (e.g., `toxcore` and `libssh`), violations can be missed because we only evaluate violations at the top-level layers and within depth-one modules, which can make some relations appear well-aligned. Moreover, violation ratios can be affected by small DRSpace sizes (e.g., `lz4`) and by low relation coverage (e.g., *Cast*), where sparse dependencies yield low ratios regardless of the underlying data-flow structure.

To complement the analysis in Figure 4.2, we now reverse the direction and analyze data-flow as a secondary relation within DRSpaces constructed from the other relations.

Figure 4.3 reports the corresponding violation ratios $vr(d, data\text{-}flow)$ for each primary relation and project. Because the DRSpaces in this direction differ in size, we compare alignment within the DRSpaces of a project using the violation ratio rather than the violation count. We also exclude *Cast* from cross-project comparison, because cast-based DRSpaces cover only a small subset of files in most projects. As a result, its violation ratios are strongly influenced by sparse dependencies rather than reflecting meaningful alignment.

Figure 4.3 reveals three broad patterns across projects. The first case is observed in the projects `opus`, `htop`, and `libvpx`, where data-flow shows comparatively low violation ratios across most primary relations. In `opus` and `htop`, the violation ratios range between 8-19%

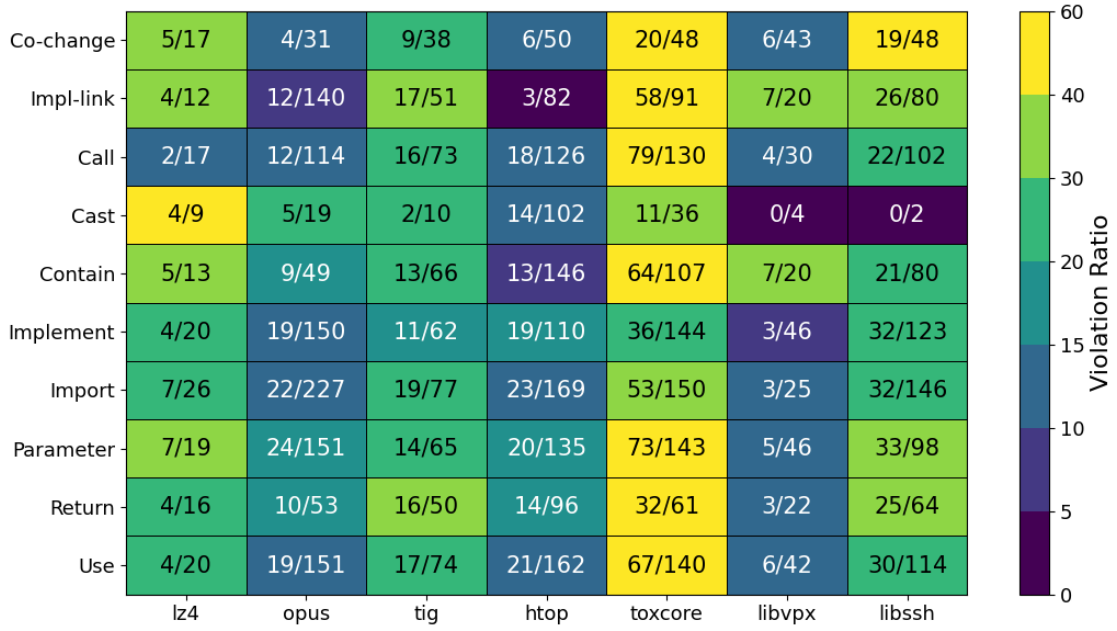

Figure 4.3: Alignment of data-flow as a secondary relation with DRSpaces constructed from other relations. Rows correspond to primary relations and columns to projects. Each cell represents the DRSpace d constructed from the primary relation. Each cell reports the violation count $vc(d, \textit{data-flow})$ over the size of the DRSpace $|d|$. The cell color encodes the violation ratio $vr(d, \textit{data-flow}) = \frac{vc(d,\textit{data-flow})}{|d|}$ using discrete bins, as shown in the scale.

and 3-18%, respectively. In `libvpx`, most relations remain below 15%, with higher values mainly for the static relations implementation-link and contain (up to 35%).

The second case is observed in the projects `lz4`, `tig`, and `libssh`, which show higher and more variable data-flow violation ratios, ranging between 11-30% in `lz4`, 17-34% in `tig`, and 21-40% in `libssh`.

