Bachelor's Thesis

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

SIMON STEUER

December 29, 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

In software engineering, features play a central role in implementing functionality of applications and making them configurable. Their code is maintained in Git repositories, where commits are used to introduce the latest source-code changes, thus gradually building the overall application and its features. To allow for a deeper understanding of commits and features as well as their interplay, we connect both entities in the form of commit-feature interactions (CFIs). For this, we extend the interaction analysis tool VaRA to implement the detection of structural and dataflow-based CFIs. While the former provides information about the direct involvement of commits in implementing features, the latter can uncover seemingly unrelated commits still influencing their functionality.

Instead of focusing on the number of features a commit changes, we study the responsiblities a commit deals with in relation features giving us a more accurate reflection of its purpose in feature development. Specifically, we argue that a commit still deals with a single responsibility when it affects code of a nested feature or code controlling the interplay of several features. We derive a best practice for commits, namely that they should usually have only one feature-related responsibility, which we find to be enforced for the majority of commits (>69%).

Furthermore, we choose a novel approach to fully assess the interplay between commits and features by being the first to study dataflow interactions between the two entities. We find that a substantial fraction of a project's commits interact with features through dataflow, although the respective fraction varies strongly between the examined projects ($11 - 37\%$). Said fraction also contains a signifcant number of commits who are directly related to the features they affect through dataflow, as their code is, at least partially, located inside the respective features. However, our goal of employing a dataflow analysis is to uncover interactions between seemingly *unrelated* commits and features, which is why we distinguish between commits located either inside or outside of features. This allowed us to clearly underline the importance of a dataflow analysis, as the majority of investigated features are mostly affected by outside commits.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

CFI commit-feature interaction

VaRA variability-aware region analyzer

IR intermediate representation

1 INTRODUCTION

Features play an important part in modern programming, which shows in program paradigms such as feature-oriented software development (FOSD), where they are used to implement specific functionalities, can be activated or deactivated and thus add configurability to software systems. As commits are used to contribute new source-code to a repository, it follows that commits can structurally interact with features by introducing source-code implementing their functionality. Furthermore, seemingly unrelated code, that was changed or added by commits, might influence soure-code of a feature through dataflow. We aim to give initial insights into these topics by investigating commit-feature interactions (CFIs).

Within a program, there exist many different abstract entities, such as commits and features, each serving different responsibilites. For a better understanding of them, it is advantageous to know whether they interact with or among each other. In the context of CFIs, bugs occuring in a feature could be linked to the latest commits affecting data of said feature, and consequently the authors responsible for these bugs. Especially for dataflow spanning over multiple files and many lines of code, it might be difficult to determine the respective interactions by studying the program yourself. To enable an automatic detection of these interactions and allow for a complete overview of them inside a software project, Sattler created the interaction analysis tool VaRA [10].

In a prior study, Sattler et al. [11] used VaRA to examine interactions between commits showing that research on this topic can be applied to improve many aspects of software development. For example, their research allowed for a deeper understanding of interactions between developers by linking commit interactions to their respective authors [11]. Similarly to commit interactions, internal feature interactions are dependencies between features at source-code level. While they have been the topic of some studies [3], the main intention was to utilize them as a predictor for external, e.g. performance interactions [12] of features.

Similarly to this thesis, previous studies have also investigated interactions between commits and features, particularly structural interactions to gain insights on the life cycle of features [5]. There, Michelon et al. [5] found that features are frequently changed by commits including substantial modifications to their code motivating that structural CFIs are a research topic worthy of further investigation. A major difference between their and our work is that we examine projects where features are implemented via run-time configuration variables, while their research has focused on projects where variability is implemented through the C Preprocessor. Not only does this allow for the investigation of a new set of projects, but they way features are implemented might also be different in these projects. Besides that, our work also qualifies as unqiue and novel as we, to the best of our knowledge, are the first to investigate dataflow interactions between commits and features. Specifically, we speak of dataflow-based CFIs when there exists dataflow from the code representations of commits to that of features. Thus, commits of these interactions introduced changes in the program affecting data, which is later used inside a feature. CFIs based on dataflow examine the program on a deeper layer, allowing us to detect additional interactions missed by structural CFIs. By

combining dataflow-based and structural interactions, we can specify whether dataflow stems from outside or from inside the code constituting a feature, thus producing more informative data.

In a second analysis step, we follow the discussed approach of Sattler et al. by enriching CFIs with high-level repository information [11]. In the same way in which we examine the influence of commits on features, we can now examine how the authors contributing these commits interact with features. As with commits, we say that developers can affect features structurally or through dataflow. There are some important differences though, as a developer can be the author of multiple commits, which can in turn affect features in different ways. Thus, developers implementing features might also be the ones contributing changes located outside of feature-code influencing them through dataflow.

1.1 GOAL OF THIS THESIS

The primary focus of this thesis is to gain an overview of how commits interact with features in software projects. Our goal is to lay basic groundwork regarding this subject, while leaving more detailed questions to future research. As previously mentioned, we investigate two types of commit-feature interactions, namely structural and dataflow-based CFIs. Using structural CFIs, we aim to reveal insights about the development process of features and usage of commits therein. We also investigate to what extend our dataflow analysis can reveal interactions that cannot be discovered with a purely syntactical analysis. By examining the interplay between the two types of CFIs, we hope to establish dependencies that will be considered in future studies. Finally, we provide first insights on interactions of authors with features to incentivize that this subject could be worthy of further investigation.

1.2 OVERVIEW

The Background chapter serves as an introduction to the concepts of code regions and interaction analysis, which are necessary to properly define structural and dataflow-based CFIs. Their definition then takes place in the Commit-Feature Interactions chapter, where we thouroughly dicuss their meaning in software projects as well as our Implementation of their detection in VaRA. We also define properties surrounding features and commits, that are used in the following chapters, like the size of a feature (3.4) and the feature-related concerns of a commit (3.6). The research questions of this paper are established in the Methodology chapter, in which we also explain their investigation. The Evaluation includes a detailed description of our results for the three posed research questions, followed by a separate discussion of the findings for each RQ. In the last section of Chapter 5, we address potential internal and external Threats to the Validtiy of this work. In Related Work, we summarize previous studies investigating interactions inbetween commits and inbetween features. We also cover topics such as the detection methods of feature-code and previous research on feature development. To finish the thesis, we summarize our results as well as their discussion and formulate thoughts adressing future research in our Concluding Remarks.

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 typically get translated into machine code by a compiler. Some compilers translate the source-code of a program into an intermediate representation (IR) before that to simplify the implementation of code optimizations. IR plays an important role in our work, as the static program analysis we use is conducted on top of IR.
Code regions can be used to connect abstract entities of a software project, such as commits or features, to IR [11]. They are comprised of IR instructions that are consecutive in their control-flow. For this, each code region carries some kind of tag, detailing its meaning. It should be noted that there can exist several code regions with the same tag scattered across the program. We dicuss two kinds of tags, namely commit and feature tags, allowing us to define two separate code regions.

COMMIT REGIONS Commits are used within a version control system to introduce 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. 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 all commit regions that stem from source-code lines of the commit.

FEATURE REGIONS In general, features are program parts implementing specific functionality. In this work, we focus on features that are modelled with the help of runtime 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 [4].

Definition 2. 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 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 these entities by computing interactions between their concrete representations (code regions). This can improve our understanding of them and give us insights on how these entities are used inside software projects. An important component of computing interactions between code regions is the concept of interaction relations between them as introduced by Sattler et al. [11]. 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. The structural 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. The dataflow 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.

In interaction analyses, the computation of interactions between different entities is based upon abstract interaction relations, like the ones we have defined above. Essential to the research conducted in this paper is the interaction analysis created by Sattler [9]. Their interaction analysis tool VaRA is implemented on top of LLVM and PhASAR. In their novel approach SEAL [11], VaRA is used to determine dataflow interactions between commits. This is accomplished by computing dataflow interactions between the respective regions of commits according to the dataflow interaction relation. 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 code generation. In their paper, Sattler et al. [11] focus on commit regions, which contain information about the commit's hash and its respective repository. The commit region of an instruction is determined by the commit that last changed the source-code line the instruction stems from. During the IR's construction, the commit of each source-code line is determined by accessing repository meta-data, after which the instructions stemming from each line are annotated with the hash of the respective commit [11].

The second analysis step involves the actual computation of the interactions. For this, Sattler et al. implemented a special, inter-procedural taint analysis. It is 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. This is done by propagating taints, specifically commit taints, through the program. 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 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 Background 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 knowledge on how seemingly unrelated changes, in the form of commits, affect features.

3.1 STRUCTURAL CFIS

In Chapter 2, 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 implement 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, which is dicussed in Section 3.8, facilitates the detection of these dataflow interactions.
Commit interactions based on dataflow were explained in Section 2.2 and can be considered cousins of 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 allocated 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: Illustration of Structural and Dataflow-based CFIs

The above code snippet 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. It becomes clear that this relationship between structural and dataflow interactions exists 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 them. 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.

3.5 FEATURE OVERLAP

Often, multiple features are implemented in the same code space. These features might structurally interact with each other, meaning that the respective code is responsible for controlling the interplay between them. On the other hand, we might be dealing with a feature nested inside larger ones that encapsulate the code implementing functionality of the nested feature. This can occur due to various reasons, for example being a sub-feature of an overarching feature or depending on data generated by other features. In both cases, that of structurally interacting or nested features, the instructions stemming from the respective code are part of several feature regions. As our analysis cannot detect whether we are dealing with structurally interacting or nested features, it does not make sense for us to differentiate between the two cases. We say that in both cases we are dealing with, at least partially, *overlapping features*, as their respective regions overlap.

3.6 FEATURE-RELATED CONCERNS OF A COMMIT

If a developer introduces changes to code containing overlapping features, the according commit will also produce structural CFIs with all of the respective features. In the case of structurally interacting features, the commit does affect code implementing functionality of both features. However, the commit technically only deals with a single concern in relation to features, as it does not implement functionality of either feature separately and rather a

combination of both features. In the case of code belonging to a nested feature the commit only deals with a single concern as well, since the affected code only changes functionality of one feature. From the features the commit structurally interacts with, we would assume that it changes functionality of multiple features, while it only does so for one of them. In both cases of *feature overlap*, the produced structural CFIs do not accurately depict the responsibilites of the respective commit. We therefore say that a commit only deals with a single feature-related concern as long as the code its interactions stem from belong to the same set of features. In our analysis, we are working with instructions and therefore speak of the same code space for features when the instructions we are dealing with belong to regions of the same features. It follows that we can assign a commit a number of concerns according to its structural CFIs and the instructions they occur in:

Definition 8. The *feature-related concerns* of a commit are the different sets of features the commit structurally interacts with at the same time, i.e. in the same instructions.

Feature-related concerns should be used as a measure to gauge the responsibilities of a commit that are difficult to decipher due to feature overlap. An examplatory use-case is shown in the listing below.

```
26. if (Min) {                          ▷ b9ea2f3    ▷ Min
27.    val = min(x,y)                   ▷ b9ea2f3    ▷ Min
28.    if (CheckZero) {                 ▷ e37a1c8    ▷ Min ∧ CheckZero
29.       if (val == 0) {               ▷ e37a1c8    ▷ Min ∧ CheckZero
30.          throw Error();             ▷ e37a1c8    ▷ Min ∧ CheckZero
31.       }                             ▷ e37a1c8    ▷ Min ∧ CheckZero
32.    }                                ▷ e37a1c8    ▷ Min ∧ CheckZero
33. }                                   ▷ b9ea2f3    ▷ Min
```

Listing 3.2: Example Use-case for the Feature-related Concerns of a Commit

The above code-snippet illustrates how considering the feature-related concerns of a commit is more insightful than simply looking at the features it structurally interacts with. We can see that the commit e37a1c8 exclusively implements functionality of the CheckZero-feature. Still, it also structurally interacts with the Min-feature, as the code of CheckZero is nested inside of it. However, commit e37a1c8 only has a single feature-related concern, as it only affects source-code lines that encompass regions of the same set of features. If commit b9ea2f3 instead introduced the entire code-snippet, it would have two feature-related concerns. In both cases, the feature-related concerns accurately reflect the number of features that were implemented by a commit. At the same time, this is not true for the number of features a commit structurally interacts with.

