Bachelor's Thesis

COMMIT-FEATURE INTERACTIONS: ANALYZING STRUCTURAL AND DATAFLOW RELATIONS BETWEEN COMMITS AND FEATURES

SIMON STEUER

October 6, 2023

Advisor:
Sebastian Böhm Chair of Software Engineering

Examiners:
Prof. Dr. Sven Apel Chair of Software Engineering
Andreas Zeller CISPA Helmholtz Center for Information Security

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

UNIVERSITÄT DES SAARLANDES

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

Short summary of the contents in English...a great guide by Kent Beck how to write good abstracts can be found here:

https://plg.uwaterloo.ca/~migod/research/beckOOPSLA.html

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

1 INTRODUCTION

1.1 GOAL OF THIS THESIS

1.2 OVERVIEW

2

BACKGROUND

In this section, we summarize previous research on the topic of code regions and interaction analysis. Thereby we give defintions of terms and dicuss concepts that are fundamental to our work. In the next chapter, we use the introduced definitions and concepts to explain and investigate structural as well as dataflow-based commit-feature interactions.

2.1 CODE REGIONS

Software Programs consist of source code lines that are translated into an intermediate representation (IR) upon compilation. IR accurately represents the source-code information and is utilized in many code improvement and transformation techniqes such as code-optimization. Furthermore, static program analyses are conducted on top of IR or some sort of representation using IR.
Code regions are comprised of IR instructions that are consecutive in their control-flow. They are used to represent abstract entities of a software project, such as commits and features. For this, each code region carries some kind of variability information, detailing its meaning. It should be noted that there can be several code regions with the same meaning scattered across the program. We dicuss two kinds of variablity information, namely commit and feature variability information, allowing us to define two separate code regions.
Commits are used within a version control system to represent the latest source code changes in its respective repository. Inside a repository revision, a commit encompasses all source code lines that were added or last changed by it.

Definition 1 (commit regions)**.** The set of all consecutive instructions, that stem from source code lines belonging to a commit, is called a *commit region.*

As the source code lines changed by a commit are not necessarily contigious, a commit can have many commit regions inside a program. To properly address the entire set of these commit regions within a program, we introduce the term regions of a commit. We say that the *regions of a commit* are the set of all commit regions that stem from source code lines of the commit.

In general, features are program parts implementing specific functionality. In this work we focus on features that are modelled with the help of configuration variables that specify whether the functionality of a feature should be active or not. Inside a program, configuration variables decide whether instructions, performing a feature's intended functionality, get executed. Detecting these control-flow dependencies is achieved by deploying extended static taint analyses, such as Lotrack[3].

Definition 2 (feature regions)**.** The set of all consecutive instructions, whose execution depends on a configuration variable belonging to a feature, is called a *feature region.*

Similarly to commits, features can have many feature regions inside a program as the instructions implementing their functionalities can be scattered across the program. Here, we also say that the *regions of a feature* are the set of all feature regions that stem from a feature's configuration variable.

2.2 INTERACTION ANALYSIS

In the previous chapter, we have already shown how different abstract entities of a software project, such as features and commits, can be assigned concrete representations within a program. We can use interaction analyses to infer interactions between them by computing interactions between their concrete representations. This can improve our understanding of them and give us insights on how they are used inside software projects. An important component of computing interactions between code regions is the concept of interaction relations between them introduced by Sattler [4]. As we investigate structural and dataflow-based interactions, we make use of structural interaction ($\circledcirc$) and dataflow interaction ($\rightsquigarrow$) relations in this work.
The structural interaction relation between two code regions r_1 and r_2 is defined as follows:

Definition 3 (structural interaction relation)**.** The interaction relation $r_1 \circledcirc r_2$ evaluates to true if at least one instruction that is part of r_1 is also part of r_2.

The dataflow interaction relation between two code regions r_1 and r_2 is defined as:

Definition 4 (dataflow interaction relation)**.** The 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.