The third case is the project `toxcore`, which is the most visible outlier among projects. In nearly all primary relations, the violation ratios are usually above 50%. This means that data-flow introduces hierarchy violations for more than half of the files contained in these DRSpaces, indicating that data-flow strongly crosses the modular boundaries implied by the primary relations and that the system is highly entangled with respect to data-flow. This also indicates of modularity issues in the system, since files that are separated by static structure still exchange data across those boundaries.

This result also highlights a strong asymmetry with Figure 4.2. When other relations are analyzed as secondary relations within the `toxcore` data-flow DRSpace, many of them appear to align well. In contrast, when data-flow is analyzed as the secondary relation, misalignment becomes consistently visible across almost all primary relations. This supports our interpretation that, in highly complex data-flow DRSpaces, the direction used in Figure 4.2 can hide violations and therefore overstate alignment.

We also observe low violation ratios when data-flow is used as a secondary relation in `htop`, although `htop` was found to have a very complex data-flow DRSpace structure in Section 4.1.1. Conversely, `lz4` shows comparatively high violation ratios even though the

structure of its data-flow DRSpace was highly modular. Based on these cases, we find that the complexity of the data-flow DRSpace structure has no effect on how well data-flow aligns as a secondary relation, and that this structure alone has limited meaning when interpreting modularity issues in these systems.

Across relations, data-flow alignment with co-change is the most consistent and most symmetric relative to the overall alignment trends observed for static relations across the projects and the three cases distinguished earlier. In the first case (projects `opus`, `htop`, and `libvpx`), co-change yields violation ratios between 12% and 14%. In the second case (projects `lz4`, `tig`, and `libssh`), the co-change violation ratios are higher, namely 30%, 24%, and 40%, respectively. Co-change also shows its own highest misalignment in `toxcore` with 42%. This indicates that co-change is a stable reference relation for tracking projects with modularity issues.

Overall, we observe asymmetric alignment patterns in projects with complex data-flow DRSpaces when comparing the two analysis directions. Several relations show low violation ratios when analyzed as secondary relations within the data-flow DRSpace (Figure 4.2), while data-flow shows consistently higher violation ratios when analyzed as a secondary relation within DRSpaces constructed from other relations (Figure 4.3). In addition, the results in this direction are more consistent and stable across relations than in the inverse direction.

4.2 Discussion

In this section, we discuss our results and their implications with respect to our research questions.

4.2.1 Structural Characteristics (RQ1)

The results of RQ1 show that data-flow-based DRSpaces are generally structurally more complex than DRSpaces constructed from static relations. A similar level of complexity is observed for co-change DRSpaces and, to some extent, for DRSpaces constructed from the static relation *implementation-link*. We also observe that DRSpaces with higher structural complexity often have a higher dependency density. At the same time, there are exceptional cases in which a DRSpaces remains modular despite having a high dependency density. This shows a consistent pattern, though not a deterministic rule, that higher dependency densities often lead to more complex DRSpaces hierarchies.

The high dependency densities observed in co-change DRSpaces are influenced by the fact that the co-change analysis is done over the full project history, which accumulates many dependencies. For data-flow, the results suggest that data-flow dependencies between files are generally more numerous than structural dependencies. One possible explanation is that the data-flow analysis capture many runtime-relevant interactions that are all represented as data-flow dependencies, whereas the structural dependencies are separated into many static relations and, therefore, appear more distributed.

An additional factor is our preprocessing and normalization of the data-flow output of the tool VaRA. By mapping data-flow interactions to declaration files, we increase the number of participating files by adding additional declaration-target edges for symbols whose declaration and implementation reside in different files. This increases the number of dependencies represented in the data-flow relation. However, this normalization affects only a subset of symbols, as many do not require an additional edge. Therefore, even with this factor, the observed dependency densities in the data-flow DRSpaces remain substantially higher than those of all static-relation DRSpaces. This suggests that the files are tightly coupled through data-flow dependencies in the analyzed systems.

Structurally, data-flow DRSpaces exhibit substantial variation in characteristics compared to static-relation DRSpaces, and they are often closer in structure to co-change DRSpaces. The DRSpaces model is designed to identify *independent modules* (top-level layers) and *design rules* (leading files) based on the dependency structure of the chosen primary relation. For data-flow DRSpaces, we found that in three of the five analyzed projects, the resulting DRSpaces contains only two layers, with most files placed in the upper layer and therefore classified as leading files. This indicates that, in these systems, the data-flow dependencies between files are highly interconnected, limiting the ability of the DRH algorithm to separate files into layered modules and to identify a small set of dominant design rules with respect to data-flow. In contrast, DRSpaces constructed from static relations consistently exhibit clearer hierarchies and more analyzable structures when used as the primary relation.