3.7 FEATURE OVERLAP DEGREE

As dicussed in the two previous sections, feature overlap is an important issue worth taking into account. Instructions that are part of several feature regions will, by definition, also consitute the size of the respective features. At the same time, we cannot be certain that the instruction actually implements functionality for every one of them. Thus, the size of a feature

can encompass many instructions not implementing its functionality. Technically, we can only be certain of an instruction's purpose if it is exclusively part of a single feature region. In order to achieve a more accurate description of a feature's size, we introduce the concept of a feature's overlap degree:

Definition 9. The *overlap degree* of a feature is the fraction of its size where its respective regions do not appear exclusively.

Besides that, the overlap degree of a feature informs us about the frequency of feature overlap within it and, in combination with other features, the entire project.

3.8 IMPLEMENTATION

The detection of structural as well as dataflow-based commit-feature interactions is implemented in VaRA [10]. Additionaly to commit regions, VaRA maps information about its feature regions onto the compiler's IR. 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.

To accommodate later evaluations using feature-related concerns of a commit and feature overlap factors, we create reports featuring a more complex datatype than a sole structural CFI. For this, we specifically save which features a commit structurally interacts with at the same time, i.e. inside the same instructions. We know that an instruction is always part of exactly one commit region, but could possibly belong to any number of feature regions. According to Definition 5, we encounter a structural CFI, if an instruction belongs to at least one feature region. For every such instruction, we store the discussed interaction, as the commit and the features present in the instruction. For each 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. This allows us to calculate the size of a feature (Definition 7), as the sum over the instruction counters of all found interactions the feature is part of. To determine a feature's overlap degree (Definition 9), we have to additionally compute the number of instructions where its feature regions do not appear exclusively. The computation is identical to that of a feature's size with the sole difference being that we only consider interactions encompassing multiple features. According to its definition, the overlap degree of a feature can subsequently be calculated by dividing the just determined number of instructions by the feature's size.

In Section 2.2, 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.

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 Chapter 3, we dicussed the different meanings of structural and dataflow-based commit-feature interactions. With this knowledge, we investigate patterns in feature development and the usage of commits therein and how often seemingly unrelated commits affect features inside a program. In RQ1, we focus on research topcis that can be answered with structural CFIs, while we explore the additional information dataflow-based CFIs can provide in RQ2. In RQ3, we look into the use-cases of author-feature interactions, which we extract from our previously examined commit-feature interactions.

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

We research two main properties which already provide a lot of insight into the development process of features and best practices of commits therein.

Investigating Patterns around Feature Development

Firstly, we examine the number of commits features interact with structurally. This gives us an estimate on how many commits were used in the development of a feature and the according distribution among the features of a project. 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 overlap degrees of features (Definition 9). In sections 3.5 and 3.7, we discussed feature overlap and, in the case of it being caused by feature nesting, the risks it bears for our analysis, namely that a fraction of a feature's size can consist of instructions that do not implement its functionality. The same is true for commits structurally interacting with a feature, meaning that a portion of them possibly did not participate in its development, but in the development of other features. We therefore consider the frequency of feature overlap within a project, so that we can tell how accurately our initial data reflects the development of features.

Examining the Usage of Commits in Feature Development

Secondly, we investigate the number of responsibilites in relation to features a commit usually deals with by examining the features it structurally interacts with. This is especially interesting when considering best practices surrounding the usage of commits. It is preferred to keep commits atomic [2] meaning they should only have a single responsibility. Transferring this

to our work, high quality commits should mostly have single feature-related responsibility. Acquiring data on this issue might show how strictly this policy is enforced in the development of features across different projects. In Section 3.6, we discussed that, due to feature overlap, a commit can structurally interact with multiple features at the same time while only dealing with a single responsibility. For example, a commit could be introducing changes to a feature nested inside other features or coordinate the interplay between several features. To achieve a more accurate prediction for the responsibilites of a commit, we examine the number of a commit's feature-related concerns (Definition 8).

In our analysis, we should ideally filter commits whose purpose was refactoring code, since they do not fit our criteria of changing or adding functionality to a feature. Exceptionally large commits might be best considered for this, as this is where we expect most commits used for large-scale refactoring to appear. A qualitative review of these commits is still necessary to make a final decision on whether they should be filtered.

4.1.2 ***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. Previous studies have 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 is why we aim to provide first insights into the properties of dataflow-based CFIs.

The Proportion and Dependencies of Commits Affecting Features through Dataflow

Firstly, we investigate how connected commits and features are by analyzing the number of features a commit 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 the dependency dicussed in Section 3.3, that structural interactions heavily coincide with dataflow interactions. This implies that commits constituting code of a feature are very likely to influence said feature through dataflow as well. In Section 3.3, we also mention that the dataflow of commits not structurally interacting with a feature, must stem from outside the regions of the feature. From a developer's point of view, dataflow originating outside of a feature is less intentional and subsequently more interesting, than dataflow occuring inside the regions of a feature. Programmers are less aware that changes introduced with these commits might affect the data of seemingly unrelated features. Therefore, we intend to differentiate between dataflow interactions with an outside and those with an inside origin in our anaylsis. This allows us to especially focus on commits part of outside dataflow interactions and examine their proportion among all commits.

Understanding Features and the Commits Affecting them Through Dataflow

Dataflow-based CFIs 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 do not contribute code to them. We examine the ratio of outside to inside commits influencing a feature to gauge where most dataflow interactions of a feature stem from. Considering our previous assumptions, the ratio of outside to inside commits could decrease with increasing size of a feature. Smaller features are likely to structurally interact with less commits than their larger counterparts, resulting in them also interacting with less inside commits through dataflow. However, the number of outside commits affecting data of a feature might not be dependent on feature size to such an extent. This leads us to another relationship worth investigating, namely the relation between the size of a feature and the number of outside and inside commits interacting with it through dataflow. We examine the two commit kinds separately, as we already have a strong supposition for inside commits, but are less certain for outside commits. Determining to what extent feature size is the driving factor in this relation, could tell us whether it is worth considering other possible properties of features in future analyses. Discovering features with unusually many outside commits in relation to their size could also allow us to detect feature code that is central code for a project.

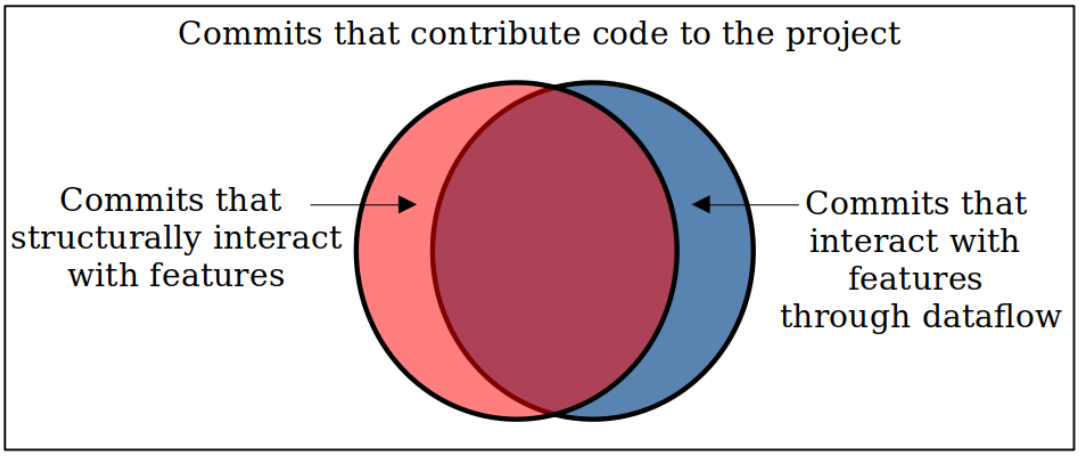

Figure 4.1: Illustration of Different Commit Kinds

In the first two RQs we have discussed different kinds of commits and the ways in which they interact with features. The above figure showcases them in a venn diagram and illustrates the dependencies and divisions between them.

4.1.3 ***RQ3**: How do authors interact with features?*

A simple yet 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 that participated in its development. The same applies to commits interacting with features through dataflow,

Table 4.1: The examined projects including relevant information for them

Projects	Projet Size (LOC)	History Size	Investigated Features
xz	48450	1866	10
gzip[1]	8121	605	14
bzip2	6866	168	8
lrzip	17647	978	3

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, since 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 its developers. This lets us determine 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. Regarding structural interactions, this allows us to determine whether a feature's extent in the source-code of a project is driving factor in the amount of authors needed to implement it. Software companies could use evidence on this issue as advice on how to allocate programmers on to-be implemented features. Findings on the investigated dataflow interactions of authors could tell developers that their changes might affect small and, at first sight neglegible, features surprisingly often. We also have a look at the specific functionality of features since their purpose could also play an important role in the number of authors interacting with 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 complex structural CFIs (see Section 3.8) 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 bar or regression plots.

The projects we investigate in this work are listed in Table 4.1 with additional information for them. All of them 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.

1 we exclude GZIP's submodule GNULIB here since we only consider features of GZIP's main code

4.2.1 *Operationalization: RQ1*

For this RQ, we examine the interactions contained in the structural reports created with our Implementation in VaRA. As there could be vast differences between the projects, we also work with and present their respective data separately.

Investigating Patterns around Feature Development

From the collected structural reports, we first extract the number of structurally interacting commits and size for each feature. The datapoints of the two properties are shown in two respective bar plots, where the values for each feature are shown in a separate bar. The bars are labelled after the name of the feature they present, to allow for comparisons between the two plots and between different projects. For comparisons between the investigated projects, we also calculate the average number of interacting commits and size of a feature for each project. For a simpler recognition of the existing distribution, the features are sorted in an increasing order according to their value in the metric of the respective plot. This also allows us to see whether the rank of a feature in one metric is indicative of its rank in the other metric and vice versa. Additionally to the size of a feature, we also determine its overlap degree (Definition 9). The average overlap degree among the features of a project is subsequently shown in a table. The strength of a project's overall overlap degree can in turn give us a direct estimate on the frequency of feature overlap inside of it.

Finally, we relate the two investigated properties in a regression plot. Here, we show whether features increasing in size, in turn structurally interact with more commits, i.e. need more commits to be implemented. Therefore, the value of the x-axis shows the size of a feature, whereas the y-axis shows the number of interacting commits. Each feature is represented by a scatter point inside the plot combining its respective y-values from the two previous plots. A linear regression line is drawn in the plot matching the occuring scatter points, where a rising graph could already indicate a positive correlation. To check whether we are dealing with a statistically significant linear correlation, we compute the pearson correlation coefficient and its p-value[2] and display both values inside the regression plot. To confirm our initial suspection of a strong positive correlation, the according correlation coefficient must be close to one, while the p-value must be close to zero falling below our rejection interval of 97.5%.

Examining the Usage of Commits in Feature Development

For each commit part of at least one structural interaction, we determine the number of its feature-related concerns (Definition 8). For this, we examine the complex structural CFIs discussed in our Implementation. There, we increment the number of a commit's concerns upon finding a new set of features it structurally interacts with at the same time, i.e. within the same instructions. Following this, we give a comprehensive overview of our results in a histplot for every investigated project. The x-axis of the histogram shows the number of a commit's concerns, whereas the y-axis shows the number of commis for which the respective x-value matches. If 50 commits deal with a single concern inside a project, the y-value of

2 the p-value indicates the probability that a dataset would have produced a given correlation coefficient if the correlation coefficient was in fact zero

one will be 50 accordingly. Thus, we can quickly see to what extent our derived best practice surrounding commits in feature development is enforced in a project. The more commits are located at one in comparison to other x-values, the more commits are likely to have only dealt with a single responsibility in relation to features. To facilitate comparisons between projects, we choose the same x- and y-labels for every histogram. Commits with exceptionally many feature-related concerns are shown in an additional x-tick to avoid an overly long x-axis. Instead, we mention these commits explicitly during our evaluation and relate the number of their concerns to their specific purpose.