Essential to the research conducted in this paper is the interaction analysis created by Sattler [4]. Their interaction analysis tool VaRA is implemented on top of LLVM and PhASAR. In their novel approach SEAL[6], VaRA is used to determine dataflow interactions between commits. This is accomplished by computing dataflow interactions between the respective regions of commits. Their approach for this will be shortly discussed here, however we advice their paper for a more thorough explanation.
The first step of their approach is to annonate code by mapping information about its regions to the compiler's intermediate representation. This information is added to the LLVM-IR instructions during the construction of the IR. In their paper, Sattler et al. [6] focus on commit regions, which contain information about the commit's hash and its respective repository. The commit region of an instruction is extracted from the commit that last changed the source code line the instruction stems from. Determining said commit is accomplished by accessing repository meta-data.
The second analysis step involves the actual computation of the interactions. For this, Sattler et al. implemented a special, inter-procedural taint analysis. It's able to track data flows between the commit regions of a given target program. For this, information about which commit region affected it, is mapped onto data and tracked along program flow. In this context, we would like to introduce the concept of taints, specifically commit taints. Taints are used to to give information about which code regions have affected an instruction through dataflow. For example, an instruction is tainted by a commit if its taint stems from a commit region. It

follows, that two commits, with their respective commit regions r_1 and r_2, interact through dataflow, when an instruction is part of one's commit region, r_2, while being tainted by r_1. Consequently data produced by the commit region r_1 flows as input to an instruction within the commit region r_2, matching the dataflow interaction relation.

3

COMMIT-FEATURE INTERACTIONS

In this section, we define structural and dataflow-based commit-feature interactions as well as properties related to them. Furthermore, their meaning and relationship inside a software project is explained here. In the Overview chapter, we discussed what purpose commits and features serve in a software project. Commits are used to add new changes, whereas features are cohesive entities in a program implementing a specific functionality. In this work, structural interactions are used to investigate how commits implement features and their functionality. In addition, dataflow interactions are examined to gain additional knowledge on how new changes to a program, in the form of commits, affect features.

3.1 STRUCTURAL CFIS

In the background chapter, we dicussed the concept of Code Regions, especially commit and feature regions. Logically, we speak of structural interactions between features and commits when their respective regions structurally interact. This structural interaction between code regions occurs when at least one instruction is part of both regions. This is the case when code regions structurally interact through the structural interaction relation ($\circledcirc$).

Definition 5. A commit C with its commit regions r_{1C}, r_{2C},... and a feature F with its feature regions r_{1F}, r_{2F},... structurally interact, if at least one commit region r_{iC} and one feature region r_{jF} structurally interact with each other, i.e. $r_{iC} \circledcirc r_{jF}$.

Structural CFIs carry an important meaning, namely that the commit of the interaction was used to implement or change functionality of the feature of said interaction. This can be seen when looking at an instruction accounting for a structural interaction, as it is both part of a commit as well as a feature region. From the definition of commit regions, it follows that the instruction stems from a source-code line that was last changed by the region's respective commit. From the definition of feature regions, we also know that the instruction implements functionality of the feature region's respective feature. Thus, the commit of a structural interaction was used to extend or change the code implementing the feature of that interaction. Following this, we can say that the commits, a feature structurally interacts with, implement the entire functionality of the feature. That is because each source-code line of a git-repository was introduced by a commit and can only belong to a single commit. Thus every instruction, including those part of feature regions, is annotated by exactly one commit region.

Knowing the commits used to implement a feature allows us to determine the authors that developed it. This is made possible by simply linking the commits, a feature structurally interacts with, to their respective authors. By determining the authors of a feature, we can achieve a deeper insight into its development than solely focusing on the commits that implemented it.

3.2 DATAFLOW-BASED CFIS

Determining which commits affect a feature through dataflow can reveal additional interactions between commits and features that cannot be discovered with a structural analysis. Especially dataflows that span over multiple files and many lines of code might be difficult for a programmer to be aware of. Employing VaRA's dataflow analysis, that is dicussed in section 3.6, facilitates the detection of these dataflow interactions.
Commit interactions based on dataflow were explained in the Interaction Analysis section and can be considered as precursors to dataflow-based commit-feature interactions. Similarly to commits interacting with other commits through dataflow, commits interact with features through dataflow, when there exists dataflow from a commit to a feature region. This means that data allocared or changed within a commit region flows as input to an instruction located inside a feature region. This pattern can also be matched to the dataflow interaction relation ($\rightsquigarrow$) when defining dataflow-based commit-feature interactions.

