Bachelor's Thesis

Comparing Architectural Models Based on Co-Changes and Data-Flow Interactions

Johannes Victor Weissmann

June 30, 2025

Advisor:
Sebastian Böhm   Chair of Software Engineering

Examiners:
Prof. Dr. Sven Apel   Chair of Software Engineering
Prof. Dr. Jan Reineke   Real-Time and Embedded Systems

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

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

Declaration of Consent

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

Technical debt poses a grave threat to the continued development of a software project, with flawed software architecture causing unnecessary maintenance effort. Numerous techniques have been developed to gain a more profound understanding of the architecture of a software system, with several of them utilizing graph clustering. Graph clustering refers to the process of grouping objects represented by vertices into partitions based on user-defined criteria and has been used with various sources such as file names, documentation and data flow in the past in order to decompose a software system into its logical components.

Frequently occurring common changes of files (”co-changes”) are an indicator of logical coupling. Silva et al. define a set of patterns which capture different co-change behaviors. In order to identify which components in the software system are responsible for causing superfluous maintenance effort, they cluster a graph representation of the revision history of a software system and apply their patterns to the cluster graph.

In our work, we explore what the data flow of a program can reveal about its architecture by applying the patterns designed by Silva et al. to a model based on data-flow interactions. For five test projects, we compute the data-flow interactions between source files and use graph clustering in order to detect the set of patterns proposed by Silva et al. We compare the co-change based model and the data-flow based model and find substantial differences both in terms of overall similarity and detected patterns.

Contents

List of Figures

List of Tables

1 Introduction

Software maintenance accounts for a substantial amount of cost associated with a software project. Researchers have estimated that more than a third of development time is spent on dealing with technical debt [2] which potentially brings about ripple effects when making changes to a component in the software system [16]. To manage technical debt, numerous metrics for its measurement and quantification have been developed and integrated into automated tools which suggest when and what to refactor . However, such tools typically focus on code-level quality metrics and do not take technical debt arising from architectural design decisions into account. In order to characterize software issues which span multiple functions, files, or classes, Garcia et al. [8] introduce the concept of architectural smells. Architectural smells describe common design solutions applied by software engineers which are functionally correct but negatively affect maintainability. Since then, the concept has been refined and extended by other researchers to include revision history and other information to precisely identify the components responsible for degrading the maintainability of a software system [18]. In particular, the occurrence and the impact of co-changing files have been studied extensively. Sas et al. [20] have been able to correlate co-changing files and architectural smells. Similarly, Le et al. [14, 15] have shown that files which are part of architectural smells are more likely to suffer from frequent changes than files not involved in architectural smells. In addition to the co-change behavior of a software system, several other architectural views have been proposed [1]. Data flow describes the usage and propagation of values within the source files of a software system and has been used to decompose software systems into substructures in the past [11]. Sattler [21] created VaRA, the Variability-aware Region Analyzer, which facilitates data-flow interaction analysis of software programs. As the information which it yields is closer to the operational semantics of a software project than its history of co-changes, it provides an interesting point of view on program decomposition.

1.1 Goal of this Thesis

Our goal is to explore the relationship between architectural smells present in a software system and its data-flow behavior. We rely on an established approach for identifying architectural smells, namely co-change patterns among files in the revision history of a software project. We investigate how similar both architectural models are in terms of software system decomposition. Additionally, we examine what differences the two models exhibit in terms of architecture smell detection. We are particularly interested in investigating which architectural smells can only be identified by using data-flow information.

1.2 Overview

The remaining portion of this thesis is structured as follows. First, we introduce the relevant background information in Chapter 2. In Chapter 3, we go over the changes which we implement in VaRA in order to conduct our experiment. Then, we explain our experiment procedure and our project selection in Chapter 4. We present and discuss our findings and answer our research questions in Chapter 5. We briefly go over related work in the realm of software architecture in Chapter 6. Finally, in Chapter 7, we summarize our results and give an outlook on potential future work.

2 Background

In this section, we briefly introduce software architecture, co-change graphs, and architectural smells. Additionally, we explain graph clustering, the LLVM compiler infrastructure and VaRA, a tool for computing data-flow interactions.

2.1 Software Architecture

Over the course of more than three decades, several definitions of what constitutes software architecture emerged. Taylor et al. define software architecture as ”the set of principal design decisions governing a system” [25]. While decisions continue being made throughout the entire development process, there is a time in the software life-cycle when the focus lies on making such principal design decisions. Bass et al. [1] put forward several structures of software architecture. In their work, they describe structures as the set of all elements making up the software in question and define three architectural structures, each of which allows the architect to approach the software system from a different point of view. *Module structures* consider the software system in question as units of implementation, i.e., pieces of code, whereas *component-and-connector structures* consider the runtime components of the system and the communication pieces among them. *Allocation structures* consider the relationship between software elements and external environments where the software is created or executed. They note that, while the structures are not independent from one another, ”often, but not always, the dominant structure is module decomposition” [1] because it usually represents the project structure.

2.2 Co-Change Based Architecture Metric

In this work, we focus on an architecture model based on co-changes [23]. Silva et al. [23] extract the architecture of a software system based on co-changes among source files. A co-change between two files occurs when they both change between the same two revisions. In order to identify logically coupled components in a software system, Silva et al. [23] first construct a graph with files connected by co-changes extracted from its revision history. Next, they divide this co-change graph into clusters using a clustering algorithm. Finally, they map the resulting clusters to the actual project structure.

Originally introduced by Beyer and Noack [3], co-change graphs are an abstract representation of a version control system, where its vertices represent software artifacts such as files, classes, functions, lines of code or documentation. The weight of an edge connecting two vertices in the graph represents how often both artifacts change together. Consequently, the more often two files change in tandem, the greater the weight of the edge becomes. Mathematically speaking, a co-change graph $G = (V, E)$ corresponding to a version control system is an undirected graph with the set of nodes V representing software artifacts and the set E containing all edges between two nodes. An edge between two nodes a and b (where $a \neq b$) exists if there is a transaction t which modifies both artifacts a and b.

Silva et al. [23] propose six co-change patterns in order to capture the clusters of co-changing files in a software system. To formally define them, Silva et al. [23] make use of the following functions. Here, C is the set of all clusters in the clustering solution, with $c \in C$, and D is the set of directories. A directory $d \in D$ and clusters c contain files.

$$focus(c, D) = \sum_{d_i \in D} touch(c, d_i) * touch(d_i, c)$$

where

$$touch(d, c) = \frac{|d \cap c|}{|c|}$$

Additionally, they use a function called *spread* which returns across how many directories a cluster is spread.

$$spread(c) = \#\text{ directories containing a file from cluster } c$$

Using the above functions, Silva et al. [23] define the following co-change patterns. In Figure 2.1, we provide visualizations for each of the patterns. Colored squares represent the files belonging to the respective co-change pattern, and the labels below the boxes represent the package structure.

- *Encapsulated* clusters only affect one directory, with changes touching all files within it. Therefore, Encapsulated clusters represent the best possible pattern of co-changes. Formally, a cluster q is categorized as Encapsulated if $focus(q) == 1.0$.
- *Well-Confined* clusters are conceptually similar to encapsulated clusters. While they only affect files in one directory, they share their directory with another cluster which means that there is at least one file within said directory which does not belong to it. Formally, a cluster q is categorized as Well-Confined if $focus(q) < 1.0 \land spread(q) == 1$.
- *Crosscutting* clusters represent the worst possible pattern of co-changes. They affect a large number of directories (four or more) but only few code files within each directory. Formally, a cluster q is Crosscutting if $spread(q) >= 4 \land focus(q) <= 0.30$.
- *Black Sheep* clusters are conceptually similar to crosscutting clusters. They affect two or three directories but only touch few code files in them. A cluster q is categorized as a Black Sheep cluster if $spread(q) > 1 \land spread(q) < 4 \land focus(q) <= 0.10$.
- *Octopus* clusters consist of a body B and a set of tentacles T. While most files are confined within the body in one directory, a small number of files belonging to the cluster resides in other directories (the tentacles). A cluster q with a body B and tentacles T is classified as a Octopus cluster if $touch(B, q) > 0.60 \land focus(T) <= 0.25 \land focus(q) > 0.30$.

- Similar to Octopus clusters, *Squid* consist of a body B and one or more tentacles T. However, Squid clusters are smaller than Octopus clusters. Formally, a cluster q consisting of a body B and tentacles T is a Squid cluster if $touch(B, q) > 0.30 \land touch(B, q) <= 0.50 \land focus(T) <= 0.25 \land focus(q) > 0.3$.

It is worth noting that said co-change patterns vary in desirability, the spread across different directories and the number of files affected. Silva et al. [23] postulate that Encapsulated clusters and Well-Confined clusters in their co-change based model are indicative of a properly modularized system, whereas Octopus and Squid clusters are associated with ripple effects and bug-fixing activities. Notably, the patterns proposed are not exhaustive. Together, they cover approximately 95.4% of the co-change clusters extracted from the software projects analyzed by Silva et al. [23]. Silva et al. [23] state that their design of the patterns aims to be precise, meaningful and to cover a large percentage of found clusters.