In a second analysis step, we filter exceptionally large commits, whose purpose we expect to be something else than implementing features, such as code-refactoring. Similiarly to the regions of a commit, we determine its size in a repository as the number of source-code lines that were last changed or added by it. We then apply tukey's fence[3] with a k-value of 3 to filter far outliers of commit sizes among commits interacting with features within their respective project. Afterwards, we perform a qualitative review of the filtered commits to check whether our initial suspicion about their purpose is correct. To asses the effect of filtering exceptionally large commits, we compute the average number of concerns within a project before and after the application of our restriction. Both averages are subsequently shown in a cross-project table so that we can compare them within a project and recognize a general trend among them.

4.2.2 *Operationalization: RQ2*

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.

Analyzing the Proportion and Dependencies of Commits Affecting Features through Dataflow

We do not consider LRZIP here, as we are only able to detect regions of 3 out of 10 features in the project. This means that we can only find a fraction of all commits interacting with features, resulting in significantly lower determined proportions.

A preliminary step of evaluating what fraction of commits affect features through dataflow, is determining which set of commits to consider in the first place. Logically, we should only consider commits that could potentially be part of a dataflow-based CFI. The only prerequesite for this is that commits are represented by commit regions inside a program. This is the case when there exists at least one source-code line that was last changed or added by them. We call the respective commits, *active* commits as their contributed code is still "active" in the repository. Following the calculation of the number of active commits in the projects, we begin examining the created dataflow reports. For each commit part of at least one dataflow-based CFI, we save the set of features it interacts with. By dividing the number of these commits by the number of all active commits, we can determine their proportion within their respective project. To allow for an overview of all projects, we show the calculated percentages in a bar plot.

3 tukey's fence is an outlier detector for one-dimensional number arrays

For the number of active commits part of dataflow interactions in GZIP, we do not consider commits belonging to its sub-module GNULIB. That is, because the active commits of GNULIB are also excluded when calculating the number of GZIP's active commits. This choice guarantees more accurate results in respect to our research question due to several reasons:

- GNULIB encompasses >50x more active commits than GZIP
- We only consider features from GZIP specifically
- The majority of commits part of CFIs stem from the main code of GZIP and not from GNULIB

Note: This is the only time we exclude commits of GNULIB.

To provide evidence for our claim that structural CFIs heavily coincide with dataflow-based CFIs, we now compute the proportion of commits with dataflow interactions among the commits with structural interactions. Given that the latter is significantly higher than the overall proportion of commits with dataflow interactions, our hypothesis will be confirmed. Alongside the percentage of commits part of dataflow-based CFIs, we also show the percentage of active commits that are part of structural CFIs in the first bar plot. As we expect most of the latter commits to be part of dataflow-based CFIs as well, the difference between their respective percentages indicates what proportion of commits interact with features exclusively through dataflow.

Following the broad overview of commits affecting features through dataflow, we intend to specify and differentiate between the origin of said dataflow. For each feature a commit interacts with through dataflow, we check whether they also structurally interact with each other. If they do structurally interact, we speak of an inside dataflow origin, otherwise we speak of an outside origin. Naturally, dataflow occuring inside the regions of a feature is more intentional and therefore less interesting, than dataflow originating outside its regions. In Section 3.3, we explain, that the dataflow of commits and features not structurally interacting with each other, must be outside dataflow. Even though the regions of a commit and regions of a feature must partially overlap in order for them to structurally interact, such a commit can still have regions not part of any feature regions. Therefore, we cannot be sure whether the dataflow of these interactions originates from outside or inside the regions of a feature. Due to the inherent properties of structural interactions, i.e. heavily coinciding with dataflow interactions, we assume their dataflow to originate inside the regions of a feature. Upon determining the dataflow origin of each CFI, we split the commits with dataflow interactions into three categories. Next to commits that only affect features either through inside or through outside dataflow, a commit can also interact with multiple features, once through inside and once through outside dataflow. For each project, we calculate the respective percentages of each category and present them in a stacked bar plot. Logically, each bar representing a project adds up to 100% as every commit falls into exactly one category. The commits falling into categories of at least partial outside dataflow origins are part of interactions, which we could only detect with our dataflow analysis. To quantify the according commits within a project, we calculate their number and percentage among a project's active commits.

Both bar plots are displayed next to each other in the same figure, where the sequence in which the projects are shown in both plots is determined by the proportion of commits with dataflow interactions. This allows for an easier comparison within a project, as its respective bars will have the same position in both plots. Statistical values, including the number of active commits and the probability for a commit to be part of a dataflow-based CFI given that it structurally interacts with features, are not included in the plots. As they can aid the understanding of the plots and add important context to them, we display the respective values in a table.

Exploring Features and the Commits Affecting them Through Dataflow

Intially, we determine the set of commits each feature interacts with through dataflow. The according information is extracted from the same dataflow reports used in the previous section of this RQ. Depending on whether the commit also structurally interacts with a feature, we split the set of commits for each feature into two categories. Outside commits are fully located outside the regions of a feature, meaning that their data flows into a feature. Inside commits are at least partially located inside a feature making it very likely that their dataflow occurs within the regions of a feature. Naturally, said dataflow affecting a feature is less interesting than dataflow stemming from seemingly unrelated commits. To gauge what commit kind makes up the majority of commits affecting features through dataflow, we present both values for each feature in a bar plot. For every examined project, we display the number of outside and inside commits with separate bars for the different features. To enable an exact comparison, we also calculate the median for both outside and inside commits of a project. We choose the median and not the mean, as we are especially interested in detecting a general trend across the features of a project and intend to avoid outliers skewing the data in one direction. Besides that, features with an exceptionally high number of interacting commits or a high proportion of commits of a certain kind are already shown in the bar plots. The features are explicitly named to allow for cross-project comparisons of certain types.

In a second analysis step, we now relate the size of a feature to the number of commits affecting it through dataflow. Here, we are interested in determining whether and to what extent there exists a positive linear relationship between the two. We employ two linear regression analyses, relating the size of a feature once to its outside and once to its inside commits. Similarly to our investigation in RQ1, the results are displayed in a regression plot for each project. In each regression plot, we also show the two computed pearson correlation coefficients and their respective p-values. We decide for a 97.5% rejection interval again, meaning that the p-values must be lower than 0.025 to provide enough evidence that the two datasets are indeed not un-correlated. Furthermore, the correlation coefficient must range close to 1 in order to warrant a strong positive correlation.

Lastly, we investigate our claim that smaller features have a higher proportion of outside to inside commits affecting it through dataflow. For this, we examine the relation between the size of a feature and the ratio of its outside to inside commits. Again, we create linear regression plots for each project and calculate the acccording pearson correlation coefficients and the respective p-values. We could also perform other regression analyses, for example a logarithmic regression, as they could fit our datapoints better. Following our explanations in Section 4.1, we expect the ratio of outside to inside commits to decrease with increasing

feature size. This would result in a negative linear correlation coefficient and a falling linear regression line. It remains to be seen whether the respective values are statistically significant across several projects.

4.2.3 *Operationalization: RQ3*

Technically, we perform the same analysis regarding features as discussed for the two previous RQs. The only difference is that we replace the commits interacting with a feature with their respective authors. To faciliate expressing this, we simply say that the respective authors interact with features based on a certain interaction type or that they are a feature's structural or dataflow authors. In Section 4.1, we mentioned that we only consider commits outside of features when examining the authors interacting with them through dataflow. The computation of these commits has already been explained in the operationalization of RQ2, which we reuse for this RQ. The number of authors that interact with a feature either structurally or through dataflow are presented in a bar plot. Similarly to the previous RQs, the results of each project are shown in separate plots with labeled ticks for each feature. We also calculate the mean number of dataflow and structural authors of a project to give concrete information from what type of interaction most author interactions stem from. In a third bar for each feature, we display the number of authors that exclusively interact with them through dataflow. There, we exclude authors who also structurally interact with the feature, i.e. the authors making up the first bar. This shows us how many authors affecting a feature can only be determined via our dataflow analysis. For the structural interactions of authors, we compute what fraction of them also affect the respective features through dataflow. This can tell us to what extent the developers implementing a feature are likely to introduce code changes whose data the feature uses later on. Additionally considering whether there are more structural than dataflow authors lets us determine whether the developers of a feature are also the ones mainly responsible for changes affecting them through dataflow.

Finally, we perform the same linear regression anaylsis as we have done for feature-sizes and the number of commits that interacting with them. By calculating the strength of their correlation, we are able to determine whether the presumed extent of a feature inside a project is an accurate predictor for the number of developers needed to implement it. The authors interacting with a feature through outside dataflow are also indirectly involved in its development, since the data stemming from their contributed commits is necessary for the functionality of the feature.

5 EVALUATION

This chapter evaluates the thesis core claims. At first, we describe the results of our three RQs in Section 5.1. Following this, we dicuss our results and to what extent they answer our posed research questions. In Section 5.3, we focus on internal and external threats to the validity of our findings and gathered data.

5.1 RESULTS

Here, we evaluate the results of each RQ separately. As in Chapter 4, we divide the first two RQs into two sub-sections each. Most of our results are represented by plots with at least one figure containing them per RQ and sub-section. Important statistical values of the examined projects, which we refer to in the text, are additionally shown in tables.

5.1.1 *Results: RQ1*

Examining the structural CFIs of different projects can give us insights into the patterns of feature development and the usage of commits therin. We first generally desribe our results with the help of Figure 5.1 and Figure 5.2 as well as the tables 5.1 and 5.2. We also explain particularly interesting results of individual commits and features by investigating their specific purpose and properties.

Patterns of Feature Development

In the first part of RQ1, we examine the number of commits involved in the development of a feature and relate it to its size. Figure 5.1 illustrates our results in three different plots for each project. Each row displays the results for one project with the name of the respective project being shown on the far left. In the first column, we show the number of structurally interacting commits for each feature in a bar plot. For all projects, we notice a wide distribution of structurally interacting commits between features. Numerous features, across all projects, interact with less than 5 commits suggesting that they need very little work to be implemented in comparison to other features. Both XZ and LRZIP have features structurally interacting with more than 20 commits, while GZIP even has four features that interact with more than 60 commits. With a mean of 29, the features of GZIP have by far the highest number of interacting commits on average. XZ, LRZIP and BZIP2 have much lower averages at 9, 16 and 4 respectively. As all projects are of a compression domain, we can find several features with the same general functionality across the examined projects. The verbosity-feature structurally interacts with the most commits in XZ and BZIP2, while the same is not the case for the verbose-feature of GZIP. Although GZIP encompasses the highest average number of interacting commits, its recursive-feature only interacts with 2 commits, while the recursive-feature of LRZIP interacts with 24 commits.

Table 5.1: Average feature overlap degree of all projects

Projects	Avg. Overlap Degree
xz	0.34
gzip	0.93
bzip2	0.64
lrzip	0.37

In the second column of Figure 5.1, we display the calculated feature sizes for each project. Again, we see a wide range of different feature sizes within all projects. Most notabely, there are large jumps between adjacent features, for example from to_stdout with a size of 450 to no_name with a size of 7300 in GZIP. Similarly to the number of interacting commits, GZIP has by far the highest average feature size at 2500. The average feature sizes for the other projects range from 190 for XZ to 390 for BZIP2. In Table 5.1, we display the average overlap degree of features for all projects. Feature overlap is the least common for XZ and LRZIP and most common for GZIP with an average of 0.93 indicating that the majority of features overlap with other features to large extents. This is also the case for the four, by far largest, features of GZIP, namely no_name, method, force and decompress. Specifically, 90 to 100% of the instructions constituting their size are identical among the mentioned features. It is therefore not surprising that most, particularly 66, of the commits interacting with each feature are identical for all four features. For them, it is highly doubtful whether the size of a feature and the number of interacting commits reflect the actual number of *implementing* commits and instructions of a specific feature.