In general, the findings of RQ1 show that data-flow-based DRSpaces can differ substantially in structure from DRSpaces constructed from static relations in some systems, and that their complexity is often comparable to DRSpaces constructed from co-changes between files.

4.2.2 Relations Alignment (RQ2 and RQ3)

The results of RQ2 and RQ3 show that alignment analysis using secondary relations provides additional meaningful information, but only when interpreted together with the structure of the underlying DRSpaces. A key observation is that alignment is not symmetric: analyzing relations as secondary to a data-flow DRSpace (RQ2) and analyzing data-flow as secondary to other DRSpaces (RQ3) can lead to different conclusions, particularly in projects whose data-flow DRSpace is structurally complex.

Across our research questions, the relations that yield the most structurally complex DRSpaces are data-flow, co-change, and implementation-link. These relations were also found to exhibit a higher dependency density than most other static relations (RQ1). In the alignment results, co-change and implementation-link consistently produce higher violation ratios than the remaining static relations in both analysis directions. A possible explanation is that denser relations introduce more cross-hierarchy edges and increase the likelihood of contradicting the hierarchy imposed by the primary relation. This also highlights a methodological condition for interpreting violation ratios: relations with very dense dependency may inflate violation ratios, so misalignment should be interpreted together with the individual relation characteristics. In our study, this implies that the

misalignment observed when data-flow is used as a secondary relation (RQ3) may be amplified by the inherent density of data-flow dependencies.

The combined results further show that alignment must be interpreted in relation to data-flow participation. In RQ3, data-flow shows its lowest violation ratios in `opus` and `htop`, which are also the projects where data-flow dependencies are concentrated in a smaller subset of files in RQ1 (≈30% and ≈32%). In contrast, `toxcore` shows broad data-flow participation (126/150 files) and consistently high violation ratios across most primary relations in RQ3. When data-flow is analyzed as a secondary relation, the violation ratio measures how often data-flow violates the hierarchy of a DRSpace constructed using a given primary relation, considering only the files that are present in that DRSpace. If data-flow dependencies are concentrated in a smaller subset of files, fewer files contribute data-flow edges that can cross the hierarchal boundaries, which reduces the number of observable violations in this direction. In contrast, when data-flow is spread across a large fraction of the system's files, cross-hierarchy data-flow dependencies become more frequent and the violation ratios increase.

A broader interpretation of these results is that data-flow-based DRSpaces behave structurally closer to co-change than to static relations. In RQ1, both data-flow and co-change produce DRSpaces that are often deeply nested and have fewer layers than static-relation DRSpaces. This structural similarity matters for how the relations should be used in practice: when the DRSpace itself lacks a clear hierarchy, violations introduced by some secondary relations can be difficult to observe and to interpret reliably. This effect is most visible in the project `toxcore`, where the two analysis directions produced asymmetric results because of the complexity of the data-flow DRSpace structure. This also explains why the alignment results of RQ2 (data-flow as primary) are less stable than RQ3 (data-flow as secondary). In RQ2, the data-flow DRSpaces are often structurally complex and provide fewer clear hierarchical boundaries. As a result, the measured alignment varies more across secondary relations and can be inconsistent. In RQ3, the static-relation DRSpaces are typically modular and well-separated. Consequently, data-flow violations in a system tend to be more consistent across primary relations, yielding more stable alignment patterns across projects.

Taken together, these findings support treating data-flow in the same role as co-change in prior DRSpaces work [25]: not as a primary relation to define the architectural decomposition, but as a secondary relation to test decompositions derived from structural relations. When used as a secondary relation on DRSpaces constructed from structural dependencies, data-flow can directly reveal interactions that cross otherwise well-separated modules and indicate hidden couplings in the system similar to co-changes between files. This also suggests a practical analysis workflow, which is to use static relations to define the expected boundaries, and then use data-flow to identify where runtime-relevant interactions violate those boundaries and to localize files involved in these interactions that do not share structural dependencies between them.

In summary, the results of our research questions show that data-flow reveals architectural insights that differ from static relations. However, we find that the most useful contribution of data-flow in the DRSpaces model is as a secondary relation. Compared to using data-flow as the primary relation, the secondary-relation analysis produces more interpretable and consistent signals about modularity across the studied systems.

4.3 Threats to Validity

This section outlines potential threats to the validity of our study, categorized into internal and external threats.

4.3.1 Internal Validity

Our evaluation is subject to several internal threats to validity.

First, the correctness of our analysis depends on the accuracy and completeness of the tools we use. We use VaRA to extract data-flow dependencies and DV8 to extract static dependencies, co-changes, and construct DRSpaces. Any inaccuracies or limitations in these tools may directly affect the quality of the resulting DRSpaces and thus the validity of our findings.