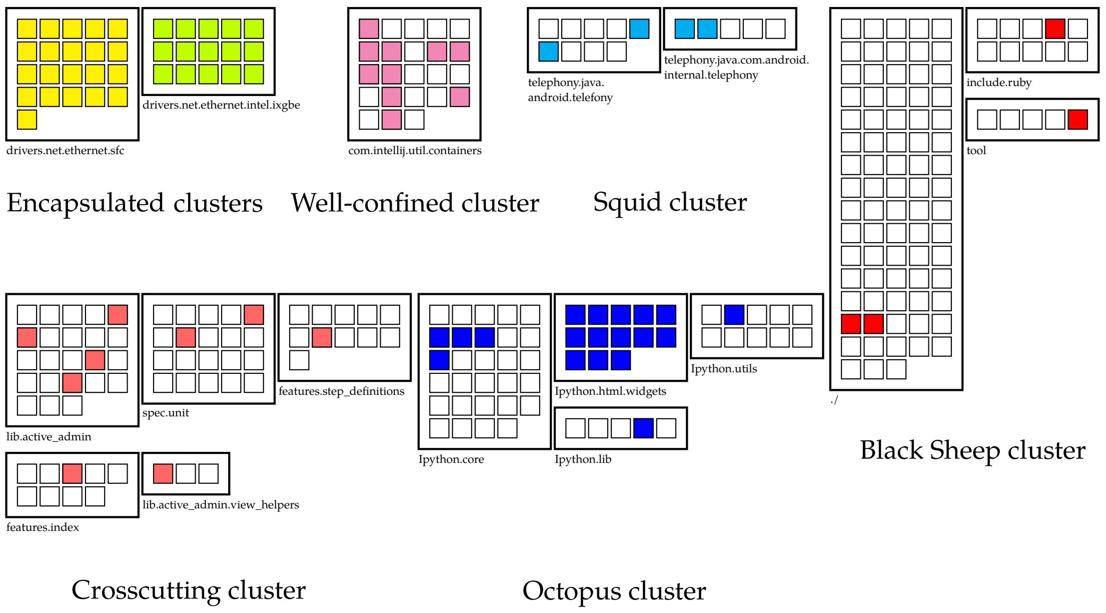

Figure 2.1: Co-change patterns proposed by Silva et al. [23]

2.3 Architectural Smells

Architectural technical debt is a form of technical debt and describes suboptimal design choices regarding the architecture of a software system, leading to negative consequences such as superfluous maintenance effort [16]. In contrast to code smells (also referred to as ″code anomalies″ or ″bad smells″ in literature) which manifest themselves on a granular layer such as the class or method level, architectural smells describe problems identified on the architecture level [9]. It is important to note that there are several definitions of the term ″architecture smell″ used by researchers, some of which overlap with the definition of code smells. However, architectural smells have been shown to be independent from code smells [7]. In this work, we use the definition by Garcia et al. and define an architectural smell as a "[...] commonly (although not always intentionally) used architectural decision that negatively impacts system quality" [8]. Except the "Encapsulated" and "Well-Confined" co-change pattern, we consider all of the co-change patterns proposed by Silva et al. [23] to be architectural smells.

2.4 Graph Clustering

Graph clustering refers to the act of assigning nodes in a graph to groups in such a way that nodes in one group are more similar to one another than to nodes outside of it. As explained above, Silva et al. use graph clustering based on co-changes to group files into modules and to identify architectural issues in a software system. In our work, we adapt their approach and use graph clustering to find modules based on data-flow dependencies between files. To this end, we use the *Chameleon* clustering algorithm.

Chameleon [12] is an agglomerative hierarchical clustering algorithm designed to overcome limitations of previously developed clustering algorithms. Its distinguishing feature is the fact that it takes both, the interconnectivity *and* the closeness of the clusters, into account when determining the most similar pair of clusters to be merged during the merging process, whereas previously developed clustering algorithms only considered interconnectivity *or* closeness of clusters. Being designed to operate on a sparse graph, Chameleon scales well even to large data sets. The clustering process of Chameleon consists of two phases. As input, it requires an adjacency matrix whose entries represent the similarity between data items. In the first phase, Chameleon clusters the data from the input matrix using the graph partitioning algorithm *hMetis*, which is based on the k-nearest-neighbor graph clustering approach. This results in a sparse graph where each node represents a data item and weighted edges represent similarities among two nodes. An edge between two nodes u and v exists if and only if u is among the k most similar data items of v or vice versa. This, in turn, results in unrelated data items being disconnected from each other in the graph. In the second phase, Chameleon repeatedly combines the subclusters created during the first phase into the final clusters. It does so by merging pairs of clusters with the highest relative connectivity and relative closeness until a user-specified threshold is reached.

The authors of Chameleon provide a software package called CLUTO [1] for clustering datasets. After computing a clustering solution for a given dataset, CLUTO presents several statistics on it. Among them are two values *ISim* and *ESim* which CLUTO computes for every cluster in the clustering. *ISim* refers to the average similarity between objects themselves *within each cluster*. *ESim*, on the other hand, refers to the average similarity between the objects within each cluster *and the remaining clusters*. In their work, Silva et al. [24] cluster the co-change graph multiple times in an effort to find the best clustering. In order to compare the quality

1 `https://glaros.dtc.umn.edu/gkhome/cluto/cluto/overview` (retrieved from the Wayback Machine on 2025-04-15)

of two clustering solutions, they introduce the $coefficient$ clustering quality function using the $ESim$ and $ISim$ metrics.

$$coefficient(M) = \frac{1}{k} * \sum_{i=1}^{k} \frac{ISim_{C_i} - ESim_{C_i}}{max(ISim_{C_i}, ESim_{C_i})}$$

In the function, k refers to number of clusters which remain after removing clusters containing fewer objects than a minimum threshold. The parameter M refers to the number of partitions to create during the first phase of the Chameleon algorithm. Using this function, Silva et al. [24] execute Chameleon several times in order to find the best clustering for a software project, where a higher $coefficient$ value indicates a better clustering solution.

MoJoFM [27] is an effectiveness measure which allows one to compare how much a clustering solution differs from a reference decomposition. It is defined as follows.

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

Here, $mno(A, B)$ denotes the minimum number of *Move* or *Join* operations necessary to transform a clustering A of a graph into a clustering B.$MoJoFM(A, B)$, where A is the clustering solution to be evaluated and B is the reference decomposition, yields a score of 100% if A and B are identical and a value of 0% if they are completely different. It is important to note that MoJoFM is not a symmetric function, meaning that the minimum number of *Move* or *Join* operations required in order to transform a partition A into a partition B is not necessarily the same as the other way around. As part of our experiment, we make use of the implementation provided by the MoJoFM authors [2] in order to compare the co-change based and the data-flow based model to one another.

2.5 LLVM Compiler Infrastructure

The LLVM project hosts various components for compiling, debugging and optimizing source code [3]. Originally, compilers were designed to only handle a particular programming language as input, making reusing parts of compilers and sharing code between projects very difficult. For this reason, LLVM was designed to work as a set of reusable libraries for optimizing and compiling code whose functionality can easily be extended and integrated into other projects [13]. Figure 2.2 and Figure 2.3 depict the conceptual design of classical compilers and LLVM, respectively. As part of the compilation process of LLVM, a frontend transforms human-readable source code into the LLVM Intermediate Representation, or LLVM IR for short. LLVM IR is a code representation based on Static Single Assignment which features type safety, allows low-level operations and is capable of cleanly representing high-level programming languages. It bears visual similarity to assembly language and is used as a common representation throughout all phases of the compilation process of LLVM. Figure 2.4 depicts a short C program performing a basic arithmetic operation and subsequently returning its result. Figure 2.5 shows the same program translated into LLVM IR.