Finally, the metrics used in the two previously dicussed plots are compared to each other using the regression plot of Figure 5.1. That is, we compare the size of a feature with the number of commits that structurally interact with it. Each blue dot represents a feature and its coordinates are identical to the values of the bars in the two previous columns. For both XZ and GZIP, we observe a strong positive correlation between the size of a feature and its number of structurally interacting commits. Their linear correlation coeffecients are close to 1 at 0.91 and 0.978 respectively with p-values smaller than 10^{-4}. This data provides strong evidence that the observed correlation is statistically significant. While BZIP2 and LRZIP have positive correlation coefficients of 0.585 and 0.968 respectively, their p-vlaues are relatively high at over 0.1. The high p-values can be explained by a lack of datapoints for LRZIP and conflicting datapoints for BZIP2. For example, both the opMode- and srcMode-feature of BZIP2 structurally interact with 6 commits, but encompass vastly different feature sizes. Overall, the two projects do not produce conclusive statistical evidence that the size of a feature and the number of interacting commits is positively correlated.

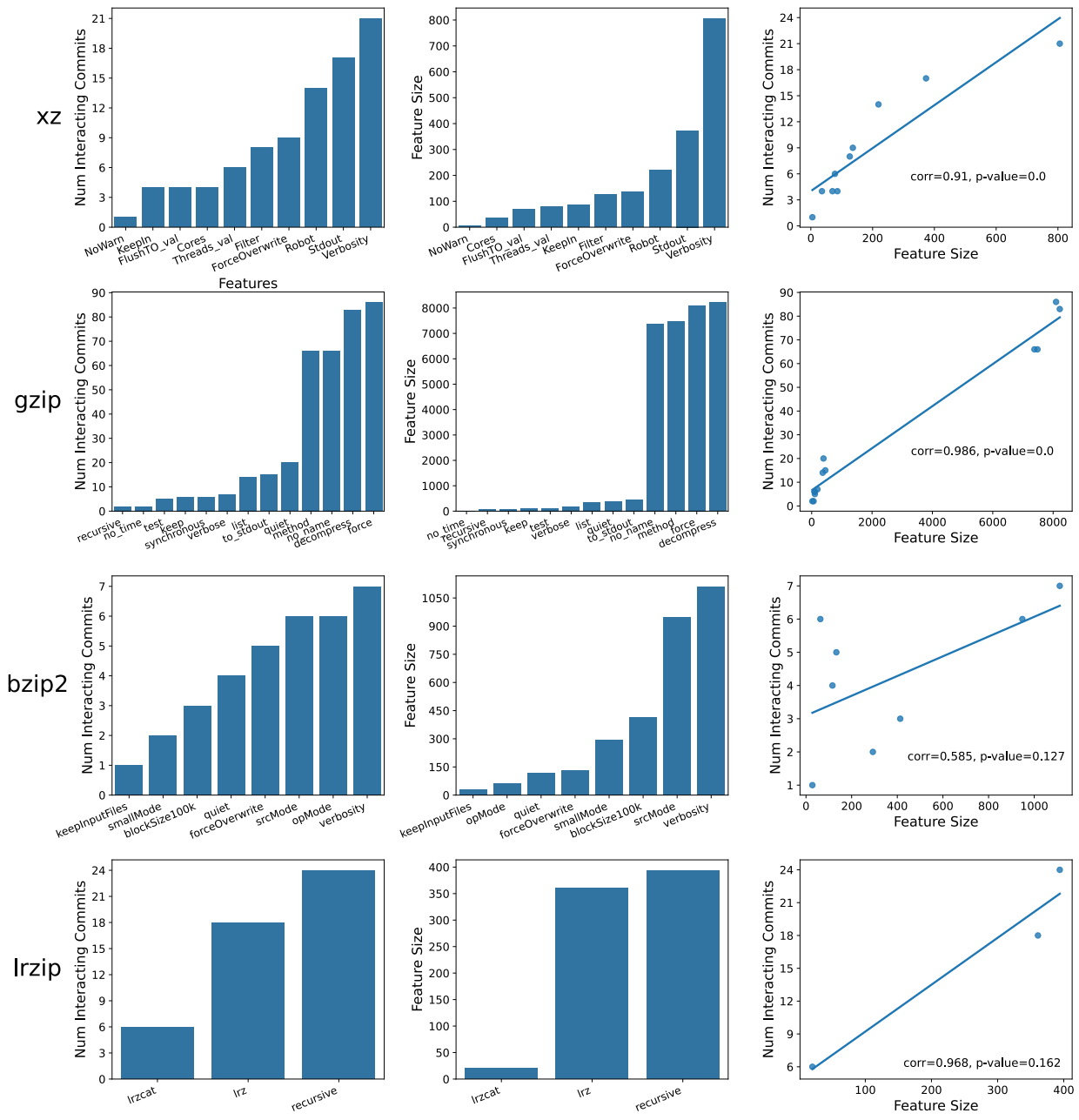

Figure 5.1: The distribution of feature sizes and structurally interacting commits of features and their linear correlation

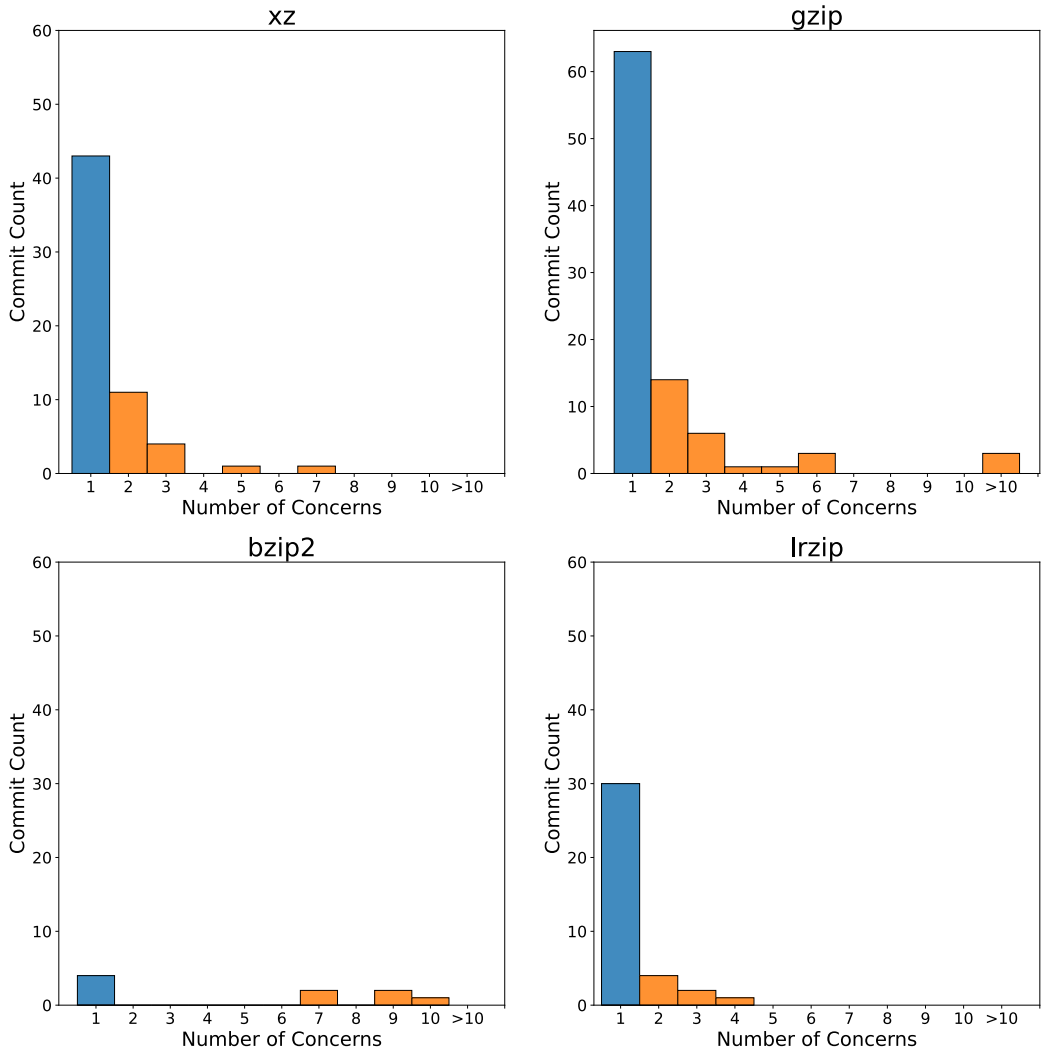

Figure 5.2: The distribution of the number of feature-related concerns for commits

Usage of Commits in Feature Development

As discussed in chapter 4.1, a best practice for commits in feature development is that a commit should generally deal with only one responsibility in relation to features. We investigate to what extent this holds in our investigated software projects by examining the number of feature related concerns of a commit. The distribution of how many commits have a certain number of concerns is displayed in Figure 5.2. xz, gzip and lrzip show similar distributions, which is why we discuss them now and bzip2 later. For xz, gzip and lrzip, we note that the majority of commits only have single feature-related concern. The according percentages for commits with a single concern among all considered commits are 72%, 69% and 81%. The commit count gradually drops with an increasing number of concerns until it subsides to 0 at an x-value of 7, 6 and 4 respectively. Interestingly, three commits of gzip have more than 10 concerns with the commit 33ae4134cc even encompassing 47 concerns. Said commit originally introduced the *gzip.c*-file containing the main-function among other things. The

Table 5.2: Average number of concerns for a commit before and after filtering large commits

Projects	Avg. Concerns	After Filtering Large Commits
xz	1.48	1.29
gzip	2.40	1.55
bzip2	5.11	4.62
lrzip	1.30	1.30

exceptionally high number of 33ae4134cc's concerns can be explained by the fact that many of GZIP's 14 features are partially implemented within *gzip.c*. Additionally, the code of features heavily overlaps with each other, which means that the instructions stemming from said code encompass many different combinations of feature regions. For BZIP2, there are much less commits used in feature development compared to the other projects with only 9 commits part of structural CFIs. The distribution consists of two clusters, the first being at a single concern and the second cluster being located between 7 and 10 concerns. The second cluster encompassing 5 commits is slightly bigger than the first cluster with a commit count of 4. Since we are only dealing with a few commits here that still produce quite interesting and unique data, we have a closer look at the purpose of these commits and relate it to the number of their concerns. We find that all commits part of the second cluster were used to publish new versions of BZIP2, always encompassing over 1000 additions. Three out of the four commits part of the first cluster introduced minor changes and bug fixes, never changing more than 30 source-code lines. The remaining commit updated BZIP2 to version 1.0.4 fixing minor bugs from the previous version with many of the added source-code lines being used for comments.

In Table 5.2, we display the average number of feature related concerns within a project both before and after filtering exceptionally large commits. The first column contains the project to which the averages in their respective row belong. In the following column, we show the initial averages before applying the mentioned restriction. XZ and LRZIP encompass the lowest average number of concerns at 1.48 and 1.3 respectively. Although the distribution of GZIP shown in Figure 5.2 is similar to those of the formerly mentioned projects, its initial average is comparatively high at 2.4. The main cause for this are the discussed three commits of GZIP with exceptionally many concerns. After filtering large commits within a project, we note that the averages of XZ, LRZIP and GZIP become more similar. While the average slightly decreased for XZ to 1.29, it stayed the same for LRZIP. The three dicussed commits of GZIP were part of the filtered commits, which explains the large drop of its average to 1.55. Unsurprisingly, the initial average of BZIP2 is by far the highest among all projects. Since most of BZIP2's commits are relatively large, this is also the reason why they are not excluded from our analysis. Here, only one commit with 9 concerns is removed, which lowers the average to 4.58. We also performed a qualitative analysis of the filtered commits to check whether our suspicion that they were mostly used to refactor code is true. We find that many of them were used to import releases of the project to Git. The fact that such commits exist is unsurprising considering that the investigated tools have been published before the release of Git or before Git was widespread. While their purpose might not be refactoring code, it does make sense to filter

Table 5.3: Relation between structural and dataflow-based CFIs

Projects	Active Commits	P(∃F: F ⇝ C | ∃F: C ◎ F)
xz	1039	0.883
bzip2	37	0.889
gzip	194	1.000

these commits as they are not accurately depicting the development process of the project itself or of its features. Only two out of the eighteen commits we filtered in all projects were actually used to refactor code, specifically moving contents between files or adapting the code to a newer version of C. For half of all exceptionally large commits we found no reason, neither refactoring code nor importing the project to Git, to be left out of our analysis.