Definition 6. A commit C with its commit regions r_{1C}, r_{2C},... and a feature F with its feature regions r_{1F}, r_{2F},... interact through dataflow, if at least one commit region r_{iC} interacts with a feature region r_{jF} through dataflow, i.e. $r_{iC} \rightsquigarrow r_{jF}$.

```
1. int calc(int val) {                          ▷ d93df4a
2.     int ret = val + 5;                       ▷ 7edb283
3.     if (FeatureDouble) {                     ▷ fc3a17d    ▷ FeatureDouble
4.         ret = ret * 2;                       ▷ fc3a17d    ▷ FeatureDouble
5.     }                                        ▷ fc3a17d    ▷ FeatureDouble
6.     return ret;                              ▷ d93df4a
7. }                                            ▷ d93df4a
```

Listing 3.1: This code example contains both structural as well as dataflow-based commit feature interactions. Commit `fc3a17d` implements the functionality of `FeatureDouble` for this function. It follows that a structural commit-feature interaction can be found between them, as their respective commit and feature regions structurally interact. Commit `7edb283` introduces the variable `ret` that is later used inside the feature region of `FeatureDouble`. This accounts for a commit-feature interaction through dataflow, as data that was produced within a commit region is used as input by an instruction belonging to a feature region of `FeatureDouble` later on in the program.

3.3 COMBINATION OF CFIS

When investigating dataflow-based CFIs, it is important to be aware of the fact that structural CFIs heavily coincide with them. This means that whenever commits and features structurally interact, they are likely to interact through dataflow as well. As our structural analysis has already discovered that these commits and features interact with each other, we are more interested in commit-feature interactions that can only be detected by a dataflow analysis. That this relationship between structural and dataflow interactions exists becomes clear when looking at an instruction accounting for a structural CFI. From definition 5, we know that the instruction belongs to a commit region of the interaction's respective commit. It follows that data changed inside the instruction produces commit taints for instructions that use the data

as input. Now, if instructions that use the data as input are also part of a feature region of the interaction's respective feature, the commit and feature of the structural interaction will also interact through dataflow. However, such dataflow is very likely to occur, as features are functional units, whose instructions build and depend upon each other. Knowing this we can differentiate between dataflow-based CFIs that occur within the regions of a feature and those where data flows from outside the regions of a feature into it. From prior explanations, it follows that this differentiation can be accomplished by simply checking whether a commit that influences a feature through dataflow, also structurally interacts with it.

3.4 FEATURE SIZE

When examining commit-feature interactions in a project, it is helpful to have a measure that can estimate the size of a feature. We can use such a measure to compare features with each other and, thus, put the number of their interactions into perspective. Considering our implementation, it makes most sense to define the size of a feature as the number of instructions implementing its functionality inside a program. As the instructions inside the regions of a feature implement its functionality, we can the define the size of a feature as follows:

Definition 7. The *size* of a feature is the number of instructions that are part of its feature regions.

It's possible to calculate the defined size of a feature by calculating the number of instructions in which structural CFIs occur. That is, because every instruction that is part of a feature region accounts for a structural commit-feature interaction, as every instruction is part of exactly one commit region as shown in the beginning of this section. It follows, that we do not miss any instructions that are part of feature regions and do not count any such instruction more than once.

3.5 NESTING DEGREE OF STRUCTURAL CFIS

In contrast to commit regions, multiple regions of different features can appear in a single instruction. Here, we say that the number of feature regions inside an instruction equals its nesting degree. With increasing nesting degree of an instruction, it becomes less certain which purpose the instruction serves specifically. It could implement functionality of only a single feature present in the instruction's feature regions or any combination of them potentially. For an instruction with a nesting degree of one, we can be most certain about its purpose, namely implementing the functionality of the one feature present in the instruction. This observation also bears consequences in the way we look at and judge structural CFIs, as we can also define a nesting degree for them:

Definition 8. The *nesting degree* of a structural CFI is the minimum nesting degrees of all instructions the interaction occurs in.