2 `http://www.cs.yorku.ca/~bil/downloads/mojo.tar` (retrieved on 2025-01-14)
3 `https://llvm.org/` (retrieved on 2025-03-05)

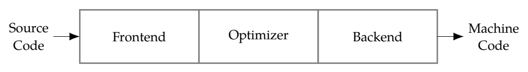

Figure 2.2: Classical compiler design (adapted from [13])

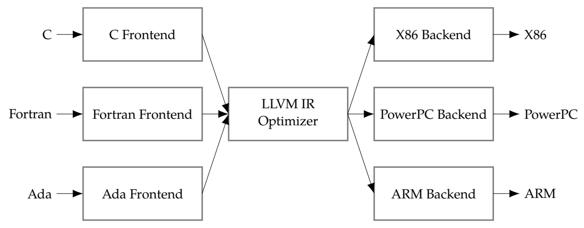

Figure 2.3: Conceptual design of LLVM (adapted from [13])

```c
#include <stdio.h>

void main() {
    int term1 = 34;
    int term2 = 8;
    int sum = term1 + term2;
    printf("%d\n", sum);
}
```

Figure 2.4: Simple addition in C

```llvm
; An externally declared function with at least one argument
declare i32 @printf(ptr noundef, ...)

; A string constant which is four characters long.
@.str = private unnamed_addr constant [4 x i8] c"%d\0A\00"

define dso_local void @main() {
  ; Allocate memory for three 32-bit integers on the stack frame
  %1 = alloca i32
  %2 = alloca i32
  %3 = alloca i32
  ; Store the integers 34 and 8 in memory
  store i32 34, ptr %1
  store i32 8, ptr %2
  ; Read read a 32-bit integer from memory (twice)
  %4 = load i32, ptr %1
  %5 = load i32, ptr %2
  ; Add two 32-bit integers together
  %6 = add nsw i32 %4, %5
  ; Store the result in memory
  store i32 %6, ptr %3
  %7 = load i32, ptr %3
  ; Call the printf function with two arguments
  call i32 (ptr, ...) @printf(ptr noundef @.str, i32 noundef %7)
  ret void
}
```

Figure 2.5: Simple addition in LLVM IR

LLVM IR allows annotating instructions with metadata in order to store additional information related to given source code, such as debugging information for use during code generation and code optimization. During the compilation process, it may be attached to instructions, functions and global variables. Additionally, LLVM features an optimizer which performs optimization passes on the IR. In our work, we annotate LLVM IR instructions with metadata in order to encode from which file an instruction originated.

2.6 VaRA

VaRA is a framework which allows researchers to perform static and dynamic analyses of interactions between user-defined regions of code, such as interactions between software features or code from different commits [21, 22]. Static analyses generate analysis results during compilation time or in a subsequent post-processing step.

At the heart of VaRA is the concept of code regions [21]. A code region is a consecutive set of instructions, each of which is mapped to the same user-defined tag representing some higher-level concept such as which feature an instruction belongs to. To formally define code regions, we assume that every program p consists of a number of functions $f_1, ..., f_n \in p$ which, in turn, consist of instructions, denoted by $inst(f_i)$. For each program p, we define a set τ in order to model variability information and a mapping function $tags(i)$. The set τ contains domain-specific tags where each tag $t \in \tau$ carries variability-specific information.

The function $tags(i)$ maps each instruction i to the set of tags which relate to said instruction, denoted by the @ symbol. Sattler points out that "the issues of determining how a specific tag is related to (@) an instruction i is defined by the concrete instance" [21]. Formally, Sattler [21] defines the function $tags(i)$ as follows.

Definition 1 $tags(i) = \{ t \mid t \in \tau \wedge t @ i \}$

A code region then groups consecutive instructions with the same tags. This framework approach allows researchers to define code regions of their own and, as a result, to run custom program analyses with ease. In order to compute the data-flow interactions between code regions, VaRA computes the data-flow interactions between individual instruction first. To this end, Sattler [21] introduces an interaction relation $\rightsquigarrow$.

Definition 2 $r_1 \rightsquigarrow r_2 \ = \ \exists\, i \in r_1\, \exists\, i' \in r_2\ DF(i, i')$

Intuitively speaking, the relation $r_1 \rightsquigarrow r_2$ holds if data created by at least one instruction i from code region r_1 is used as input by an instruction i' from code region r_2. In other words, a data-flow interaction between two code regions r_1 and r_2 exists if there is at least one instruction i in code region r_1 whose data used in an instruction i' from code region r_2.

VaRA supports several static analyses. In our work, we introduce a new kind of code region, namely *architecture regions*, by defining an architecture tag (see Section 3.1). This enables us to map each LLVM IR instruction to the source file from which it originated before subsequently performing a data-flow analysis which, due to our architecture tag, yields the data-flow interactions between architecture regions.

3 Data-Flow Based Architecture Analysis

In this chapter, we define architecture code regions and explain how we implement them in VaRA. Additionally, we give an overview over the implementation of our data collection and preparation pipeline.

3.1 Architecture Regions

In our work, we evaluate projects written in C. As we lack manually created architecture models at our disposal, we consider the architecture of a project to be a nested structure of directories, each of which represents a module containing one or more source files. In order to compute the data-flow behavior of a project, we compile the project in question and use the data-flow interaction analysis of VaRA. Since the data-flow interaction analysis in itself does not allow us to tell in which file an instruction is defined, we adapt the compilation process to suit our needs. This allows us to read out in which file an instruction is defined after the data-flow interaction analysis and, in turn, allows us to construct a data-flow interaction graph for our experiment. As part of the changes which we make to VaRA, we introduce a new code region, namely an *architecture region* which we use in order remember from which file an instruction originated. Mathematically speaking, we extend the set of tags τ (see Definition 1) by a tag which carries architecture information. This, in turn, allows us to annotate code regions and their instructions so that the architecture information carried by the LLVM IR metadata node can be processed by VaRA. Conceptually, we add a new element t_{arch} to the set of tags τ.

Definition 3 $\tau_{arch} \;=\; \{t_{arch}\} \;\cup\; \tau$

Said tag t_{arch} carries a string corresponding to the path of the file in which an instruction is defined. Additionally, we implement a new tagging function $tags_{arch}(i)$.

Definition 4 $tags_{arch}(i) = \{\; t \;\mid\; t \in \tau_{arch} \;\wedge\; t \,@\, i \;\}$

This allows us to use the analysis framework of VaRA to analyze the architecture of a project.

3.2 Architecture Interactions

When data from one variable defined in architecture region r_{a_1} is used in an instruction from architecture region r_{a_2}, there is a data-flow relation between the architecture regions r_{a_1}

and r_{a_2}. That is, the relation $\rightsquigarrow$ (see Definition 2) holds for regions r_{a_1} and r_{a_2}. After annotating the instructions with our architecture tag and grouping them into code regions, we perform a data-flow interaction analysis in order to compute interactions between architecture regions. Put differently, we check for which architecture regions relation $\rightsquigarrow$ holds. We obtain the interactions for every instruction and, using the string carried by our architecture tag t_{arch}, map the instruction back to the source file where it is defined.

3.3 Implementation

In this section, we explain the process for preparing and analyzing data for our experiment. First, we explain the process of annotating instructions and grouping them into architecture regions in Section 3.3.1. Then, we go over how we collect data-flow interaction information in Section 3.3.2. Finally, we explain how we prepare the resulting data for our experiment procedure in Section 3.3.3. Figure 3.1 depicts an overview of the steps for data preparation and data analysis in our experiment.

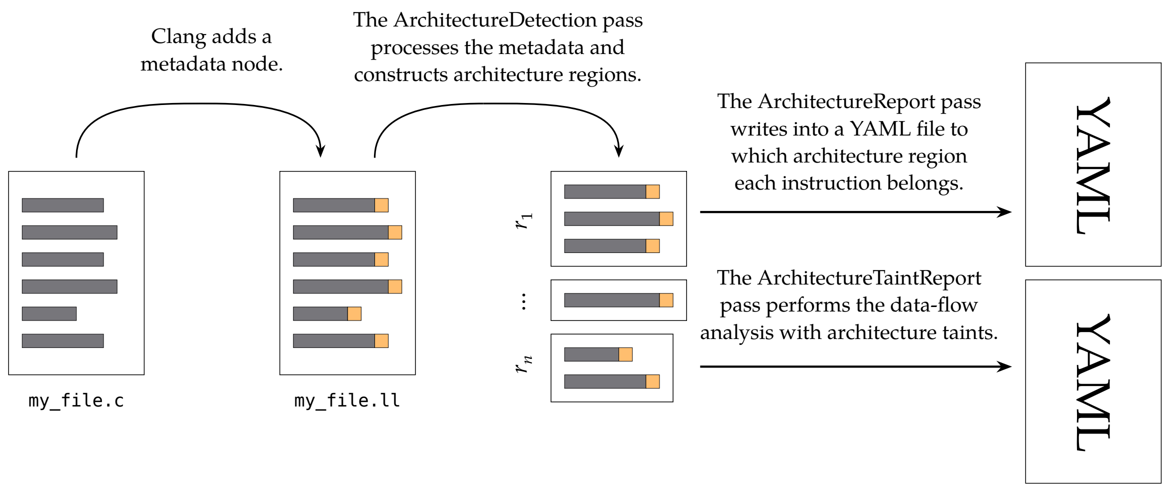
Clang adds a metadata node.
The ArchitectureDetection pass processes the metadata and constructs architecture regions.
The ArchitectureReport pass writes into a YAML file to which architecture region each instruction belongs.
The ArchitectureTaintReport pass performs the data-flow analysis with architecture taints.
my_file.c
my_file.ll
r_1
⋮
r_n
YAML
YAML

Figure 3.1: After Clang transforms source code into IR and annotates it with metadata, VaRA performs several steps in order to generate results files for further processing.

3.3.1 Architecture Annotation

VaRA builds on the LLVM Compiler Infrastructure and heavily utilizes its framework based on IR for source code and binary analysis. For this reason, in order to implement a new kind of code region in VaRA and use it as part of our experiment, we to have to extend Clang, the C frontend of LLVM, first.