5.1.2 *Results: RQ2*

By investigating dataflow-based CFIs, we are able to determine the proportion of commits with dataflow interactions in a project as well as the number of commits that affect a feature through dataflow. We factor in whether a dataflow interaction between a commit and a feature coincides with a structural interaction, to differentiate between dataflow stemming from outside or inside the regions of a feature. This way, we aim to underline our hypothesis that commits inside of features are more likely to affect them trough dataflow, allowing us to especially focus on commits with an outside dataflow origin. Our results regarding the first part of RQ2 are shown in Figure 5.3 and in the tables 5.3, 5.4 and 5.5. Figure 5.4 displays our results for the features of a project and the commits that affect them through dataflow.

Proportion and Dependencies of Commits Affecting Features through Dataflow

At first, we examine the fraction of commits affecting features through dataflow among the *active* commits of a project. There, we also evaluate the effects and dependencies of commits part of structural CFIs on said fraction. Following this, we evaluate the proportion of outside dataflow origins within the set of commits with dataflow interactions. This also allows us to quantify the commits whose interactions with features can only be discovered through our dataflow analysis.

The number of active commits are shown in the second column of Table 5.3 for each project. We determined xz to have the highest number of active commits at 1039, while bzip2 only has a tiny fraction of that at 37. In the first plot of Figure 5.3, the bars colored in red show what percentage of commits interact with features through dataflow. We notice that the respective percentages are vastly different from project to project. A significant portion of gzip's active commits are part of dataflow-based CFIs at 37.1%[1]. This percentage drops to about 27% for bzip2, with another large reduction to 11.3% for xz. This means that more than every third active commit in gzip, roughly every fourth in bzip2 and every ninth in xz affects features

1 this percentage is calculated only considering active commits of gzip and not its sub-module gnulib; we explained this choice more thouroughly in the Operationalization chapter

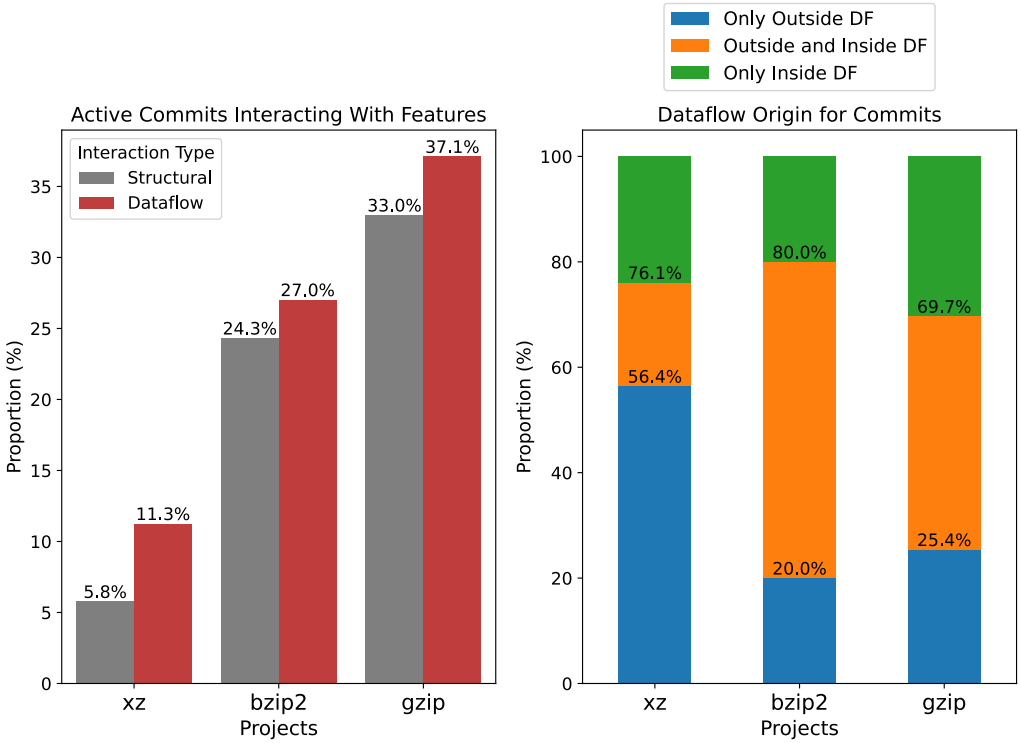

Figure 5.3: The proportion of commits with dataflow interactions and their dataflow origin

through dataflow. The bars colored in grey display the fraction of active commits structurally interacting with features. In addition to the obvious fact of how often commits are used to implement features, this also bears important consequences for the fraction of commits with dataflow interactions. Comparing the two bar-types, we notice that the percentage of commits with structural interactions is lower than the percentage of commits with dataflow interactions for each project. This phenomenon can be explained considering the values presented in the third column of Table 5.3. There, we show the probablity for a commit to be part of any dataflow-based CFI, given that said commit is part of any structural CFI. We see that the probabilities are slightly below 90% for xz as well as BZIP2 and even at 100% for GZIP. This means that by only taking into account commits part of structural CFIs, we already encounter a lot of commits that affect features through dataflow. If we then consider the entire set of active commits, the number of commits with dataflow interactions will likely exceed the number of commits with structural interactions. This also implies that most, if not all, commits constituting the grey bar of a project in turn constitute parts of the red bar as well.

We explore the topic of different dataflow interaction types more thoroughly with our second plot, which focuses on the *origin* of dataflow. There, we have a closer look at the commits affecting features through dataflow and where their dataflow stems from[2]. In the second plot of Figure 5.3, we separate the commits with dataflow interactions into three categories based on dataflow origin. The first category, shown as the bar colored in blue, represents commits that only interact with features through outside dataflow. They make up

2 for GZIP, we now factor in every commit part of dataflow interactions; namely the commits stemming from its sub-module GNULIB as well

Table 5.4: Commits exclusively/partially interacting with features *only* through dataflow

Projects	Commit Count		Percentage of all active commits	
	Exclusive	Partial	Exclusive	Partial
xz	66	23	6.4	2.2
bzip2	2	6	5.4	16.4
gzip	8	46	4.1	23.7

the majority of commits for xz at 56.4%, but only 20% as well as 25.4% of commits for bzip2 and gzip respectively. The second category represents commits that interact with features through outside and inside dataflow. Logically, these commits affect at least two features through dataflow and only structurally interact with a subset of them. Surprsingly, they form the majority of commits for bzip2 and gzip at 60 and 44.3% respectively and around 20% of commits for xz. The proportion of commits that only interact with features through inside dataflow varies less between the projects. gzip has the highest percentage at 31.3%, bzip2 the lowest at 20% with xz inbetween the two at 23.9%.

Combining the data presented in both plots, we calculated the number and percentage of active commits in a project whose interactions with features were exclusively discovered by our dataflow analysis. The respective values are shown in Table 5.4 on the left-hand side of the second and third column. With a count of 66, most commits were revealed for xz, although its percentage remains rather small at 6.4% due to its high number of active commits. The percentages of gzip and bzip2 are comparatively small at 4.1 and 5.4% respectively. On the right-hand side of the columns in Table 5.4, we display the number and percentage of commits where only a subset of their interactions with features was discovered exclusively by our dataflow analysis. These commits are most common in gzip with a count of 46 making up almost every fourth active commit at 23.7%. Similarly to the overall percentage of commits with dataflow interactions, they are less common in bzip2 at 16.4% and least common in xz at 2.2%. Interestingly, 23 and, thus, the vast majority of commits affecting features in gzip exclusively through dataflow stem from its sub-module gnulib. In contrast to this observation, only 8 commits stemming from gnulib affect features in gzip both through inside and outside dataflow. This constitutes a much lower number than the 46 commits of gzip's main code that do so. Overall, we find that commits exclusively interacting with features through dataflow are more likely to stem from the sub-module gnulib than gzip's main code, while the opposite is the case for commits partially interacting with features only through dataflow.

We notice that commits interacting with features through inside dataflow, have a high chance to also interact with *other* features through outside dataflow. By definition, all commits interacting with features through inside dataflow also structurally interact with them. From previous explanations, we also know these commits are almost identical to the entire set of commits with structural interactions. It follows that commits part of structural CFIs must also have a high, albeit slightly lower, chance to affect other features through outside dataflow. We compare this probability to the probability of any active commit to interact with features through outside dataflow in Table 5.5. The strongest contrast occurs for xz, where 41.7% of commits structurally interacting with features interact with other features through outside

Table 5.5: Relating outside dataflow to structural interactions

Projects	**P(∃F: F (out⇝) C | ∃F: F ◎ C)**	**P(∃F: F (out⇝) C)**
xz	0.417	0.086
bzip2	0.667	0.216
gzip	0.719	0.278

dataflow, making them 4.8 times more likely to do so compared to any active commit of xz at 8.6%. The contrast is slightly weaker for BZIP2 with a 3.1 times higher likelihood and according percentages of 66.7 and 21.6% respectively. The project GZIP has the weakest contrast at 2.6 and respective probabilites of 71.9 and 27.8%.