Another internal threat is posed by the nature of data-flow DRSpaces. Since data-flow analysis is based on static compile-time analysis, it may fail to capture certain files. This is particularly true in C/C++ systems, where some files serve purely declarative or structural roles (e.g., configuration files, interface definitions, or resource files). These files may be included by static relations but will not appear in data-flow. This issue was especially observed during data preprocessing when mapping data-flow interactions to declaration files. We tried to mitigate this threat by manually inspecting unresolved cases and resolving missing declarations. However, this manual intervention introduces the possibility of human error.

Another potential threat arises from the selection of thresholds used during preprocessing and analysis, specifically the minimum number of co-changes considered. Although this threshold was empirically evaluated and was shown not to change the overall structure of the co-change DRSpace in the analyzed projects, different threshold choices could affect the results in other systems.

Finally, our study involves qualitative analysis and interpretation, which introduces subjectivity. This particularly affects RQ1, where we relate the structural complexity of the DRSpace to its dependency density, and RQ3, where we use data-flow as a secondary relation and interpret misalignments as indications of modularity issues. To mitigate this threat, we grounded our interpretations in prior work that uses similar reasoning. Xiao et al. [25] use dependency density to interpret the hierarchical structure of DRSpaces, and prior work uses secondary relations, specifically co-change, to indicate and localize modularity issues in software systems [8], [25].

4.3.2 External Validity

Our evaluation is also affected by several external threats to validity.

The generalization of our findings is limited by the focus of our evaluation on C/C++ projects. Some relations are inherently more used and better expressed in other programming languages. This is because other programming languages often follow different paradigms, which influence architectural structure and the nature of data-flow. For example,

object-oriented languages typically make more extensive use of inheritance and polymorphism, which may affect the structure and relevance of certain DRSpaces (e.g., Implement DRSpaces). As a result, the findings of this study do not generalize to systems implemented in other languages.

In addition, the size and complexity of the projects influences the results. Architectural patterns and data-flow characteristics in small projects can differ substantially from those in larger systems. This effect was observed in the case of `lz4`, a comparatively small project that exhibited a different alignment behavior due to its size. Since most projects in our study are of medium size, the conclusions drawn may not generalize to small or very large systems. Furthermore, the computational cost of data-flow analysis and data preprocessing limited our ability to include larger projects.

Finally, the number of projects analyzed is limited. This limitation stems from the effort required for data preprocessing, the computational cost of data-flow analysis, and the qualitative analysis needed to answer our research questions. Although the selected projects provided a range of characteristics, a larger set of projects could reveal additional patterns or contradict some of the observed trends. As many conclusions in this exploratory study are supported by a small number of representative cases, further large-scale studies are necessary to confirm the findings.

5

Related Work

In this chapter, we present related work that addresses problems similar to those explored in our study.

The chapter is divided into three sections, each focused on a different area of relevant research. First, we review approaches to software architecture recovery and the methods that are used to extract architectural views from source code and their discoveries. Second, we discuss existing work in the area of data-flow analysis in the context of software architecture recovery. Finally, we examine existing research related to Design Rule Spaces and their results.

5.1 Software Architecture Recovery from Source Code

Currently, there are various approaches to software architecture recovery from source code, each presents different perspectives providing valuable information.

Bunch is a software architecture clustering method that divides software components into clusters by maximizing the modularity quality (MQ) of the dependency graph [18]. The MQ is computed on the basis of intra-cluster cohesion and inter-cluster coupling. The results have shown that Bunch produces modular decompositions similar to the designed architectures.

Andritsos et al. [2] introduced LIMBO, a hierarchical clustering technique that uses information theory concepts to organize software components based on dependency relations. The results show that LIMBO recovers meaningful architectural views, especially in large software systems.

SARIF is an automated recovery technique that combines multiple sources of information, including dependencies, code text, and folder structure, to recover the software architecture from source code [26]. Their research has shown that SARIF produces better results compared to techniques that use fewer sources of information.

The reviewed approaches demonstrate that software architecture recovery from source code can be achieved using a variety of clustering and analysis techniques. Each of these techniques is unique and provides a different perspective on software architecture.

5.2 Data-Flow Analysis in Software Architecture Recovery

In recent years, several architecture recovery approaches have been proposed that incorporate data-flow analysis, either as a primary focus or in combination with other techniques.

Design Verification through SAR is a static analysis technique to recover software architecture from legacy embedded systems [21]. The technique extracts module hierarchies by analyzing control flow and data flow dependencies. They applied it to automotive ECU software, where they identified data-flow dependencies that violate the intended architecture.