3.3.1.1 *Adding a New Metadata Node*

Every kind of code region implements its own $tags()$ function. Consequently, to implement architecture regions, we first have to implement the $tags_{arch}()$ function. We do so by adding a new metadata node to Clang. Because we consider files to be the fundamental unit in our architecture model, we use our newly created metadata node in order to encode from which file an instruction originates. More precisely, the information carried by the metadata node says in which file the source code is defined from which the instruction was generated. To this end, we extend the code generation step of Clang which transforms source code written in C into LLVM IR. We introduce a step to the code generation process which annotates each IR instruction with our metadata node. This metadata node represents our tag t_{arch} and allows us to remember in which file an instruction is defined. It is important to note that, in theory, an instruction may potentially be defined in several files. We take this into account and allow Clang to annotate an instruction with several pieces of metadata. To make the metadata usable by the data-flow analysis, we define a new code region in VaRA.

3.3.1.2 *Constructing Architecture Regions*

We introduce a new region to VaRA, namely an architecture region. Every architecture region has a unique identifier. This identifier corresponds to the path of the source file in which the instructions in the architecture region are defined. During the `ArchitectureDetection` pass, VaRA parses the IR metadata and groups consecutive instructions into code regions based on their architecture tag. This allows us to process architecture regions and to collect data to conduct our experiment.

3.3.2 Data Collection

In the compilation step, we instruct Clang to compile the project under investigation without optimizations and to enable architecture metadata generation. We deliberately disable optimizations in order to improve the accuracy of metadata annotations. In the analysis step, we instruct VaRA to construct architecture regions and to generate an architecture report using our `ArchitectureDetection` and `ArchitectureReport` passes. We would like to stress that a third party added the `ArchitectureTaintReport` pass to VaRA while this thesis was in preparation. The `ArchitectureReport` pass yields raw architecture information, that is, a mapping between each instruction and its source file of origin, and merely serves debugging purposes. On the other hand, the `ArchitectureTaintReport` pass performs a data-flow interaction analysis between architecture regions. The result file, which the `ArchitectureTaintReport` pass generates, contains, for every function in each source file, between how many architecture regions from other source files the data-flow relation holds.

3.3.3 Data Analysis

Using the result file of the `ArchitectureTaintReport`, we transform the machine-readable format into a data-flow interaction graph for further processing. In the data-flow interaction graph $G_{df} = (V_{df}, E_{df})$, nodes constitute source files and weighted undirected edges between nodes u and v, with $u \neq v$, represent the number of data-flow interactions between architecture regions from the files u and v.

We rely on the CLUTO package in order to cluster the data-flow interaction graph using the Chameleon clustering algorithm and adapt the procedure employed by Silva et al. [24] in order to find the optimal clustering solution. For each $n \in 2..|V_{df}|$, we compute a clustering solution with n partitions. Next, we calculate the $coefficient$ value in order to determine the quality of the clustering solution. We repeat this process for all $|V_{df}| - 1$ clusterings and pick the clustering solution with the highest $coefficient$ value. If the $coefficient$ value is the same for two or more n, we opt for the clustering solution with the highest mean $ISim$ value. We obtain a data-flow interaction cluster graph for conducting our experiment.

4 Methodology

In this chapter, we describe the methodology of our thesis. First, we present our research questions with which we aim to investigate the relationship between co-change patterns and data-flow interactions. Next, we discuss the criteria for our project selection and detail our operationalization.

4.1 Research Questions

In our evaluation, we investigate the differences between architectural models extracted from co-changes and data-flow interactions. Specifically, we formulate the following research questions.

RQ1 How well do architecture models extracted from co-changes and data-flow interactions agree with one another?

In our first research question, we explore the relationship between architecture models extracted from the co-changing files of a project and its data-flow interactions. Specifically, we investigate the commonalities and differences of these two kinds of models.

RQ2 Which architectural smells can we detect using our data-flow based architecture model?

In our second research question, we consider the architectural smells which we can find using our architecture model. Specifically, we apply the co-change patterns introduced by Silva et al. to our setting and compare the patterns found between co-changes and data-flow interactions.

4.2 Project Selection

We choose the projects to be part of our experiment based on their size and longevity. We ensure that each project is both sufficiently large and sufficiently old so that the data-flow analysis and the co-change analysis respectively yield meaningful results. At the same time, we limit our selection to medium-sized projects due to the fact that the data-flow interaction analysis is a computationally expensive procedure. Importantly, the directory structure of the files used in the compilation process of the software project must not be flat in order to answer RQ2. Table 4.1 depicts our project selection.

Project	Language	Category	Commit	SLOC	Longevity
`htop`	C	UNIX utility	`4102862`	35.6 k	Mar 2006 - Oct 2024
`libvpx`	C	Codec	`027bbee`	315.6 k	May 2010 - Mar 2025
`opus`	C	Codec	`b5aad6a`	89.1 k	Nov 2007 - Mar 2025
`toxcore`	C	Cryptographic library	`1d4cc78`	82.5 k	Jun 2013 - Mar 2025
`xz`	C	Compression utility	`74c3449`	48.9 k	Dec 2007 - Aug 2023

Table 4.1: List of projects selected for our experiments.

4.3 Operationalization

In the following section, we describe our experiment design in more detail and explain our data collection and evaluation process.

Figure 4.1 depicts our experiment structure. For every project under investigation, we perform the following steps. First, we obtain its source code along with its revision history. We start by building a data-flow interaction graph as part of our experiment. On the project under investigation, we perform the data-flow interaction analysis of VaRA in order to obtain data-flow interactions between architecture regions. Next, we follow the steps in Section 3.3.3 and construct a data-flow interaction cluster graph. Subsequently, we use the revision history of the project to build a co-change graph. We consider the entire revision history up until the commit listed the column labeled "Commit" in Table 4.1. It is important to note that, unlike Silva et al. [23], we do not preprocess the revision history of the project as part of our experiment procedure. This means that we do not merge commits, neither based on reports in issue tracking systems nor using sliding windows of time. The nodes of the co-change graph represent the source files of the project and each edge between two nodes is assigned a weight. The weight represents how often which the corresponding files change in the same revision. Consequently, the more often a pair of files changes in tandem, the greater the weight of the edge connecting them is. Next, we remove all files from the co-change graph which are not present in the data-flow interaction graph in order to remove noise from our dataset. In the ideal outcome, both the co-change graph and the data-flow interaction graph contain the same set of nodes, only differing in their edge weights. However, files which are present in the data-flow interaction graph do not necessarily exhibit co-change behavior. As a consequence, the resulting co-change graph may contain fewer nodes than its corresponding data-flow interaction graph. After successfully obtaining both graphs, we use CLUTO and repeat the clustering procedure in Section 3.3.3 in order to find the optimal clustering solution for the co-change graph. As part of this process, every node in the graph is assigned a cluster to which it belongs and we obtain a co-change cluster graph. Using the data-flow interaction cluster graph and the co-change cluster graph, we perform the following steps in order to answer our research questions.

RQ1. In order to answer our first research question, we assess the similarity between the co-change cluster graph and the data-flow interaction cluster graph. We do so by computing the MoJoFM score of the two cluster graphs. We opt for using the data-flow interaction cluster graph as the reference as the co-change cluster graph constitutes the greater bottleneck in terms of number of nodes. This yields a straightforward measure which allows us to compare the graphs to one another and, in turn, to quantify the difference between the architecture models under study.

RQ2. To answer our second research question, we map the nodes in the data-flow interaction cluster graph to their respective files in each directory of the project. Next, we use the co-change patterns by Silva et al. [23] in order to identify architectural smells. We repeat the procedure with the co-change cluster graph and compare the results to each other.

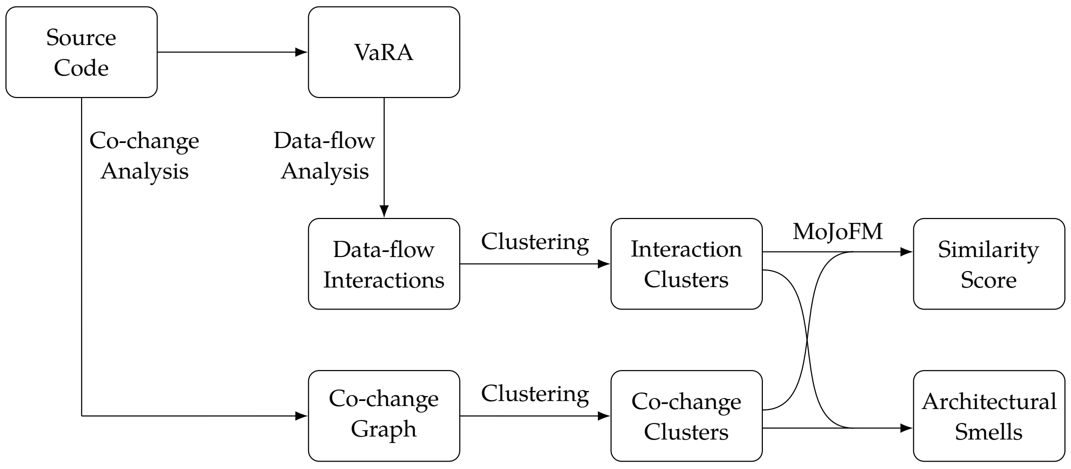

Figure 4.1: Our experiment procedure

5

Evaluation

In this chapter, we evaluate our experiment results and answer our research questions. In Section 5.1, we present and describe the results of our experiment. Then, we discuss our experiment results and use them to answer our research questions in Section 5.2. Finally, we go over threats to the internal and external validity of our experiment in Section 5.3.