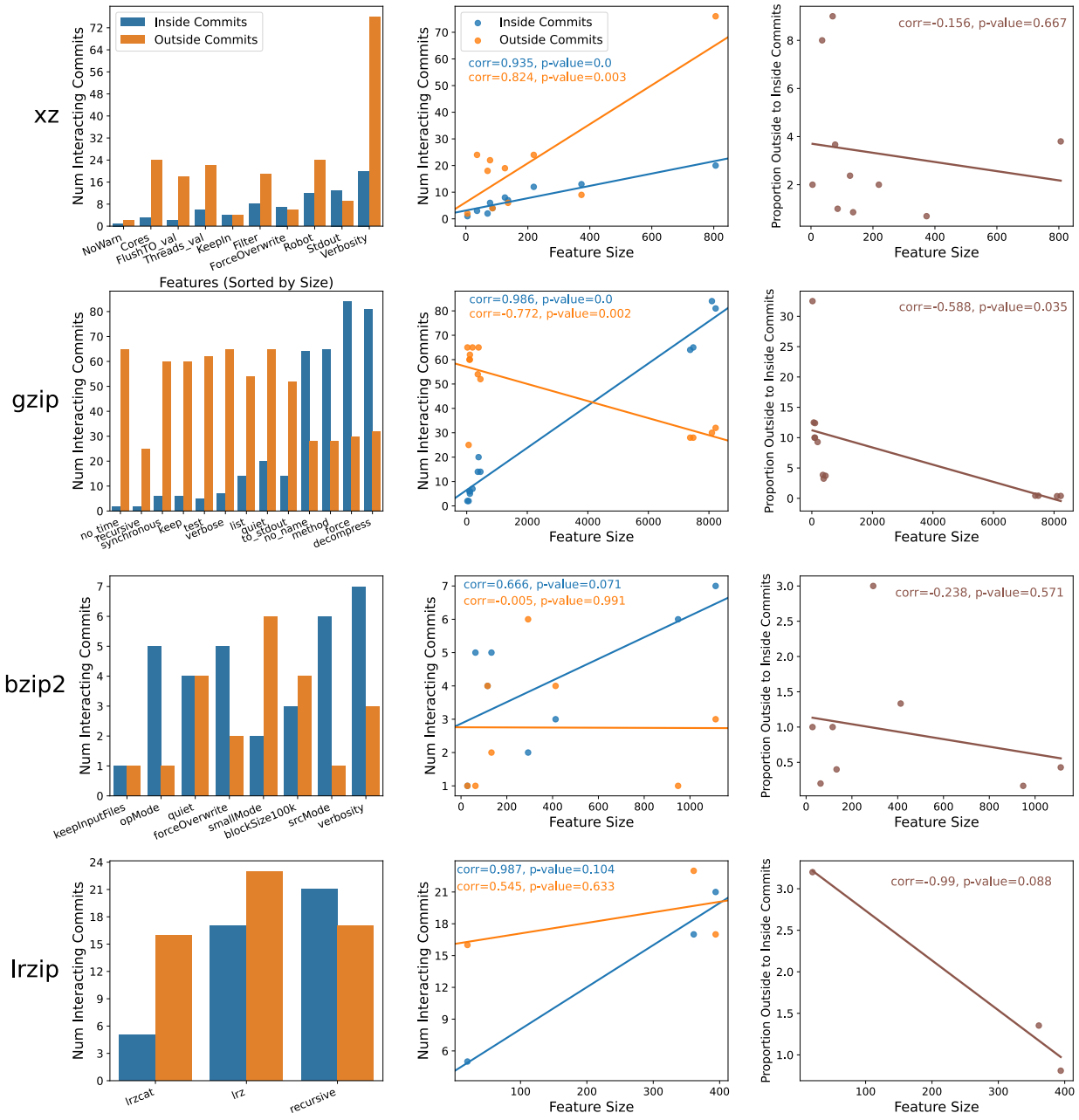

Figure 5.4: The distribution and proportion of outside and inside commits affecting a feature through dataflow and their linear correlation to the size of a feature

Understanding Features and the Commits Affecting them Through Dataflow

When examining the commits affecting a feature through dataflow, we differentiate between those located outside and those, at least partially, located inside of the feature they interact with. Investigating the ratio of outside and inside commits allows us to determine whether most dataflow interactions with commits occur inside or outside of the regions of a feature. In a second evaluation step, we also relate the two metrics to the size of a feature. Furthermore, we want to put our hypothesis, that the proportion of outside to inside commits of a feature decreases as its size increases, to the test.

Our results are shown in Figure 5.4 in three plots for each of our investigated projects. In column one of the respective figure, we display the number of outside and inside commits for each feature in a bar plot. We note that the majority of features in XZ, GZIP and LRZIP are affected by more outside than inside commits through dataflow. For XZ and GZIP this is underlined in median-values of 18.5 and 54 for outside, and 6.5 and 14 for inside commits, among their features. For features of XZ and GZIP with more outside than inside commits, we also notice very high ratios of outside to inside commits. They range from 2 to 9 in XZ with a mean of 4.4 and from 3 to 32 in GZIP with a mean of 10.8. For features with more inside than outside commits affecting them through dataflow, we determined much lower ratios and according ranges of inside to outside commits. They range from 1.17 to 1.44 in XZ with a mean of 1.3 and from 2.3 to 2.8 in GZIP with a mean of 2.5. At least for these two projects, the number of outside commits is often not only slightly higher than the number of inside commits, but instead exceeds it by many maginutes. The features of BZIP2 exhibit a different general behavior than the features of the projects discussed above. Specifically, 6 out of 8 features encompass more or at least the same number of inside than outside affecting them through dataflow. The range and maximum number of interacting commits is also much lower compared to the other examined projects.

In the second column of Figure 5.4, we compare the number of inside and the number of outside commits of a feature with its size in a regression plot. As suspected during the formulation of RQ2, we find a strong positive correlation regarding the inside commits of features across all projects. Especially, the correlation coefficients for XZ, GZIP and LRZIP range close to 1 with a minimum value of 0.935. XZ and GZIP encompass p-values of less than 10^{-4} supporting a statisitically significant correlation. The correlations regarding the outside commits affecting features through dataflow are vastly different from project to project. XZ is the only project with a strong positive correlation with a coefficient of 0.824. A low p-value of 0.003 outside of our rejection interval makes the data seem statistically significant at first. However, the dots of its regression plot are scattered rather randomly and the number of interacting commits generally does not increase with rising feature-size. We determined the Verbosity-feature to be the main reason for our calculated strong positive correlation. This becomes clear when we disregard Verbosity in our analysis, which results in the coefficient dropping close to 0 and the p-value rising to 1. GZIP encompasses a negative correlation coefficient of -0.772 and a p-value of 0.002 justifiying the existence of significant statistical evidence. The falling linear regression line is mainly formed by the two existing clusters, where one consists of four very large and the other cluster of nine relatively small features. The former cluster consists of features that each interact with around 30 outside commits through dataflow. The cluster consisting of features with small sizes encompasses higher

values between 50 and 65. The only feature of GZIP that cannot be assigned to a cluster is the recursive-feature, which rather fits into the first cluster according to its number of outside commits and into the second cluster according to its size. In BZIP2, the number of outside commits affecting a feature through dataflow and its size are un-correlated. A correlation coefficient of 0.545 and a high p-value of 0.633 do not provide significant statistical evidence for the existence of a correlation in LRZIP.

In the third column of Figure 5.4, we finally relate the size of a feature to its proportion of outside to inside commits. At first, we notice that the calculated correlation coefficients are negative across all projects. The fact that the coefficients of XZ and BZIP2 are close to 0 and the p-values correspondingly high is illustrated by the rather random distribution of dots in their respective regression plots. Their dataset should therefore not be regarded as correlated. Compared to the aforementioned projects, GZIP's p-value of 0.035 is very low, while it also has a lower correlation coefficient of -0.588. Similarly to GZIPS's previous plots, we can identify two clusters. A cluster of dots is again formed by the four largest features, encompassing proportions of outside to inside commits slightly below 0.5. The other cluster is located at the beginning of the x-axis with a wide spread across the y-axis, indicating its low feature-size and a wide range of proportions from 3 to 32. The coefficient of LRZIP is by far the smallest among all projects at -0.99, but the small number of features results in a p-value of 0.088. It follows that none of the determined negative correlations are statistically significant, as their p-values fall into our rejection interval of 97.5%.

5.1.3 *Results: RQ3*

Here, we examine the authors structurally interacting with features, i.e. those involved in their development, as well as the authors introducing commits affecting features through outside dataflow. As the sets of authors interacting with features structurally or through outside dataflow can overlap, we also investigate the number of authors that can only be discovered with our dataflow analysis. The respective values for a feature's structural, dataflow and unique dataflow authors are shown in the first column of Figure 5.5 for each project. In the second column of said figure, we also display our results regarding the comparison of the two metrics with the size of a feature.

The project XZ has mostly been developed by a single author, which is reflected in our results. As can be seen in Figure 5.5, all except one feature have been exclusively implemented by the same author. We can conclude that this also is the developer affecting features through outside dataflow, since no feature has unique dataflow authors. The features of BZIP2 mostly interact with one author structurally and one or two authors via dataflow. Except for the keepInputFiles-feature, our dataflow analysis discovered one unique interacting author for each feature. This changes for LRZIP, where our dataflow analysis reveals at least 2 and up to 5 unique authors for the three examined features. Around half of the authors structurally interacting with a feature in LRZIP also interact with it through outside dataflow. Overall, the linear regressions relating the size of a feature to the number of its structural or dataflow authors show no significant correlation for the three mentioned projects.

GZIP, on the other hand, provides us with much more comprehensive results. In GZIP's bar-plot in Figure 5.5, we can see that every feature has a large number of interacting authors. The combined number of unique authors that interact with a feature in any way can be

computed by the sum of its structural and its unique dataflow authors. This means that the lowest number of interacting authors is 7 for the recursive-feature and the highest 14 for the force-feature. On average, a feature has more dataflow than structural authors with respective averages of 9.1 and 5.7. The average number of unique dataflow authors is 5.9, which means that our dataflow analysis discovers many additional interacting authors for the features in GZIP. The number of structurally interacting authors, i.e. authors implementing a feature, range from 2 to 13 between the features of GZIP. There is less variance in the number of a feature's outside dataflow authors ranging from 7 to 12. Features with relatively few structural authors, have a comparatively high number of dataflow authors. Features with many structural authors, starting from the no_name-feature, have less outside dataflow authors. At the same time, the number of their unique dataflow authors is comparatively low, implying that most of them also structurally interact with the according features. The fact that the group of features with few structural authors has much more dataflow authors conditions them to have many unique dataflow authors. Still, more than 80% of their structural authors also interact with them through outside dataflow. This is not the case for the group of features with many structural authors, where this is the case for less than 45% of them. Overall, there is a 57% likelihood for the structural author of any feature to interact with the same feature through outside dataflow. Similarly to the bar-plot of GZIP, we notice two clusters when examining its regression plots in Figure 5.5. The first cluster is made up of features with few structural authors, whose respective sizes range between 20 and 450. The second cluster consists of the four features who encompass more than 10 structural authors and much higher feature-sizes ranging from 7400 to 8200. It is therefore unsurprising that we compute a strong positive correlation between the size of a feature and the number of its structurally interacting authors. The respective correlation coefficient is 0.98, while the p-value is smaller than 10^{-4} providing evidence for a statistically significant correlation. Regarding the dataflow authors of features, we note two similar clusters that consist of the same features of small and much larger sizes respectively. Now, the smaller features have slightly more dataflow authors centering around 10 than the much larger features with an average of 5. This shows itself in a strong negative correlation coefficient of -0.872 and a p-value of less than 10^{-4} falling out of our rejection interval of 97.5%.

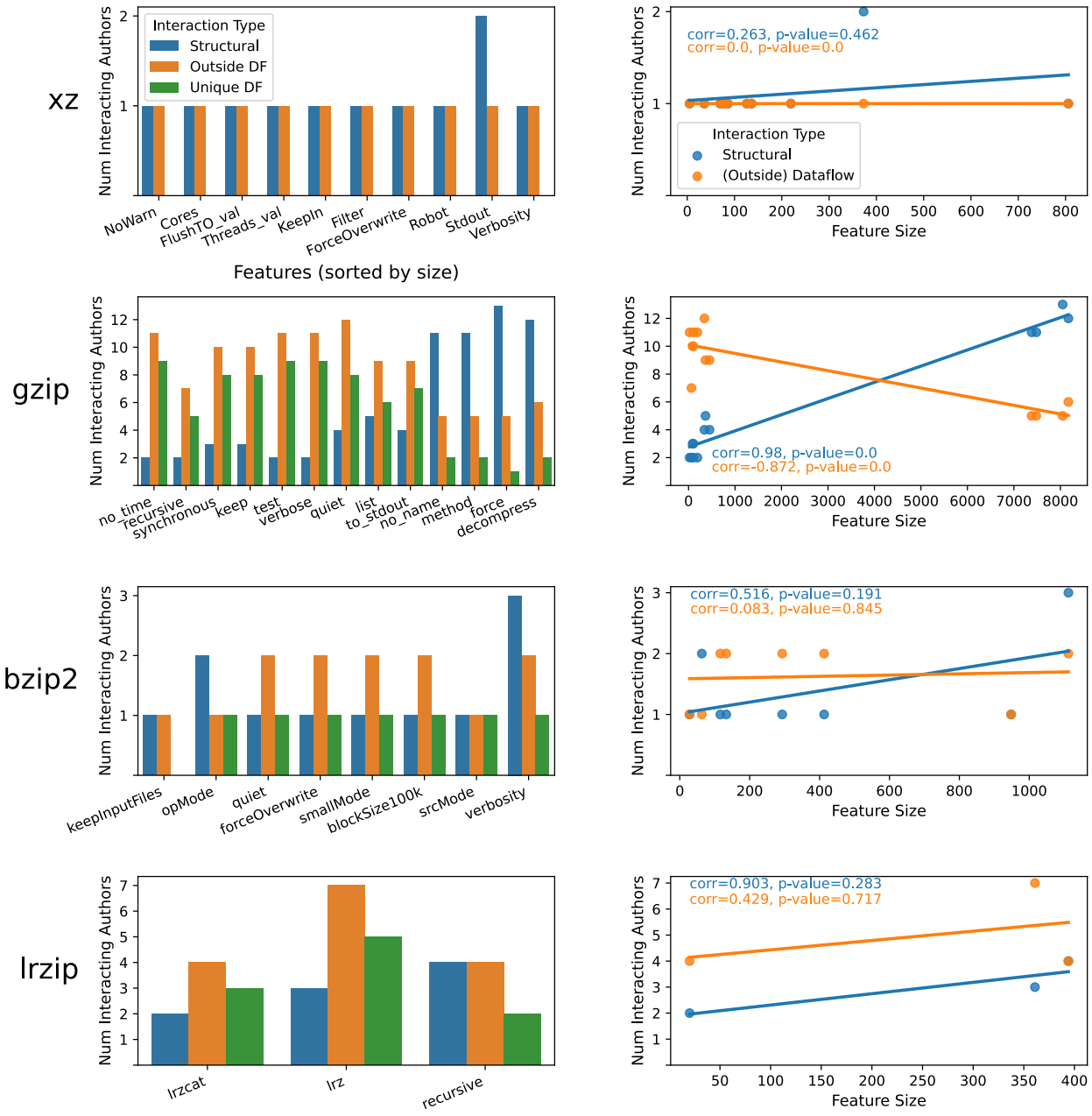

Figure 5.5: The distribution of structural, outside dataflow and unique dataflow authors of features and their linear correlation to the size of a feature