CAESAR is a context-aware architecture recovery method that uses program slicing to cluster code into meaningful components [13]. It performs slicing by analyzing both static and dynamic data flows. CAESAR produces cohesive modules and helps to detect architectural violations by incorporating domain-specific rules (e.g., architecture constraints and naming conventions) during recovery.

Multi-Model SAR is a cluster-based architecture recovery method that combines multiple dependency graphs such as call relations, data access, and file structure to extract modularity information from large systems [1]. The technique aggregates clusters from each model to improve stability and modularization quality, as shown in its evaluation.

These approaches demonstrate that data-flow analysis plays an important role in architecture recovery, particularly for detecting architectural flaws and improving modularization quality. In comparison, our work investigates the architectural implications of data-flow when used independently through a recovery approach, and examines how the insights it provides compare to those obtained from static relations and evolutionary coupling.

5.3 Design Rule Spaces

Design Rule Spaces (DRSpaces) have been investigated in several studies and applied in different contexts of architectural analysis.

First, there are many early studies that lead to the creation of DRSpaces as an architectural analysis model [6], [17], and [24]. These studies examined the modularity and structure of complex software systems and their designs based on Baldwin and Clark's design rule theory [3].

DRSpaces were introduced by Xiao et al. [25] as a model to represent software architectures based on Baldwin and Clark's design rule theory [3]. In their initial work, they evaluated DRSpaces on three large open-source projects, in which they found that error-

prone files often lead error-prone DRSpaces, and most of the files in these DRSpaces are also error-prone. They also showed that most of the error-prone files of a project can be captured by a few error-prone DRSpaces.

In an extended study, Cai et al. [8] refined the DRSpaces model and introduced an Architecture Root Detection Algorithm. They evaluated their approach in 15 open source projects of varying sizes. They showed again that a few defect-prone DRSpaces captured a large number of the system's most defect-prone files. They also found that throughout multiple project releases, these defect-prone DRSpaces persisted and contained architectural flaws that propagated defects over time and caused long-term maintenance costs.

Kazman et al. [14] investigated the use of DRSpaces to locate the architectural roots of technical debt. Their work modeled architecture as a set of overlapping DRSpaces and used this representation to identify design flaws and sources of architectural debt. By analyzing clusters of files within DRSpaces, they were able to localize architectural problems and trace how these flaws propagated maintenance issues throughout the system. Their results show that DRSpaces can effectively expose architectural flaws that contribute to long-term technical debt.

Mo et al. [20] further explored DRSpaces by studying recurring patterns in the structure of DRSpaces that occur in error-prone and change-prone regions of software systems. These patterns were referred to as *hotspot patterns*. Their study analyzed DRSpaces across nine projects and identified patterns such as Unstable Interface and Implicit Cross-Module Dependency. Their results show that files involved in these patterns had significantly higher bug frequency and change frequency than average files. They also showed that the more hotspot patterns a file participates in, the higher its likelihood of being error-prone or change-prone.

Prior work has shown the effectiveness of DRSpaces for analyzing architectural quality, detecting architecture smells, and locating technical debt.

6 Concluding Remarks

6.1 Conclusion

This thesis explores the use of data-flow interactions between files as a relation to construct Design Rule Spaces (DRSpaces) and to analyze software architecture. While prior DRSpace research has focused on static relations and co-change between files, the use of data-flow has not been previously explored. To address this, we construct and analyze data-flow-based DRSpaces and compare them with DRSpaces constructed from nine static relations and co-change across C/C++ open-source projects. We assess whether data-flow provides distinct or complementary architectural insights to static relations and co-change.

To perform this analysis, we analyzed the structural properties of the data-flow-based DRSpaces for five projects and compared these properties with those of DRSpaces constructed from other relations within the same projects. In addition, we analyzed seven projects exploiting the hierarchical nature of DRSpaces to assess how different relations align with data-flow-based DRSpaces, as well as how data-flow aligns when used as a secondary relation within DRSpaces constructed from other relations.

Our main findings show that DRSpaces constructed from static relations consistently exhibit modular, well-defined hierarchies across all analyzed systems. In contrast, the structure of data-flow DRSpaces varies substantially between systems. In some systems, they resemble the modular view implied by static relations, while in others, they reveal more entangled and complex structures. DRSpaces constructed from co-change are structurally closer to data-flow and similarly complex. However, co-change also shows a higher misalignment with data-flow in the alignment analysis. The alignment analysis further shows that the two directions of analysis can lead to different conclusions, especially in systems where the data-flow DRSpace is structurally complex. When data-flow is used as the primary relation, the resulting DRSpace lacks a clear hierarchy in some systems, which limits the reliability of the alignment analysis. In contrast, using data-flow as a secondary relation yields more stable and interpretable patterns, because DRSpaces constructed from static relations have clearer hierarchies and are less dense. Therefore, in our study, data-flow is most useful as a secondary relation for testing the hierarchy induced by the structural dependencies and for identifying where data-flow occurs between files that are not structurally dependent, which indicates modularity violations.