5.1 Results

In this section, we present the results of our experiment and point out noteworthy details in our data.

5.1.1 RQ1

In order to answer our research questions, we prepare a co-change graph and a data-flow interaction graph for every project under investigation. It is important to keep in mind that we construct the data-flow interaction graph first, meaning that only files present in the data-flow interaction graph can be present in the co-change graph as we filter out other files. A node representing a file can exist in both the data-flow interaction graph and the co-change or in the data-flow interaction graph only. Due to our experiment set-up, it is not possible for a node to be present only in co-change graph. As a result, the co-change graph of each project has at most as many nodes as the data-flow interaction graph. Table 5.1 shows the number of files used when constructing the respective graph of each project. Additionally, it shows across how many directories the files are spread. We denote the columns concerning the co-change graph using "CC", and columns relating to the data-flow interaction graph are marked with "DF". The abbreviations `#files` and `#directories` refer to the number of files and directories, respectively. First, it is easy to notice that the co-change graphs belonging to three of the five projects under investigation are substantially smaller than the corresponding data-flow interaction graphs. Out of the three projects, `toxcore` saw the greatest decrease in terms of the number of nodes, going from 86 nodes in the data-flow interaction graph to merely 49 nodes in the co-change graph, totaling a decrease of 43%. The other projects, namely `htop` and `opus`, saw a decrease of 22.6% and 24.5%, respectively. Finally, `xz` saw a modest decrease, with its number of nodes lowering from 13 to 11, or by 15.4%. Strikingly, `libvpx` remained the same. Both the data-flow interaction graph and the co-change graph corresponding to `libvpx` contain 13 nodes. In terms of the number of directories across which files are spread, `htop`, `libvpx` and `opus` share their source files across 2, 3 and 6 directories, respectively, both in the co-change graph and in the data-flow interaction graph. In the case of `toxcore` and `xz`, the number of unique directories in the the graph decreases from 4 to 3 and from 2 to 1, respectively.

Project	#files (DF)	#directories (DF)	#files (CC)	#directories (CC)
`htop`	31	2	24	2
`libvpx`	12	3	12	3
`opus`	49	6	37	6
`toxcore`	86	4	49	3
`xz`	13	2	11	1

Table 5.1: Total number of files and directories per project and architectural model

Next, we go over the optimal clustering solutions found by using the graph quality metrics provided by CLUTO *ESim* and *ISim* as well as the *coefficient* metric by Silva et al. Table 5.2 depicts the optimal clustering solutions which we build upon in order to answer our research questions. Notably, for 3 out of 5 co-change graphs, the optimal clustering solution was found to contain the same number of partitions. For `libvpx`, `toxcore` and `xz`, the respective co-change cluster graphs contain merely two clusters. On the other hand, the co-change cluster graph of `htop` contains six clusters. For `opus`, optimal clustering for the co-change graph consists of fifteen clusters. On the other hand, its data-flow interaction cluster graph with the highest *coefficient* value contains only 2 clusters. The data-flow interaction cluster graphs for `htop`, `libvpx`, `toxcore` and `xz` consist of 9, 3, 10 and 3 partitions, respectively. It is easy to see that the optimal clustering solutions for the co-change graphs in question are concentrated at the lower end of the spectrum, with the solutions of the `opus` and `htop` co-change cluster graphs being outliers. In the case of the data-flow interaction graphs, the number of clusters for the optimal clustering solutions tend both towards the lower and the higher end.

Project	Opt. Clustering (DF)	Opt. Clustering (CC)
`htop`	9	6
`libvpx`	3	2
`opus`	2	15
`toxcore`	10	2
`xz`	3	2

Table 5.2: Optimal number of clusters per project and architectural model

In our first research question, we examine the overall similarity between both architectural models using the MoJoFM score. Table 5.3 depicts the computed MoJoFM scores. The column labeled $MoJoFM(CC, DF)$ contains the MoJoFM score where the data-flow interaction cluster graph is considered the reference decomposition. For `htop` and `toxcore`, we find the lowest MoJoFM scores among our project selection, namely 27.78% and 31.71%, respectively. Out of `libvpx`, `xz` and `opus`, the MoJoFM score of `opus` of 51.43% is the highest. `xz` has a MoJoFM score of 50.0% and the MoJoFM score of `libvpx` is 44.44%.

Project	**MoJoFM(CC, DF)**
`htop`	27.78
`libvpx`	44.44
`opus`	51.43
`toxcore`	31.71
`xz`	50.0

Table 5.3: MoJoFM values per project

5.1.2 RQ2

With RQ2, we gain insight into which architectural model better aids in uncovering architectural smells. In particular, we compare which architectural smells are found using the co-change based or data-flow based architecture model. In order to do so, we examine the qualitative differences exhibited by our results for each architecture model. Overall, we find that 57 out of 133 (42.9%) files in the co-change graphs of the respective projects are part of an architectural smell. 67 files (50.4%) belong to an Encapsulated or Well-Confined cluster and 9 files (6.8%) belong to clusters which fail to match any pattern.

In the case of the data-flow interaction graph, 56 out of 191 nodes (29.3%) of the nodes belong to an instance of an architectural smell. 65 files (34.0%) belong to an Encapsulated or Well-Confined cluster and a total of 70 files (36.6%) belong to clusters which do not match any pattern. Additionally, among the projects in which source files from four or more unique directories were involved, we find no instances of the Crosscutting architectural smell. Lastly, it is worth noting that, there is only one instance of the Black Sheep architectural smell, and one instance of the Encapsulated cluster, both of which appear in the same project.

In the remainder of this subsection, we go over the individual projects, present our findings and highlight notable observations. Throughout the individual subsubsections, we employ the same color coding as in Figure 2.1, meaning that green squares represent files belonging to an Encapsulated cluster, pink squares depict files forming Well-Confined clusters, red squares represent Black Sheep clusters, and dark blue and light blue squares belong to Octopus and Squid clusters, respectively. It is important to note that, when we refer to files and directories, we only consider files and directories represented by nodes in the respective cluster graphs.

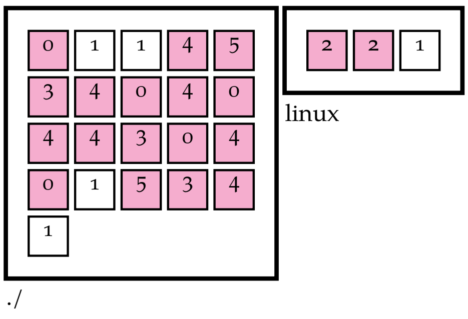

Figure 5.1: Co-change clustering results for `htop`

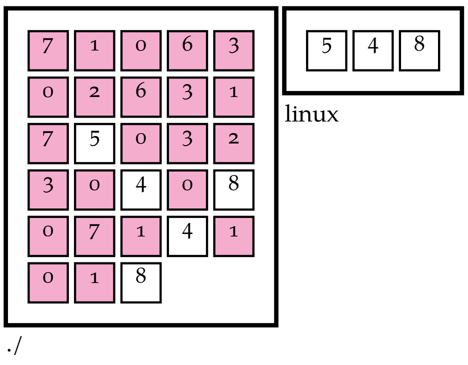

Figure 5.2: Data-flow interaction clustering results for `htop`

5.1.2.1 *`htop`*

Out of the 6 clusters in the co-change cluster graph, 5 of them are classified as Well-Confined clusters, with four of them being confined to the `./` directory. The remaining cluster, cluster 1, is spread across both the `./` and `linux` directory, and does not match any of the architectural smells which we investigate. Notably, cluster 2, which only contains the `LinuxMachine.c` and `LinuxProcessTable.c` files, is confined to the `linux` directory. However, because it shares the `linux` directory with the `Platform.c` file, which is part of cluster 1, cluster 2 is not an Encapsulated, but a Well-Confined cluster. Figure 5.1 provides a visualization of the co-change cluster graph mapped onto the directory structure of `htop`.

In the data-flow based architecture model, we again see the majority of files being categorized as Well-Confined. Out of the 9 clusters, only 3 clusters fail to match an architectural smell. Notably, the `linux` directory only contains uncategorized files whose respective clusters extend into the `./` directory. Figure 5.2 depicts a visual representation of the clustering results for the data-flow based model.