5.2 DISCUSSION

In this section, we first formulate whether and how the previously evaluated results can answer our research questions. We also relate findings between the RQs and their sub-sections to highlight similarities and differences between them. Besides that, we look at possible reasons for our results and further interesting thoughts related to them.

5.2.1 *Discussion: RQ1*

Generally, we have seen our expectations for possible results of RQ1 mostly confirmed. Here, we briefly formulate our most important results, which we discuss in more detail in the following sub-sections. Firstly, we find sufficient evidence to support the notion that the number of commits used to develop a feature is strongly positively correlated with the size of the code implementing its functionality. Secondly, we have high confidence that most commits, within our investigated projects, only deal with a single responsibility in relation to features.

Patterns around Feature Development

The features within the investigated projects exhibit a wide distribution of structurally interacting commits and sizes. We consider this as a clear indication that the number of commits as well as the extent of source-code used to implement a feature varies a lot between them. The range of least to most interacting commits and lowest to highest feature-size is hereby dependent on the specific project.

Determining the average overlap degree of features within a project showed us that, especially in the case of GZIP, feature overlap is more common than initially expected. Most of the computed sizes of GZIP's respective features are made up of instructions that are part of multiple feature regions besides its own. This makes it difficult to clearly interpret the data, as the number of instructions that actually implement a feature can differ greatly from our observed size. This phenonemon also bears consequences on they way commits are used in feature development and the CFIs resulting from this. If a developer wants to specifically change a feature nested inside other features, the according commit will necessarily produce structural CFIs with several features. This also casts some doubt how well the structural interactions of a feature with a high overlap degree accurately represent its development. A concrete example of this are the four, by far largest, features of GZIP. In our evaluation of RQ1, we have already explained that they share the vast majority of instructions constituting their size, as well as the commits that structurally interact with them. In these cases, it would be extremely helpful to be able to decide whether the respective instructions and commits are actually `implementing` commits and instructions of a particular feature. While solving this problem goes beyond the scope of this work, future research could produce more informative results by adressing and solving this subject.

We employ the pearson correlation coefficient to investigate the linear relationship between the size of a feature and the number of commits structurally interacting with it. While we find high correlation coefficients of 0.9 and above, the p-values of LRZIP and BZIP2 are not low enough to justify the data to be statistically significant. We know that p-values are highly

dependent on the number of datapoints, that is, the number of features of a project. Logically, projects with fewer features have less datapoints resulting in their p-values being quite high and falling into our rejection interval. With 10 and 14 features respectively, xz and gzip have by far the most datapoints and according p-values of less than 10^{-4} giving us most significant evidence for a strong positive correlation. Due to the above-mentioned problems caused by gzip's high overlap degree, the significance of the found strong positive correlation is somewhat weakened. On the other hand xz has a relatively low overlap degree, but still shows the same strong positive correlation. Therefore, we are still able to confirm our original thesis of a strong dependency between the size of the implementing code and the number of implementing commits of a feature.

A possibility to circumvent the problem of too few datapoints could be to combine features of all projects into one big dataset. Here, we are faced with the issue that the way commits are used among projects might differ drastically. In our case, the number of instructions per structurally interacting commit of a feature varies greatly from project to project. The respective values are 17 for xz, 48 for gzip, 87 for bzip2 and 13 for lrzip. This means that features of a similar size have, on average, vastly different numbers of structurally interacting commits across different projects. Upon these considerations, we are of the opinion that it makes more sense to test correlations within projects as we have done in this work.

Usage of Commits in Feature Development

In Section 3.6, we explained that, due to feature overlap, the number of features a commit structurally interacts with can be a faulty estimate of how many responsibilities in relation to features a commit usually deals with. To achieve a more accurate measure, we introduced the concept of feature-related concerns of a commit (Definition 8). There, we do not look at the structural CFIs of a commit separately, but rather focus on the entirety of them and the instructions they stem from. As long as the CFIs occur in the same feature related location, i.e. in instructions belonging to regions of the same set of features, we assume that the commit deals with a single responsibility. A concrete example of why the feature related concerns can be a more accurate estimate for the number of changed features is shown in listing 3.2. Excluding bzip2, roughly 70% of commits only have a single feature related concern in the remaining projects. Despite this, gzip's average is quite high at 2.4, in comparison to xz and lrzip with 1.48 and 1.3 respectively. It decreases to 1.55 after filtering large commits, such as commit 33ae4134cc with an exceptionally high number of 47 feature related concerns. Its purpose was to import the original *gzip.c*-file into the repository, which justifies the exclusion of the commit.

To summarize our results, most commits of xz, gzip and lrzip have only one feature-related concern with an average of less than 1.6 after filtering large commits. Given that the feature related concerns accurately reflect the number of responsibilites for most commits, we can confirm our initial thesis. Thus, the majority of commits that structurally interact with features only deal with a single responsibility in relation to them.

In contrast to three projects discussed above, bzip2 has shown to be a project whose results need to be treated more carefully. Only 9 commits structurally interact with its features, encompassing much less datapoints compared to the other investigated projects. 6 out of 9 commits were used to import the initial project to Git and iteratively update the code to newer

versions. These commits are not exemplary of the common development process of a project and its features in a Git repository. The actual development of the tool in the Git repository began after BZIP2 was finally updated to version 1.0.6. The remaining 3 commits introduce minor changes and small bug fixes to the respective features they interact with. This means that BZIP2's features were largely already implemented when the actual development within the repository started. Thus, the average number of concerns of a commit as well as their overall distribution are not good representatives of other projects developed in Git.

5.2.2 *Discussion: RQ2*

We are able to confirm our hypothesis discussion in Section 3.3 that structural CFIs heavily coincide data flow-based CFIs. This allows us to formulate a lower bound for the proportion of commits with dataflow interactions, namely the proportion of commits with structural interactions. While there exists a lot of variation in the proportion of commits affecting features through dataflow within the investigated projects, said proportion heavily depends on a project's extent of feature-code and the number of commits implementing said code. Furthermore, our dataflow analysis uncovers a significant amount of interactions between commits and features that we cannot discover with our previous structural analysis. We are also able to uncover that commits that already touch feature code have an increased probability of influencing other features via outside dataflow. We transfer this knowledge to features and the commits affecting them through dataflow by determining that many of their outside dataflow commits are also part of other features. In general, the majority of features are influenced by outside commits, whereas this dataflow often originates within other features. We find no statistically significant correlation between the number of outside commits of a feature and its size. This also applies to the proportion of outside to inside commits, although we suspect that more data points and examined projects could lead to more substantive evidence here.

Proportion and Dependencies of Commits Affecting Features through Dataflow

We have seen that the proportion of commits part of dataflow-based CFIs varies a lot between our investigated projects. While only 11% of active commits in xz affect features through dataflow, with 37%, a significantly higher proportion of active commits do so in GZIP. One major reason for the discussed differences is the fact that the proportion of structural commits acts as the *lowest bound* for the proportion of commits with dataflow interactions. As $90 - 100\%$ of commits part of structural CFIs are also part of dataflow-based CFIs, the percentage of commits part of dataflow-based CFIs never falls below the percentage of commits part of structural interactions. The effects of said lowest bound can be recognized in the first plot of Figure 5.3, where a project's rank among the dataflow proportions of commits reflects its rank among the structural proportions. That is also because the percentages of commits, whose interactions with features are only discovered by our dataflow analysis, are the same or only a fraction of the percentages of those with structural interactions. Similar strong differences in the structural proportions of commits can thus also be found in the dataflow proportions. Including structural CFIs in the evaluation of our dataflow-based CFIs is definitely necessary to properly categorize their proportion and the new information they could add to the

analysis of a project. A substantial share of the examined dataflow interactions stem from our previously investigated set of structural interactions. As we have determined structural CFIs to heavily coincide with dataflow-based CFIs, we already know of this set of dataflow interactions prior to our dataflow analysis.

Nevertheless, we believe that our dataflow analysis is essential in fully assessing commit-feature interactions and their frequency within a project. For xz, the majority of commits interacting with features are exclusivly revealed by our dataflow analysis. For 8, 21 and 28% of active commits in xz, bzip2 and gzip respectively, it revealed additional interactions with features that we could not discover with our structural analysis. This means that the data of, on first sight unrelated, commits and features can be highly connected.

Besides that, we find that many commits interacting with features both structurally and through dataflow, also interact with *other* features *only* through dataflow. This lead us to examine whether commits structurally interacting with features are more likely to affect features through outside dataflow in comparison to any active commit. We find sufficient evidence to support the notion that this is indeed the case, including an almost 5-times higher probability in xz. This suggests that data, changed or allocated by a commit in one feature, is often used within a different feature again. Therefore, a developer changing code of a feature should be aware that data of said code has an increased probability to affect the functionality of other features. This also provides evidence that feature code and remaining code of a program is more separate from each other than the initial percentage of commits with outside dataflow interactions would suggest. When discussing the second part of RQ2, we seek to underline this notion by determining how often outside commits affecting a feature through dataflow, structurally interact with other features.

Lastly, we discuss the extent of feature code in a project and its consequences on the proportion of commits with structural as well as dataflow interactions. In RQ1, we have confirmed that a feature's number of structurally interacting commits is strongly positively correlated with its size for xz and gzip. It follows that a higher extent of feature code implies a higher proportion of commits with structural interactions. As the latter proportion acts as a lower bound for the proportion of commits with dataflow interactions, we assume the extent of feature code to also have a strong influence on said proportion. At the same time, less feature code and a higher number of active commits could elavate the share of commits affecting features through outside dataflow. This is due to a lower share of structurally interacting commits and therefore less inside dataflow as well as a higher number of active commits located outside of feature regions leading to more possible outside dataflow. We decide to compute the extent of feature code in a project as the fraction of all instructions that are part of feature regions. Thus, we determine the extent of feature code to be 17.2% in xz and 57.6% in gzip. In our evaluation we have also seen that xz encompasses five times as many active commits than gzip. We find that our observations in RQ2 roughly match these values considering our previous explanations. Accordingly, xz has a higher share of commits affecting features through outside dataflow but a substantially lower share of active commits part of dataflow interactions. It should be noted however that the extent of feature code does not have an exact relation to the proportion of commits with structural interactions. From xz's 17% extent of feature code, we would expect a signficantly higher number than the 5.8% of commits part of structural CFIs for example.