Overall, we conclude that data-flow provides a complementary architectural perspective in the DRSpace model. It reveals architectural properties that differ from those captured by static relations, and its most useful contribution is as a secondary relation that produces more interpretable and consistent signals about the modularity of a system.

6.2 Future Work

This thesis is an exploratory study, and our findings touch on multiple aspects that require further and broader investigation through more focused studies.

As discussed in this work, data-flow help reveal modularity issues in some systems, particularly when used as a secondary relation. In the future, systematically detecting and locating modularity violations and other quality issues using data-flow as a secondary relation in DRSpaces requires dedicated studies that focus primarily on this aspect.

Another aspect concerns the preprocessing and normalization of data-flow dependencies. Our analysis combines the output of multiple tools. We used VaRA to extract data-flow dependencies between files, and DV8 to extract static relations and co-change and to construct DRSpaces. Aligning these outputs requires additional assumptions and preprocessing steps, some of which are manual, which can affect data-flow coverage and the resulting DRSpaces structure. Future work should automate this pipeline. A promising direction is tighter tool integration, for example, by incorporating data-flow analysis into DV8 or developing a unified tool that supports analyzing both structural and data-flow dependencies and DRSpaces analysis.

Additionally, we found that project size influences the violation ratio metric used in our analysis. As a result, comparisons based on this ratio should be performed between projects of similar size. Our study focuses primarily on medium-sized projects. A similar approach that includes small- and large-scale projects could help reveal different patterns.

Finally, our study is limited by the number of the analyzed projects. We analyze seven projects and draw our conclusions based on their results. A similar study on a broader scale, with a larger set of projects, could reveal additional insights provided by the data-flow-based DRSpaces that were not captured due to the limited scope of this study.

Appendix

A.1 Design Rule Space Properties (RQ1)

This section presents the structural metrics used to analyze and compare DRSpaces constructed from different relations with the data-flow DRSpaces in RQ1. The comparison metrics are the number of layers, the number of leading files, the number of files in the DRSpace, and the total number of source-code files in the project after excluding external library code (including vendor sources) and all test files.

Among all relations, co-change (Table A.1) and implementation-link (Table A.2) DRSpaces exhibit structural characteristics that differ from the remaining relations. In contrast, the DRSpaces constructed from the other static relations show highly similar structural properties across all projects. The structural properties of the data-flow DRSpaces are reported separately in Table 4.1.

Project	#Layers	#Leading Files	#DRSpace Files	#Total Files
lz4	2	11	17	26
opus	1	30	30	225
tig	1	40	40	88
htop	1	54	54	180
toxcore	2	40	48	150

Table A.1: Structural properties of Co-change DRSpaces

Project	#Layers	#Leading Files	#DRSpace Files	#Total Files
lz4	2	4	12	26
opus	3	27	138	225
tig	5	2	53	88
htop	2	84	88	180
toxcore	3	6	91	150

Table A.2: Structural properties of Implementation-link DRSpaces

Project	#Layers	#Leading Files	#DRSpace Files	#Total Files
lz4	2	6	17	26
opus	2	27	110	225
tig	3	23	79	88
htop	3	33	135	180
toxcore	2	39	136	150

Table A.3: Structural properties of Call DRSpaces

Project	#Layers	#Leading Files	#DRSpace Files	#Total Files
lz4	2	3	9	26
opus	2	4	19	225
tig	1	10	10	88
htop	2	18	109	180
toxcore	2	9	36	150

Table A.4: Structural properties of Cast DRSpaces

Project	#Layers	#Leading Files	#DRSpace Files	#Total Files
lz4	2	5	13	26
opus	4	7	49	225
tig	4	14	71	88
htop	9	9	156	180
toxcore	5	11	107	150

Table A.5: Structural properties of Contain DRSpaces

Project	#Layers	#Leading Files	#DRSpace Files	#Total Files
lz4	1	20	20	26
opus	2	13	150	225
tig	1	66	66	88
htop	2	3	118	180
toxcore	2	6	144	150

Table A.6: Structural properties of Implement DRSpaces

Project	#Layers	#Leading Files	#DRSpace Files	#Total Files
lz4	3	11	26	26
opus	11	5	223	225
tig	8	6	88	88
htop	9	9	180	180
toxcore	16	4	150	150