It follows that, if for example, a commit and a feature only structurally interact inside instructions with a nestig degree of two or higher, the nesting degree of their structural CFI is two. If we now discover one more instruction, in which they interact structurally, and there

is no other feature region present inside the instruction, their structural CFI's nesting degree will now be one.
The nesting degree of a structural CFI is a useful measure to estimate the likelihood for which a commit was used to specifically change or implement functionality of a feature. We can be most certain for a structural CFI with nesting degree of one and become less certain for interactions with higher nesting degrees.

3.6 IMPLEMENTATION

The detection of structural as well as dataflow-based commit-feature interactions is implemented in VaRA [5]. Additionaly to commit regions, VaRA maps information about its feature regions onto the compiler's IR during its construction. Commit regions contain the hash and repository of their respective commits, whereas feature regions contain the name of the feature they originated from. VaRA also gives us access to every llvm-IR instruction of a program and its attached information. Thus, structural CFIs of a program can be collected by examining its compiled instructions. According to definition 5, we can store a structural interaction between a commit and a feature, if an instruction is part of a respective commit and feature region. For each such interaction we also save the number of instructions it occurs in. This is accomplished by incrementing its instruction counter if we happen to encounter a duplicate.

With this, we are also able to calculate the size of a feature based on our collected structural CFIs and their respective instruction counters. Following the explanations from section 3.4, we can compute the size of a feature as the sum over the instruction counters of all found structural CFIs the feature is part of.

In the Interaction Analysis section, we discussed the taint analysis deployed by VaRA. There, VaRA computes information about which code regions have affected an instruction through dataflow. Checking whether a taint stems from a commit region allows us to extract information about which commits have tainted an instruction. Thus, dataflow-based commit-feature interactions can also be collected on instruction level. According to definition 6, we can store a dataflow-based interaction between a commit and a feature, if an instruction has a respective commit taint while belonging to a respective feature region. Consequently said instruction uses data, that was changed by a commit region earlier in the program, as its input, while belonging to a feature region.

4 METHODOLOGY

The purpose of this chapter is to first formulate the research questions that we examine in our work and then propose our method of answering them.

4.1 RESEARCH QUESTIONS

In the Interaction Analysis chapter we dicussed the different meanings of structural and dataflow-based interactions. With this knowledge, we can answer many interesting research topics. These topics include patterns in feature development and usage of commits therin as well as findings about how likely seemingly unrelated commits are to affect features inside a program.

RQ1: How do commits and features structurally interact with each other?

We intend to research two main properties which already provide a lot of insight into the development process of features and best practices of commits therein.
Firstly, we examine the number of commits features interact with structurally. This gives us a direct estimate on how many commits were used in the development of a feature. Our analysis also allows us to measure the size of feature, which can put the number of commits used to implement a feature into perspective. We refine our investigation by factoring in the nesting degrees 3.5 of our collected structural CFIs. This allows us to differentiate between the DEFINITE and POTENTIAL size of a feature as well as most likely and potentially implementing commits.
Secondly, we want to examine how many features a commit interacts with structurally, i.e. how many features a commit usually changes. This is especially interesting when considering best practices surrounding the usage of commits. It is preferred to keep commits atomic[1] meaning they should only deal with a single concern. As different features implement separate functionalities, it's unlikely for a commit to change several features while dealing with the same concern. Transferring this to our work, high quality commits should mostly change a single feature. Acquiring data on this issue might show how strictly this policy is enforced in the development of features across different projects. Here, differentiating between the nesting degrees of structural interactions provides us with more informative results. We can form a lower bound for the number of features a commit usually changes by only considering structural interactions with a nesting degree of one. This way we only take into account commits for which we are certain that they were used to specifically implement functionality of the feature .

RQ2: How do commits interact with features through dataflow?