For `htop`, we find similar results overall in terms of architectural smells for both models. The co-change based and the data-flow based clustering belonging to `htop` exhibit similar behavior in terms of architectural smells. Both models exclusively detect Well-Confined clusters in the source files of `htop`, with several files remaining uncategorized.

5.1.2.2 *`libvpx`*

For `libvpx`, we obtain a clustering with 2 clusters in the co-change based architecture model. In the `vpx/src` directory, the files `vpx_codec.c`, `vpx_decoder.c`, `vpx_encoder.c` make up a Well-Confined cluster. The remaining file in the `vpx/src` directory, namely `vpx_image.c`, is part of an Octopus cluster spread across all 3 directories. The entirety of the `./` belongs to the said cluster and constitutes its body. The tentacles of the Octopus cluster reside in the `vpx_image.c` file in the `vpx/src` directory and in the `vpx_mem.c` file, the latter being the only file in the `vpx_mem` directory.

On the other hand, the data-flow based cluster graph contains three clusters. We observe a similar clustering of the `vpx/src` and `vpx_mem` directories. Again, the files `vpx_image.c` and `vpx_mem.c` belong to the same cluster. However, in the case of the data-flow based model,

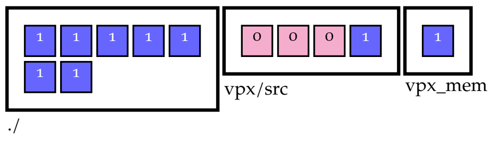

Figure 5.3: Co-change clustering results for `libvpx`

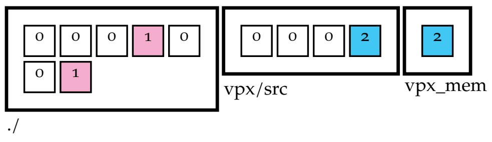

Figure 5.4: Data-flow interaction clustering results for `libvpx`

they make up a Squid cluster on their own. The remaining files in the `vpx/src` and all files but two in the `./` directory belong to a cluster of their own and are categorized as an architectural smell. In the `./` directory, the `tools_common.c` and `y4minput.c` files constitute a Well-Confined cluster.

In contrast to the data-flow based architecture model, the co-change based model classifies all files in the directories of `libvpx`. On the other hand, most of the files in the `./` and `vpx/src` directories remain uncategorized in the case of the data-flow based model. Figure 5.3 and Figure 5.4 visualize the clustering results of `libxpv` for the co-change based model and the data-flow based model, respectively.

5.1.2.3 *opus*

For `opus`, we see drastically different clustering behavior between the co-change graph and data-flow interaction graph. The majority of the clusters in the co-change cluster graph are considerably small in size. Out of the 15 clusters found, a total of 11 clusters merely contain two files. Furthermore, 2 clusters contain three files, and the remaining 2 clusters contain four and five files, respectively. Strikingly, a number of clusters of size two contain files whose names suggest that said files are related to one another or serve the same purpose. For example, cluster 0 encompasses the files `opus_decode.c` and `opus_encoder.c` from the `src` directory. In the same directory, cluster 1 contains the files `opus_multistream_decoder.c` and `opus_multistream_encoder.c`. In cluster 6, we find the files `dec_API.c` and `enc_API.c` from the `silk` directory. Similarly, cluster 10 consists of the files `NLSF_encode.c` and `NLSF_VQ.c` from the `silk` directory. In the `silk/x86` directory, the files `NSQ_sse4_1.c` and `VAD_sse4_1.c` make up cluster 8. The files `NSQ.c` and `NSQ_del_dec.c` from the silk directory belong to cluster 5. The files `vq.c` from the `celt` directory and `vq_sse2.c` from the `celt/x86` directory constitute cluster 14. With the exception of cluster 14, which is a Squid cluster, all of the clusters mentioned above are categorized as Well-Confined clusters. Further Well-Confined clusters include cluster 12, which consists of the files `cwrs.c` and `modes.c`, and cluster 9, which encompasses the files `entdec.c` and `laplace.c`, all of which reside in the `celt` directory. Additionally, the clusters 3 and 7, both of which are of size three, are Well-Confined too. The files `mapping_matrix.c`, `opus_projection_decoder.c` and `opus_projection_encoder.c` from the `src` directory make up the former, whereas the latter consists of the files `code_signs.c`, `decode_pulses.c` and `shell_coder.c` from the `silk` directory. The remaining clusters of size two deserve particular attention, as they are the only two instances of a Black Sheep cluster and an Encapsulated cluster. Cluster 2, which comprises `celt.c` from the directory `celt` and `analysis.c` from the directory `src`, is the only Black Sheep cluster we find in all of our experiment results. Cluster 13, consisting of the files `pitch_analysis_core_FLP.c` and `wrappers_FLP.c` from the

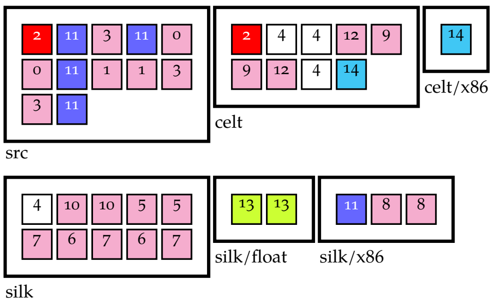

Figure 5.5: Co-change clustering results for opus

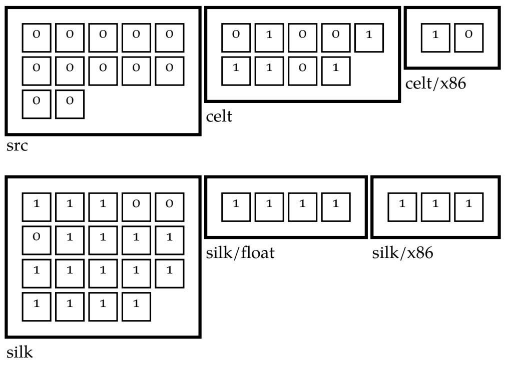

Figure 5.6: Data-flow interaction results for opus

`silk/float` directory, is the only Encapsulated cluster which we encounter. We do not find further Encapsulated clusters in other projects which we investigate. The last two clusters in the co-change cluster graph of `opus` are of size four and five, respectively. Cluster 4 spans three files from the `celt` directory, namely `celt_decoder.c`, `celt_encoder.c` and `quant_bands.c`, and `LPC_analysis_filter.c` from the `silk` directory. It does not match any co-change patterns. Cluster 11 consists of the file `NSQ_del_dec_avx2.c` from the `silk/x86` directory, and `extensions.c`, `opus.c`, `opus_multistream.c` and `repacketizer.c` from the `src` directory. Together, they form an Octopus cluster. Figure 5.5 contains a visualization of the clustering results for the co-change based model.

In the case of the data-flow interaction graph, we observe a vastly different clustering results compared to the co-change cluster graph. While the latter contains fifteen clusters, which is the highest number of clusters in our experiment results, its corresponding data-flow interaction cluster graph merely contains two. The co-change cluster graph has several smaller clusters, with the majority of clusters containing two files. On the other hand, cluster 0 in the data-flow interaction cluster graph encompasses 20 files and cluster 1 has 29 files. Notably, the entirety of the files in the `src` directory belongs to cluster 0, whereas all of the files in the `silk/float` and `silk/x86` directories are part of cluster 1. The directories `celt`, `celt/x86` and `silk` are touched by both clusters. Furthermore, several files which appear in the data-flow interaction graph are absent from the co-change graph due to not exhibiting co-change behavior. However, the `celt`, `silk/x86` and `src` directories contain the same files in both the co-change graph and the data-flow interaction graph. Figure 5.6 depicts the clustering results of `opus` for the data-flow based model.

5.1.2.4 *toxcore*

For the project `toxcore`, we find two clusters in the co-change based model. The entirety of the `toxcore` directory belongs to the body of an Octopus cluster whose tentacles touch one file out of eleven files in the `toxcore/events` directory and `toxencryptsave.c`, the only file in the `toxencryptsave` directory. The remaining files in the `toxcore/events` form a Well-Confined cluster. In the co-change based model, several files are absent from the `toxcore/events` directory, the majority of which are related to conferencing, friend and group functionality.

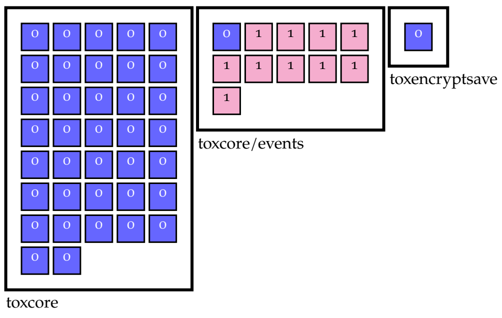

Figure 5.7: Co-change clustering results for `toxcore`

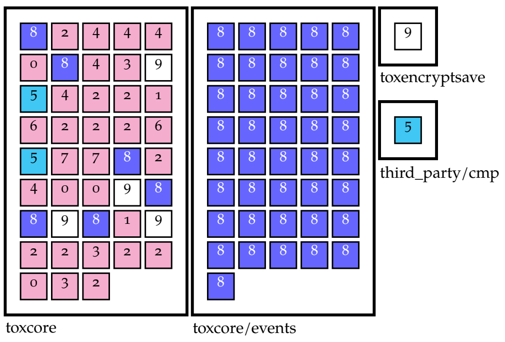

Figure 5.8: Data-flow interaction clustering results for `toxcore`

The file `cmp.c` from the `third_party/cmp` is also missing. Figure 5.7 visualizes the clustering results of `toxcore` for the co-change based model.