Table A.7: Structural properties of Import DRSpaces

Project	#Layers	#Leading Files	#DRSpace Files	#Total Files
lz4	2	3	19	26
opus	3	7	148	225
tig	3	6	69	88
htop	8	13	144	180
toxcore	6	8	143	150

Table A.8: Structural properties of Parameter DRSpaces

Project	#Layers	#Leading Files	#DRSpace Files	#Total Files
lz4	2	1	16	26
opus	3	2	50	225
tig	3	6	55	88
htop	3	13	102	180
toxcore	2	5	61	150

Table A.9: Structural properties of Return DRSpaces

Project	#Layers	#Leading Files	#DRSpace Files	#Total Files
lz4	2	6	20	26
opus	3	35	147	225
tig	3	27	81	88
htop	4	60	173	180
toxcore	2	43	140	150

Table A.10: Structural properties of Use DRSpaces

Statement on the Usage of Generative Digital Assistants

For this thesis, the following generative digital assistants have been used: We have used ChatGPT-4o and ChatGPT-5.1 for text rewriting. We are aware of the potential dangers of using these tools and have used them sensibly with caution and with critical thinking.

These tools have been used with caution and only as a means of improving the formulation of some sentences. At no point in this thesis was text directly copied from the tool. Instead, the tool's suggested formulations of individual sentences were compared with the original text, and selective improvements regarding grammar, spelling, and word choice were incorporated into the text.

Bibliography

[1] Metin Altınışık, Hasan Sözer, and Gonca Gürsun. "Software Architecture Recovery from Multiple Dependency Models." In: *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*. SAC '24. New York, NY, USA: Association for Computing Machinery, 2024, 1185–1192. DOI: 10.1145/3605098.3635917.

[2] Periklis Andritsos, Panayiotis Tsaparas, Renée Miller, and Kenneth Sevcik. "LIMBO: Scalable clustering of categorical data." In: Mar. 2004, pp. 123–146. ISBN: 978-3-540-21200-3. DOI: 10.1007/978-3-540-24741-8_9.

[3] C. Y. Baldwin and K. B. Clark. *Design Rules, Vol. 1: The Power of Modularity.* MIT Press, 2002.

[4] Len Bass, Paul Clements, and Rick Kazman. "Software Architecture In Practice." In: Jan. 2003. ISBN: 978-0321154958.

[5] Yuanfang Cai and Rick Kazman. "DV8: automated architecture analysis tool suites." In: *Proceedings of the Second International Conference on Technical Debt*. TechDebt '19. Montreal, Quebec, Canada: IEEE Press, 2019, 53–54. DOI: 10.1109/TechDebt.2019.00015. URL: https://doi.org/10.1109/TechDebt.2019.00015.

[6] Yuanfang Cai and Kevin J. Sullivan. "Modularity Analysis of Logical Design Models." In: *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. 2006, pp. 91–102. DOI: 10.1109/ASE.2006.53.

[7] Yuanfang Cai, Hanfei Wang, Sunny Wong, and Linzhang Wang. "Leveraging design rules to improve software architecture recovery." In: *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*. QoSA '13. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2013, 133–142. ISBN: 9781450321266. DOI: 10.1145/2465478.2465480. URL: https://doi.org/10.1145/2465478.2465480.

[8] Yuanfang Cai, Lu Xiao, Rick Kazman, Ran Mo, and Qiong Feng. "Design Rule Spaces: A New Model for Representing and Analyzing Software Architecture." In: *IEEE Transactions on Software Engineering* 45.7 (2019), pp. 657–682. DOI: 10.1109/TSE.2018.2797899.

[9] Anna Corazza, Sergio Di Martino, Valerio Maggio, and Giuseppe Scanniello. "Investigating the use of lexical information for software system clustering." In: *2011 15th European Conference on Software Maintenance and Reengineering*. 2011, pp. 35–44. DOI: 10.1109/CSMR.2011.8.

[10] Ural Erdemir, Umut Tekin, and Feza Buzluca. "Object Oriented Software Clustering Based on Community Structure." In: *2011 18th Asia-Pacific Software Engineering Conference*. 2011, pp. 315–321. DOI: 10.1109/APSEC.2011.33.

[11] Neil A. Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L. Nord, and Ian Gorton. "Measure it? Manage it? Ignore it? software practitioners and technical debt." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, 50–60. ISBN: 9781450336758. DOI: 10.1145/2786805.2786848. URL: https://doi.org/10.1145/2786805.2786848.