Investigating dataflow inside a program can unveil interactions between its entities that were previously hidden from programmers. This can help them understand the extent to which different parts of a program influence each other. Furthermore, deploying the introduced analysis in a direct manner could ensure improvements in the daily life of a developer. An example for this in our case could be the faciliation of finding the cause of and subsequently fixing bugs in features. Bugs occuring in certain features could be traced back to the commits responsible for them by factoring in recent commits affecting said features through dataflow. Previous research has laid the groundwork for researching dataflow interactions between different abstract entities of a program. While it has shown a wide range of interesting use-cases, it has focused solely on dataflow between commits. That's why we aim to provide first insights into the properties of dataflow-based CFIs. Specifically, we investigate how connected commits and features are by analyzing the number of features a commit usually affects through dataflow. Knowing what fraction of all commits contributing code to a project are part of dataflow-based interactions can show how often new commits affect the data of a feature. Regarding this, it is worth considering that commits constituting code of a feature are very likely to influence said feature through dataflow, as dicussed in section 3.3. Since structural interactions coinciding with dataflow interactions are so obvious, programmers are also much more aware of them. Depending on the prevalence of feature regions in a project's code space, this could heavily skew the data in one direction, as a large portion of all dataflow interactions would stem from these obvious interactions. Therefore we especially focus on commits that affect features through dataflow, while not constituting code of these features. Logically, programmers are less aware that changes introduced with these commits might affect the data of seemingly unrelated features.
- dataflow-based CFI allow us to examine another interesting property, namely the number of commits affecting a feature through dataflow.
- here, we differentiate between commits INSIDE AND OUTSIDE of features, i.e. commits that either do or don't structurally interact with them
- examine the proportion of inside to outside commits with dataflow interactions => gauge whether one could be used as a predictor for the other or whether there is no relation there (also where most of the dataflow interactions stem from)
- another relationship worth investigating, is the relation between the size of a feature and the number of outside and inside commits interacting with it through dataflow
- determining to what extent feature size is the driving factor in this, could tell us whether it is worth considering other possible properties of features responsible for the number of commits affecting its data

RQ3: How do authors interact with features?

- A simple but promising way to extract more information out of the collected CFIs is to link the interaction's commits to their respective authors
- thus, we are able to investigate how authors interact with features both structurally and through dataflow
- similarly to the number of commits implementing a feature, we can now calculate the number

of authors implementing a feature
- the same goes for commits interacting with features through dataflow, for which we can now determine the authors contributing these commits
- here, we only consider outside commits, i.e. commits not constituting code of the feature, as we have already considered all inside commits when investigating the developers implementing a feature
- we are especially interested in how many authors exclusively interact with a feature through dataflow in comparison to the number of authors implementing it
- this lets us answer the question whether the developers that implement features are also the ones mainly responsible for changes affecting them through dataflow
- finally, we relate both types of author interactions to the respective size of a feature
- determine whether the size of a feature is the main factor in the amount of developers needed to implement it
- (serve as advice for software companies now how to allocate programmers on to-be implemented features)
- dataflow: can tell develeopers whether they should care most about the size of a feature when paying attention to which features their introduced changes might affect

Usually there are many programmers working on the same software project, implementing different features, sometimes alone, sometimes with the help of colleagues. We shine some light on the exact statistics of this by combining structural commit-feature interactions with high-level repository information. One major property we investigate is the number of authors implementing a feature, where considering the size of a feature could help put this data into perspective. The collected results could serve as advice for software companies on how to allocate programmers on to-be implemented features.
- similarly to structural CFIs, we link the commits of dataflow-based CFIs to their respective authors
- here, we only consider outside commits, i.e. commits not constituting code of the feature, as we have already considered all inside commits when investigating the previous property
- furthermore, we are especially interested in how many authors exclusively interact with a feature through dataflow in comparison to the number of authors implementing it
- this lets us answer the question whether the developers that implement features are also the ones mainly responsible for changes affecting them through dataflow