In the data-flow based model, we also find an Octopus cluster. However, unlike in the co-change based model, its body resides in the `toxcore/events` directory, with its tentacle extending into the `toxcore` directory and touching six files. Furthermore, we find seven Well-Confined clusters and a Squid cluster in the `toxcore` directory, with the body of the latter being the `cmp.c` file in the `third_party/cmp` directory. One cluster, consisting of four files in the `toxcore` and `toxencryptsave.c` from the `toxencryptsave` directory remains uncategorized.

Figure 5.8 contains a visualization of the clustering results of `toxcore` for the data-flow based model.

5.1.2.5 xz

For `xz`, we find only two clusters in the co-change based model. Both clusters are Well-Confined and reside in the `src/xz` directory, which is the only directory in the co-change based model.

In the data-flow based model, we again find two Well-Confined clusters in the `src/xz` directory. Additionally, it contains the tentacles of a Squid cluster, namely `mytime.c` and `message.c`, whose body resides in the `src/common` directory and consists of the `tuklib_progname.c` and `tuklib_exit.c` files.

Figure 5.9 and Figure 5.10 visualize our results for the co-change based and the data-flow based model, respectively.

5.2 Discussion

In this section, we discuss our experiment results and relate them to our research questions.

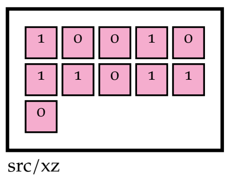

Figure 5.9: Co-change clustering results for xz

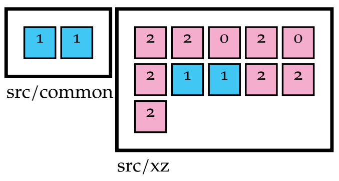

Figure 5.10: Data-flow interaction clustering results for xz

5.2.1 RQ1

In RQ1, we explore how well architecture models extracted from co-changes and data-flow interactions agree with one another. To this end, we compute the MoJoFM score for each project.

We observe that the maximum MoJoFM score computed for the projects under investigation falls into the middle range, namely `opus` with 51.43%. At the lower end, we find the MoJoFM score of `htop` which is roughly 28%. The respective scores of three out of five projects tend towards the middle range. The remaining two projects fall into the lower end of the spectrum. Based on our results, we conclude that both architecture models differ rather significantly from each other. In the best case, they bear slight similarity to one another.

5.2.2 RQ2

In RQ2, we investigate which architectural smells our data-flow based model aids in identifying. To this end, we compare our data-flow based model with the co-change based model in terms of the number of detected architectural smells. For each co-change pattern, Table 5.4 depicts the total number of detected instances per architectural model.

Co-Change Pattern	**DF**	**CC**
Encapsulated	0	1
Well-Confined	16	19
Crosscutting	0	0
Black Sheep	0	1
Octopus	1	3
Squid	3	1
Uncategorized	7	2
Total	27	27

Table 5.4: Sum of co-change pattern instances per architectural model

We find a total of 54 clusters across all projects and both architectural models, where both the data-flow based model and the co-change based model contain 27 clusters each. In the data-flow based model, 16 out of 27 clusters are classified as Well-Confined. A total of 4 clusters are categorized as architectural smells, as there is 1 instance of an Octopus cluster and 3 instances of Squid clusters. 7 clusters remain uncategorized in the data-flow based model.

In the co-change based model, only 2 clusters do not match any co-change pattern. 1 cluster is categorized as Encapsulated and 19 clusters are Well-Confined. The remaining 5 clusters are classified as architectural smells, with one being a Black Sheep cluster and 3 being categorized as Octopus clusters, and 1 Squid cluster.

Out of the nine uncategorized clusters, five of them slightly miss the criteria for Black Sheep, Octopus and Squid clusters. In the data-flow interaction clustering, for cluster 4 and 8 of `htop` and for cluster 9 of `toxcore`, we find $focus$ values between 0.16 and 0.27, which are slightly above the threshold of Black Sheep and slightly below the threshold of Octopus and Squid clusters. Similarly, cluster 1 of `htop` and cluster 4 of `opus` in the co-change clustering have $focus$ values of 0.21 and 0.28, respectively.

Using their six co-change patterns, Silva et al. were able to classify approximately 95.4% of co-change clusters in their experiment. In our experiment, we find a slightly smaller percentage of clusters, as only 92.6% of co-change clusters and 74.1% of data-flow interaction clusters are covered by their patterns. Excluding the Encapsulated and Well-Confined clusters, we see that 18.5% of clusters are detected as architectural smells in the co-change cluster graph. Moreover, 14.8% of clusters in the data-flow interaction cluster graph belong to architectural smells.

We observe that there are more Squid clusters in the data-flow interaction cluster graph than in the co-change cluster graph. Furthermore, there are more Octopus clusters in the co-change cluster graph than in the data-flow interaction cluster graph. However, we note that the sum of Squid and Octopus clusters is equal across both models. While the overall number of architectural smells detected by both models varies only slightly, the files themselves and to what architectural smells they belong exhibit significant differences between the co-change based and the data-flow based model.

In summary, we find that the co-change patterns fit better to the co-change based model than to the data-flow based model. We do not encounter architectural smells which are only detected using the data-flow based model. Instead, we see that the absolute number of architectural smells detected by the the co-change based model is greater, as it detects an instance of a Black Sheep cluster whereas the data-flow based model does not. Additionally, the number of uncategorized clusters is higher in the case of the data-flow based model.

We select the six co-change patterns designed by Silva et al. [23] to serve as indicators of architectural smells in our experiment. Considering the fact that the patterns are specifically designed to capture the co-change behavior of a software system, it seems logical that using them in conjunction with data-flow interactions instead of co-changes yields fewer matches. Designing new shapes specifically tailored to data-flow interactions may potentially overcome this issue. Silva et al. [23] note that there can be other sets of patterns and that their set of patterns can be extended, as they did in their previous work. Consequently, it may prove beneficial to design an initial set of patterns for data-flow interactions and to expand upon it subsequent experiments.

5.3 Threats to Validity

In this section, we discuss the threats to the internal and external validity of our experiment, its results and our conclusions.

5.3.1 Internal Validity

In our experiment, we use the MoJoFM effectiveness measure in order to compare clustering results. MoJoFM is intended for use with an authoritative decomposition as a reference. As it is not a symmetric function and we lack a reference decomposition for the projects under investigation, our results for RQ1 highly depend on our choice to use the data-flow interaction cluster graph instead of an authoritative decomposition. While this approach allows us to compare our data-flow based model to the co-change based model using a straightforward measure, comparing both to an authoritative decomposition would arguably yield better insight into the similarity between both models.

Additionally, the MoJoFM implementation of which we make use as part of our experiment only considers nodes present in both clustering solutions when calculating the MoJoFM score. This means that, for all projects but `libvpx`, the MoJoFM values in Table 5.3 concern the intersection between the co-change cluster graph and the data-flow interaction cluster graph. Consequently, this reduces the meaningfulness of the MoJoFM score for the purpose of comparing both architectural models. `libvpx` is unaffected by this behavior due to the fact that its co-change graph and data-flow interaction graph contain the same set of nodes.

When constructing the co-change graph, we do not preprocess the revision history of the project under investigation. Specifically, we do not to merge commits using sliding windows of time. This, in turn, potentially leads to files concerning the same development task to being split across separate commits, preventing their connection from appearing in the co-change graph.

5.3.2 External Validity

During the compilation process of the projects under investigation, we ran into issues related to the data-flow interaction analysis of VaRA. Due to memory constraints, we were unable to successfully compile a number of projects which imposed strict limits in terms of the size of the projects in our selection. As a consequence, when conducting our experiment with the projects listed in Table 4.1, we found that, despite the nested nature of their directory structure, only files from few unique directories were used during the compilation process for most projects. Three out of five projects compiled with files from three directories or less. In the case of `toxcore`, only the data-flow interaction graph contains files belonging to four directories. Only the co-change graph and the data-flow interaction graph of `opus` both contain files from more than four directories. This effectively prevents us from identifying Crosscutting clusters in three of our five projects under investigation, thereby reducing the generalizability of our findings. Additionally, our project selection mainly focuses on UNIX utilities and media codecs, which does not properly reflect the broader software ecosystem.