[12] Sunny Huynh, Yuanfang Cai, and Kanwarpreet Sethi. *Design rule hierarchy and analytical decision model transformation*. Tech. rep. Drexel University, 2008.

[13] Khaled Ibrahim, Hesham Hassan, Khaled T. Wassif, and Soha Makady. "Context-Aware Expert for Software Architecture Recovery (CAESAR): An automated approach for recovering software architectures." In: *J. King Saud Univ. Comput. Inf. Sci.* 35.8 (Sept. 2023). ISSN: 1319-1578. DOI: 10.1016/j.jksuci.2023.101706. URL: https://doi.org/10.1016/j.jksuci.2023.101706.

[14] Rick Kazman, Yuanfang Cai, Ran Mo, Qiong Feng, Lu Xiao, Serge Haziyev, Volodymyr Fedak, and Andriy Shapochka. "A Case Study in Locating the Architectural Roots of Technical Debt." In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 2. 2015, pp. 179–188. DOI: 10.1109/ICSE.2015.146.

[15] Rick Kazman and Jeromy Carriere. "Playing Detective: Reconstructing Software Architecture from Available Evidence." In: *Autom. Softw. Eng.* 6 (Apr. 1999), pp. 107–138. DOI: 10.1023/A:1008781513258.

[16] Kenichi Kobayashi, Manabu Kamimura, Koki Kato, Keisuke Yano, and Akihiko Matsuo. "Feature-gathering dependency-based software clustering using Dedication and Modularity." In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. 2012, pp. 462–471. DOI: 10.1109/ICSM.2012.6405308.

[17] Alan MacCormack, John Rusnak, and Carliss Y. Baldwin. "Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code." In: *Manage. Sci.* 52.7 (July 2006), 1015–1030. ISSN: 0025-1909. DOI: 10.1287/mnsc.1060.0552. URL: https://doi.org/10.1287/mnsc.1060.0552.

[18] S. Mancoridis, B.S. Mitchell, Y. Chen, and E.R. Gansner. "Bunch: a clustering tool for the recovery and maintenance of software system structures." In: *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*. 1999, pp. 50–59. DOI: 10.1109/ICSM.1999.792498.

[19] Janardan Misra, K.M. Annervaz, Vikrant Kaulgud, Shubhashis Sengupta, and Gary Titus. "Software Clustering: Unifying Syntactic and Semantic Features." In: *2012 19th Working Conference on Reverse Engineering*. 2012, pp. 113–122. DOI: 10.1109/WCRE.2012.21.

[20] Ran Mo, Yuanfang Cai, Rick Kazman, and Lu Xiao. "Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells." In: *2015 12th Working IEEE/IFIP Conference on Software Architecture*. 2015, pp. 51–60. DOI: 10.1109/WICSA.2015.12.

[21] Oscar Molin. "Design verification through software architecture recovery : Meeting ISO 26262 requirements on software using static analysis." In: 2013. URL: https://api.semanticscholar.org/CorpusID:108200372.

[22] Kata Praditwong, Mark Harman, and Xin Yao. "Software Module Clustering as a Multi-Objective Search Problem." In: *IEEE Transactions on Software Engineering* 37.2 (2011), pp. 264–282. DOI: 10.1109/TSE.2010.26.

[23] Florian Sattler, Sebastian Böhm, Philipp Dominik Schubert, Norbert Siegmund, and Sven Apel. "SEAL: Integrating Program Analysis and Repository Mining." In: *ACM Trans. Softw. Eng. Methodol.* 32.5 (July 2023). ISSN: 1049-331X. DOI: 10.1145/3585008. URL: https://doi.org/10.1145/3585008.

[24] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. "The structure and value of modularity in software design." In: *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-9. Vienna, Austria: Association for Computing Machinery, 2001, 99–108. ISBN: 1581133901. DOI: 10.1145/503209.503224. URL: https://doi.org/10.1145/503209.503224.

[25] Lu Xiao, Yuanfang Cai, and Rick Kazman. "Design rule spaces: a new form of architecture insight." In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, 967–977. ISBN: 9781450327565. DOI: 10.1145/2568225.2568241. URL: https://doi.org/10.1145/2568225.2568241.

[26] Yiran Zhang, Zhengzi Xu, Chengwei Liu, Hongxu Chen, Jianwen Sun, Dong Qiu, and Yang Liu. "Software Architecture Recovery with Information Fusion." In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2023. San Francisco, CA, USA: Association for Computing Machinery, 2023, 1535–1547. ISBN: 9798400703270. DOI: 10.1145/3611643.3616285. URL: https://doi.org/10.1145/3611643.3616285.