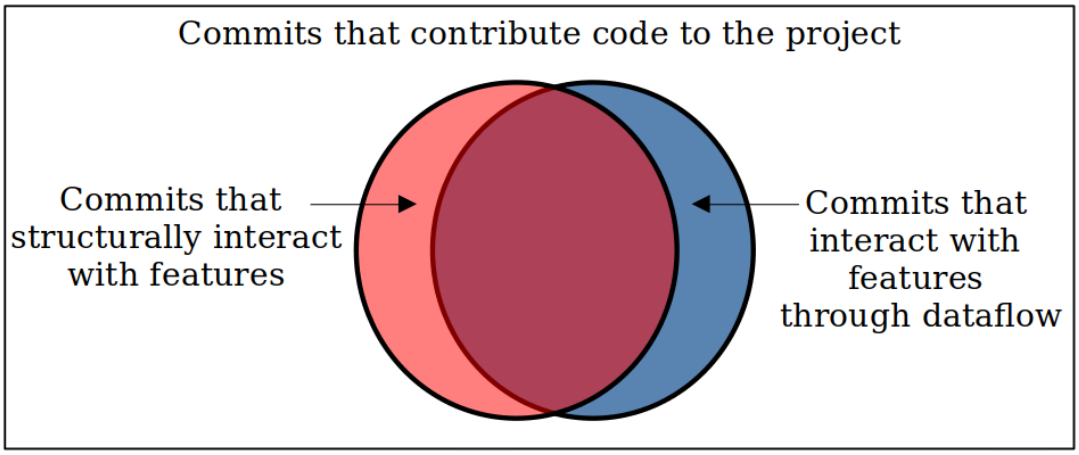

Figure 4.1: Kinds of commits in software projects investigated in this work. In the first two **RQs** we have discussed different kinds of commits and the ways in which they interact with features. Figure 1 showcases them in a venn diagram and illustrates the dependencies and divisions between them.

4.2 OPERATIONALIZATION

Here, we explain how the proposed RQs can be answered. The general experiment process is the same for all RQs. At first, we collect data comprimising all structural or dataflow-based CFIs by creating reports of a specific type for a chosen software project. The collected data is then processed in order to gain information for each commit or feature in the project, such as the number of interacting commits and size of a feature. The processed data is used to calculate statistical information, such as the mean and variance, or the strength of a correlation. To facilitate a faster and better understanding of the processed data and calculated statistics, we display them graphically via distribution or regression plots. The projects we investigate in this work, for example xz and GZIP, are of a small size and are used in a compression domain. This choice is based on the fact, that our research intends to only lay basic groundwork, where smaller projects can already offer a lot of insight. Since we only investigate a few projects, we chose them to be of a similar domain, such that a comparison between them makes more sense. In the following sections, we explain our method of investigation in more detail for each RQ.

RQ1: How do commits and features structurally interact with each other?

For this RQ, we examine reports comprising structural commit-feature interactions. From the collected data, we extract the size and the number of commits a feature structurally interacts with as well as the number of features a commit has structural interactions with. Concerning the first property of structural CFIs mentioned by us, we calculate the mean and variance of the number of commits used to implement a feature. This gives us an overview of how many commits were used during a feature's development and tells us how much this number varies between different features. A regression analysis on the relation between the size of a feature and the number of commits used to implement it, allows us to compute the strength of their

correlation. From the commits structurally interacting with features, we calculate the average number of features that a commit changes. Using the calculated average, we can determine if commits are primarily used to change just one feature. Additionally, we determine what fraction of commits that structurally interact with features do so with more than one feature, allowing us to see how likely commits are to affect code of only a single or more features. Here, filtering outliers can help produce more sensible data, as commits responsible for refactoring could change many features, while not implementing any functionality.

RQ2: How do commits interact with features through dataflow?

The projects investigated for dataflow-based CFIs are the same projects as investigated for structural CFIs. This choice gives us more insight into a single project and allows us to combine both analysis results as will be discussed below. For this RQ, we consider all commits that currently contribute code to the repository, which we can extract from high level repository information of the project. We process the collected data, such that, for each commit, we save how many features they interact with through dataflow. Here, we also indicate whether a commit structurally interacts with a feature, which we can simply check by examining the according structural report. With this information, we are able to separate commits into ones that implement parts of a feature and those that do not. We have already dicussed that commits used to implement a feature are much more likely to account for dataflow interactions. We provide evidence for this by comparing the average amount of features the two types of commits affect through dataflow. Besides that we compute what fraction of commits have dataflow-based interactions with other features, once for all commits, once for commits that are part of a feature and once for commits that aren't part of a feature. This shows how often commits affect features through dataflow based on their intended purpose, e.g. actively changing the functionality of a feature or changing something seemingly unrelated to a feature. With the data collected for each commit, we also plot the distribution of how many features they interact with. This lets us recognize potential outliers, for example commits with much more dataflow interactions than the mean would suggest. To check whether feature size is the driving factor in the number of outside commits affecting a feature through dataflow, we compute how strongly the two are correlated. While we already know the size of each feature from RQ1, it's still neccesary to calculate, for each feature, which commits have dataflow-based interaction with them and subsequently filtering the commits that are part of a feature.