The data-flow interaction analysis of VaRA is a computationally expensive process. During the preparation of our experiment, VaRA failed to complete a data-flow interaction analysis for multiple projects even after several days. On the other hand, a revision history of a project is relatively simple to obtain and analyze, making the co-change based approach arguably cheaper and faster in real-world scenarios than our data-flow based approach.

6 Related Work

In this chapter, we briefly present related work in the field of software architecture.

6.1 Architecture Models

Bass et al.[1] present several architectural views which go beyond units of code and their semantics. In their work, they state that the dominant structure which other structures follow often is the modules of which the software system is composed.

Xiao et al. [26] propose a model referred to as design rule spaces which splits a system into independent modules based on one or several rules which include evolutionary history, among others. They argue that a software system has to be considered as a number of overlapping design rule spaces, each one of which captures one aspect of the software system. Additionally, they find that not all structural problems which a software system exhibits necessarily lead to quality issues or degrade maintainability.

6.2 Architecture Recovery

Manually recovering the architecture of a software system is a highly laborious task and requires substantial human intervention. Godfrey et al. [10] recovered the architecture of early versions of the Vim text editor and the Firefox web browser by using a combination of several tools and consulting the documentation of the respective system. Similarly, Bowman et al. examined the Linux kernel in a multi-step process to recover its architecture. In their work [4], they present the intricacies of their findings regarding the File System subsystem of the Linux kernel.

Several automated approaches have been developed, simplifying the process of recovering the architecture of a software system. Hutchens and Basili [11] used clustering based on data bindings between Fortran procedures to decompose a software system into subsystems. Similarly, Mitchell and Mancoridis [17] developed Bunch, which utilizes clustering in order to decompose software projects into subsystems using a language-independent approach based on the structure and relations of the modules in the source code a software system.

6.3 Architectural Smells

Oizumi et al. [19] investigated the relation ship between groups of related code-level anomalies (referred to as *code-anomaly agglomerations*) and architectural issues. They found that using code-anomaly agglomerations indeed served as a better indicator of architectural issues than individual code smells and that some types of code-anomaly agglomerations are better suited to detect architectural issues than others. Fontana et al. [6] created a tool called Arcan in order to identify flawed structural dependencies among packages and classes of software projects written in Java. Cai and Kazman [5] developed DV8, which supports identifying several architectural smells as well as quantifying the maintenance cost associated with them.

7 Concluding Remarks

In this chapter, we briefly summarize our thesis and give an outlook on potential future work.

7.1 Conclusion

Architectural technical debt poses a grave threat to the longevity of a software system, greatly impacting its maintainability and, by extension, its developers. Among best practices during the software development life cycle, various measurement methods to detect architectural decay have been developed in order to curb the negative consequences of amassing technical debt and to allow developers to take appropriate action. In this thesis, we investigated what data-flow can reveal about the architecture in a software project using graph clustering. To this end, we performed a comparison between architectural models based on frequent common changes (”co-changes”) and data-flow interactions between files. We conducted an experiment on five projects of medium size and compared both models in terms of similarity (RQ1) and architectural smell detection (RQ2). We relied on the MoJoFM effectiveness measure in order to assess the similarity of the models and found values ranging from 28-51% across all five projects, indicating a rather strong difference. For architectural smell detection, we made use of an established set of architectural smells and observed a similar performance of both models in terms of the absolute number of smells detected. However, we found noticeably different results with regard to which files participate in which architectural smells for three out of five projects. Based on our results, we conclude that the co-change based model and the model based on data-flow interactions differ rather significantly.

7.2 Future Work

Going forward, future work may repeat our experiment with a variety of projects of considerably greater size in order to further investigate the external validity of our approach. Additionally, it appears to be a worthwhile endeavor to verify that the patterns designed by Silva et al. are indeed indicative of increased maintenance effort when used in conjunction with our data-flow based model. Finally, future research may explore designing patterns specifically tailored to data-flow interactions in order to uncover detrimental architectural designs decisions using data-flow interactions.

Bibliography

[1] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. 2nd ed. SEI series in software engineering. Addison-Wesley, 2003. 528 pp.

[2] Terese Besker, Antonio Martini, and Jan Bosch. "The Pricey Bill of Technical Debt: When and by Whom will it be Paid?" In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017.

[3] Dirk Beyer and Andreas Noack. "Clustering Software Artifacts Based on Frequent Common Changes." In: *13th International Workshop on Program Comprehension (IWPC'05)*. IEEE, 2005, pp. 259–268.

[4] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. "Linux as a case study: its extracted software architecture." In: *Proceedings of the 21st international conference on Software engineering*. ACM, 1999.

[5] Yuanfang Cai and Rick Kazman. "DV8: Automated Architecture Analysis Tool Suites." In: *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE / ACM, 2019, pp. 53–54.

[6] Francesca Arcelli Fontana, Ilaria Pigazzini, Riccardo Roveda, Damian A. Tamburri, Marco Zanoni, and Elisabetta Di Nitto. "Arcan: A Tool for Architectural Smells Detection." In: *2017 IEEE International Conference on Software Architecture Workshops (ICSA Workshops)*. IEEE Computer Society, 2017, pp. 282–285.

[7] Francesca Arcelli Fontanaa, Valentina Lenarduzzi, Riccardo Roveda, and Davide Taibi. "Are architectural smells independent from code smells? An empirical study." In: *J. Syst. Softw.* 154 (2019), pp. 139–156.

[8] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. "Identifying Architectural Bad Smells." In: *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, 2009.

[9] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. "Toward a Catalogue of Architectural Bad Smells." In: *Architectures for Adaptive Software Systems*. Vol. 5581. Springer Berlin Heidelberg, 2009, pp. 146–162.

[10] Michael W Godfrey and Eric H S Lee. "Secrets from the Monster: Extracting Mozilla's Software Architecture." In: (2000).

[11] David H. Hutchens and Victor R. Basili. "System Structure Analysis: Clustering with Data Bindings." In: *IEEE Transactions on Software Engineering* SE-11.8 (1985), pp. 749–757.

[12] George Karypis, Eui-Hong Han, and Vipin Kumar. "Chameleon: Hierarchical Clustering Using Dynamic Modeling." In: *Computer* 32.8 (1999), pp. 68–75.

[13] Chris Lattner. "LLVM." In: *The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks*. Brown, Amy and Wilson, Greg, 2011.

[14] Duc Minh Le, Carlos Carrillo, Rafael Capilla, and Nenad Medvidovic. "Relating Architectural Decay and Sustainability of Software Systems." In: *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. IEEE, 2016.

[15] Duc Minh Le, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. "An Empirical Study of Architectural Decay in Open-Source Software." In: *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2018.

[16] Antonio Martini, Francesca Arcelli Fontana, Andrea Biaggi, and Riccardo Roveda. "Identifying and Prioritizing Architectural Debt Through Architectural Smells: A Case Study in a Large Software Company." In: *12th European Conference on Software Architecture, ECSA*. Springer International Publishing, 2018, pp. 320–335.

[17] Brian S. Mitchell and Spiros Mancoridis. "On the Automatic Modularization of Software Systems Using the Bunch Tool." In: *IEEE Transactions on Software Engineering* 32.3 (2006), pp. 193–208.

[18] Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, and Qiong Feng. "Architecture Anti-Patterns: Automatically Detectable Violations of Design Principles." In: *IEEE Transactions on Software Engineering* 47.5 (2021), pp. 1008–1028.

[19] Willian Nalepa Oizumi, Alessandro F. Garcia, Thelma Elita Colanzi, Manuele Ferreira, and Arndt von Staa. "On the relationship of code-anomaly agglomerations and architectural problems." In: *Journal of Software Engineering Research and Development* 3 (2015), pp. 1–22.

[20] Darius Sas, Paris Avgeriou, Ronald Kruizinga, and Ruben Scheedler. "Exploring the Relation Between Co-changes and Architectural Smells." In: *SN Computer Science* 2.1 (2021), p. 13.

[21] Florian Sattler. "Understanding Variability in Space and Time." PhD thesis. Saarland University, 2024.

[22] Florian Sattler, Sebastian Böhm, Philipp Dominik Schubert, Norbert Siegmund, and Sven Apel. "SEAL: Integrating Program Analysis and Repository Mining." In: *ACM Transactions on Software Engineering and Methodology* 32.5 (2023), pp. 1–34.

[23] Luciana L. Silva, Marco Tulio Valente, and Marcelo A. Maia. "Co-change patterns: A large scale empirical study." In: *Journal of Systems and Software* 152 (2019), pp. 196–214.

[24] Luciana Lourdes Silva, Marco Tulio Valente, and Marcelo De A. Maia. "Co-change Clusters: Extraction and Application on Assessing Software Modularity." In: 12 (2015), pp. 96–131.

[25] Richard N. Taylor, Nenad Medvidovič, and Eric M. Dashofy. *Software architecture: foundations, theory, and practice*. Wiley, 2010. 712 pp.

[26] 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*. ACM, 2014.

[27] Zhihua Wen and Vassilios Tzerpos. "An effectiveness measure for software clustering algorithms." In: *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004*. IEEE, 2004.