RQ3: How do authors interact with features?

Here, we also examine the same projects as in the previous RQs. That way we can reuse data produced in RQ1 to map each feature to the authors that implemented it. For RQ1 we have already mapped each feature to the commits it interacts with, i.e. that contribute code to it. It's possible to extract the authors of these commits by accessing high-level repository information. This directly gives us the authors that implemented a feature. With the processed data, we are able to calculate the average number of authors used to implement a feature and plot their distribution for a sound overview. The size of each feature has also been calculated

to answer the previous RQs. With this information we aim to correlate the size of a feature with the amount of authors that implemented it in a regression analysis.

5 EVALUATION

This chapter evaluates the thesis core claims.

5.1 RESULTS

In this section, present the results of your thesis.

5.2 DISCUSSION

In this section, discuss your results.

5.3 THREATS TO VALIDITY

In this section, discuss the threats to internal and external validity.

6 RELATED WORK

Interactions between features [2, 3] and interactions between commits [6] have already been used to answer many research questions surrounding software projects. However, investigating feature interactions has been around for a long time, while examining commit interactions is a more recent phenomenon.
In an article published in 2023, Sattler et al. [6] analysed several open-source projects with their novel approach, SEAL. SEAL merges low-level data-flow with high-level repository information in the form of commit interactions. The paper shows the importance of a combination of low-level program analysis and high-level repository mining techniques by discussing research problems that neither analysis can answer on its own. For example SEAL is able to detect commits that are central in the dependency structure of a program[6]. This was used to identify small commits affecting central code that would normally not be considered impactful to a program. Furthermore, they investigated author interactions at a dataflow level with the help of commit interactions. Thus, they can identify interactions between developers that cannot be detected by a purely syntactical approach. They found that, especially in smaller projects, there often exists one main developer authoring the majority of commits[6] and thus accounting for most author interactions logically. It was also explained how SEAL makes it possible to relate occurences of bad programming practices to developers. This is accomplished by SEAL enriching program analyses with computed repository information.
Lillack et al. [3] first implemented the automatic detection of features inside programs by tracking their load-time configuration options along program flow. In our research, we also focus on features who are configured via configuration options, which we call configuration variables. Their analysis tool Lotrack can detect which configuration options must be activated in order for certain code segments, implementing a feature's functionality, to be executed. They evaluated Lotrack on numerous real-world Android and Java applications and observed a high accuracy for the predicted code execution constraints[3].
Referencing this paper Kolesnikov et al. [2] published a case study on the relation of external and internal feature interactions. Internal feature interactions are control-flow feature interactions that can be detected through static program analysis as mentioned above. They concluded that considering internal feature interactions could potentially help predict external, performance feature interactions[2].

7 CONCLUDING REMARKS

7.1 CONCLUSION

7.2 FUTURE WORK

APPENDIX

This is the Appendix. Add further sections for your appendices here.

BIBLIOGRAPHY

[1] Christopher Hundhausen, Adam Carter, Phillip Conrad, Ahsun Tariq, and Olusola Adesope. "Evaluating Commit, Issue and Product Quality in Team Software Development Projects." In: SIGCSE '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 108–114. ISBN: 9781450380621. DOI: 10.1145/3408877.3432362. URL: https://doi.org/10.1145/3408877.3432362.

[2] Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Sven Apel. "On the relation of external and internal feature interactions: A case study." In: *arXiv preprint arXiv:1712.07440* (2017).

[3] Max Lillack, Christian Kästner, and Eric Bodden. "Tracking load-time configuration options." In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 2014, pp. 445–456.

[4] Florian Sattler. "Understanding Variability in Space and Time." To appear. dissertation. Saarland University, 2023.

[5] Florian Sattler. *VaRA is an analysis framework that enables users to build static and dynamic analyses for analyzing high-level concepts using advanced compiler and analysis technology in the background.* https://vara.readthedocs.io/en/vara-dev/ [Accessed: (24.05.2023)]. 2023.

[6] Florian Sattler, Sebastian Böhm, Philipp Dominik Schubert, Norbert Siegmund, and Sven Apel. *SEAL: Integrating Program Analysis and Repository Mining*. 2023.