Understanding Features and the Commits Affecting them Through Dataflow

Especially in the case of XZ and GZIP, we find significant evidence to support the notion that features are generally affected by more outside than inside commits through dataflow. This means that the majority of commits affecting a feature through dataflow are located outside of the feature they interact with. Thus, most of the data belonging to the detected interactions of features originates outside of their regions and subsequently flows into them. The ratio of outside to inside commits is often extremely skewed towards outside commits, while the opposite is rarely the case and if so, to a much smaller degree. One reason for this is that even features with little size are affected by many outside commits, but only few inside commits. That is because the number of a feature's inside commits is strongly positively correlated with its size, while the same is not the case for a feature's outside commits. Specifically, we do not find evidence to suggest that the size of a feature and the number of outside commits affecting it through dataflow are correlated in some way or another. The positive correlation found for XZ is mainly driven by its largest feature, where the correlation coefficient subsides to 0 without it. GZIP shows a statistically signficant negative correlation, BZIP2 no correlation and LRZIP a statisitcally insignifcant positive correlation. From the previous explanations, it follows that the number of inside commits generally increases with rising size of a feature, while the number of outside commits is less predictable. This observation supports the notion that the proportion of outside to inside commits of a feature and its size are negatively correlated. While we determine the according linear correlation coefficients to be negative for all projects, they are rather high and their respective p-values too low to warrant significant evidence. Still, most data points to the existance of a, ableit slight, negative correlation, which could be confirmed by investigating more projects.

During the evaluation of the first part of RQ2, we find that commits structurally interacting with features have an increased probability of influencing other features via outside dataflow. To continue our investigation in this regard, we determine the proportion of outside commits of features that are part of structural CFIs as well. The average proportions are 0.41, 0.59, 0.75 and 0.43 for the features of XZ, GZIP, BZIP2 and LRZIP, respectively. All features except force of GZIP have proportions higher than 0.2 and the majority of values range close to the average of a project. We can therefore confirm that a significant proportion of the data that flows into a feature probably originates from or is modified in other features. This also means that features are less likely to be influenced by commits not changing any feature code than our determined numbers of outside commits would suggest.

The regression plots of GZIP show that their strong linear correlations are mainly caused by two clusters of features. One cluster consists of large features with many inside and comparatively few outside commits, the other of features with a small size and significantly more outside than inside commits. In the evaluation of RQ1, we explain that, due to feature overlap, the sets of commits that structurally interact with each feature from the first cluster are almost identical. This suggests that the inside commits of said cluster are almost identical as well and that this could also apply to the outside commits of the two clusters. Indeed, we determine the majority of outside commits from the features of each cluster to be identical. The respective percentages are $\geq$60% for the cluster encompassing the large features and $\geq$70% for the cluster of the smaller features excluding recursive. Thus, the two clusters are not formed by coincidence, but because features of a similar size are largely affected by the same

set of outside commits. For the first cluster, this phenonemon can be explained by the fact that its features share large portions of their code, which results in them interacting with the same commits by design. At the same time, this is not, or only partially, the case for the features of the second cluster. For them, we note that 25 inside and 17 outside commits from features of the first cluster make up almost all of their identical outside commits. Interestingly, this relation is especially apparent for the smallest feature of GZIP, `no_time`, which only encompasses 23 instructions and is fully located inside of the regions of the `force`-feature. All of the 65 commits affecting `no_time` through outside dataflow are inside or outside commits of `force`. Thus, we find that there also exists a lot of dataflow between features of the two clusters, where the code of the `no_time`-feature acts as a main intersection of said dataflow. This is another product of GZIP's high degree of feature overlap, where the features of the second cluster are often completely nested inside the larger features of the first cluster. The data stemming from commits interacting with these large features, eventually passes through the smaller features within them, making said commits interact with the nested features through outside dataflow. A high degree of feature overlap can therefore not only lead to many features sharing the same structurally interacting commits, but also the commits affecting them through outside and inside dataflow.

5.2.3 *Discussion: RQ3*

We find that our dataflow analysis is able to reveal many additional developers that do not directly participate in the development of a feature, but rather contribute commits whose data is later used inside features. In most features of GZIP, BZIP2 and LRZIP, we detect a similar or higher number of unique dataflow authors compared to the number of structural authors. While most unique dataflow authors are naturally found for features with more dataflow than structural authors, there also exist unique dataflow authors for features with more structural authors. That is because, for the three mentioned projects, only slightly over 50% of structural authors also interact with their respective features through outside dataflow. This means that, while likely, developers of features do not always contribute commits affecting them through outside dataflow. They also are not the ones generally introducing these commits, as the majority of features encompass more unique dataflow authors than structural authors.

The only statistically significant correlations relating the size of a feature to its number of interacting authors were produced by GZIP. Here, we are able to gather initial evidence that the number of authors implementing a feature strongly increases with its size. In contrast, we found the opposite to be true for the number of authors influencing it through outside dataflow. However, as only a single project produced statistically significant results, we consider the question to be completely open to further research. XZ and BZIP2 are unsuitable for the computation of a significant correlation due to their small number of different authors interacting with features and LRZIP due to its small number of features.

GZIP's strong positive correlation in regards to its structural authors matches its correlation between the size of a feature and its number of structurally interacting commits in RQ1. This is also the case for GZIP's strong negative correlation regarding its dataflow authors and the correlation calculated for the number of outside commits in RQ2. Overall, we notice that the number of authors interacting with a feature reflects the length of the set of commits from which we have extracted the authors. In GZIP, the two are positively correlated with a

coefficient of 0.73 and a p-value of 0.004 for the number of structural commits and authors and a coefficient of 0.54 and a p-value of 0.058 for the number of outside dataflow commits and authors. Thus, the number of commits interacting with a feature can hint at the number of authors that interact with it. One reason for these strong positive correlations is the high degree of feature-nesting, which caused features of the two discussed clusters of GZIP to have largely identical sets of interacting commits. As the set of authors for a particular feature is extracted from the commits interacting with it, it is unsurprising that the authors interacting with features of each cluster are also largely identical. This applies to 85 to 100%, depending on the respective feature, of the structural authors of the four largest features. Similarly, the percentage of identical dataflow authors is $67 - 80\%$ for the first cluster and $75 - 100\%$ for the cluster of smaller features excluding recursive. Thus, a high degree of feature overlap can lead to many features having the same set of interacting authors. This makes it difficult to trace whether, a certain set of authors has really implemented a certain set of features or whether, for example, only one author from the set has implemented one feature from the set.

In GZIP, a total of 14 authors interact with features in any way, from which 13 structurally interact with features and 12 affect features through outside dataflow. Only one author that does not touch any feature code, affects features through dataflow and only two authors structurally interacting with features do not affect other features through outside dataflow. Here it is important to note that all of GZIP's 13 developers[3] interact with features either structurally or through dataflow or both. This is not surprising given GZIP's high extent of feature code making up the majority of its instructions. That the extent of feature-code plays a role in what share of a project's authors interact with features can also be seen for BZIP2 and LRZIP with an extent of 8.2% and 1% respectively. While there are 19 and 27 contributors in BZIP2's and LRZIP's Git-repository respectivly, there only exist 4 unique structural authors of features in both projects. The fact that BZIP2 only encompasses 3 unique outside dataflow authors, while LRZIP encompasses 8, likely has to do with LRZIP having many more active commits and overall contributors.

5.3 THREATS TO VALIDITY

In this section, we dicuss the internal and external threats to the validity of this work. There are some potential threats to the internal validity of our gathered data, which stem from our implementation in VaRA. Concerning the external validity of our findings, most dangers stem from the selection of projects we investigate.

5.3.1 *Threats to Internal Validity*

From the definition of feature regions, it follows that we implement feature regions in such a way that any instruction, whose execution depends on a configuration variable, is part of a feature region. However, not every such instruction also implements the functionality of a feature, as can be seen in listing 5.1. This means that the regions of a feature can overapproximate the instructions responsible for its functionality. If the instructions, the CFI of a feature occurs in, are such overapproximated instructions, it follows that said CFI is overapproximated as

3 The remaining author from the total of 14 authors interacting with features stems from GZIP's used submodule GNULIB

well. Since feature regions are used for computing both structural and dataflow-based CFIs, our generated reports probably contain overapproximated CFIs.

The detection and tracking of configuration variables stored in structs is currently being implemented in VaRA and was not usable for this work. Specifically, the configuration variables of some features in xz and LRZIP cannot be detected, meaning that no CFIs can be collected for them. Logically, this affects the overall and project-specific results of our RQs. For example, the percentage of commits with only one feature-related concern are likely lowered by considering less features.

Especially in the case of LRZIP, we work with a small number of features resulting in very few datapoints that can be used for the computation of our results. Accordingly, the p-values of the linear correlations computed for LRZIP always fall into our rejection interval and, thus, produce no statistically significant correlation. Another problem considering LRZIP is that we only examine a small proportion (30%) of its existing features, which means that our results might not be representative for the entire project.

In this work, we have discussed the issues that arise from feature overlap (see Section 3.5) in detail. In particular, feature overlap becomes a threat to the internal validity of our work when the overlap is created by feature nesting. While the code of a nested feature only implements functionality of one feature, it belongs to several feature regions. Thus, said code can produce both structural as well as dataflow-based CFIs with features whose functionality it does not implement. This is problematic, as our analysis cannot decipher which feature is being implemented in the case of code with feature nesting. We therefore are not able to filter the respective interactions, which means that feature nesting is an additional cause for overapproximating CFIs. Feature overlap that is created by structurally interacting features does not threaten our validity in such a way. As the code, where the respective overlap occurs, coordinates the interplay between the involved features, it also implements functionality for all of them. In Section 3.5, we mentioned that we currently cannot differentiate between the two discussed causes of feature overlap, which means that we cannot separare feature overlap into safe and unsafe for our analysis.

```
1. if FeatureEncryption:                          ▷ FeatureEncryption
2.     sendEncryptedMessage(message)              ▷ FeatureEncryption
3. else:                                          ▷ FeatureEncryption
4.     sendMessage(message)                       ▷ FeatureEncryption
```

Listing 5.1: Example for the Overapproximation of Feature Regions

The function of FeatureEncryption is to send the message encrypted. According to our definition of feature regions all instructions stemming from the shown lines of code belong to a region of FeatureEncryption, as their execution depends on the configuration variable of FeatureEncryption. However only instructions stemming from the lines $1 - 2$ implement the actual functionality of the feature. Thus our analysis overapproximates the lines $3 - 4$ to also belong to the feature.

5.3.2 *Threats to External Validity*

Our pool of investigated projects is limited and it is likely that the way commits and features are used in them is different to other projects to some extent. In previous chapters, we mentioned that the chosen projects are rather small and from a compression domain. This could mean that our findings might not be applicable for projects of larger sizes or of different domains. As we already factor in the size of a feature in the analysis of our data, we are able to mitigate some doubts about the applicability of our results onto larger projects.

Apart from the general selection of our exmained projects, BZIP2 in particular is a project whose results must be treated with caution. In the discussion of RQ1, we already explained that the overall project and its features were largely already implemented when the actual development in the Git repository started. This is reflected in a total of only 37 active commits and a maximum of 7 interacting commits for a feature. Accordingly, the results of BZIP2 differ significantly from those of the other projects. It was important for us to describe this fact accordingly so that we can justifiably attribute less weight to the results of BZIP2. It is worth mentioning that the Git development of the other three projects also began with a pre-existing version. However, their features were not or only partially implemented at that point. Their entire code has also changed significantly since then, as can be deducted from the high number of active commits. We do therefore not consider this circumstance as a major threat to the validity of the results for XZ, GZIP and LRZIP.

6

RELATED WORK

Interactions between features [3, 4] and interactions between commits [11] have already been used to answer many research questions surrounding software systems. 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. [11] 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 [11]. 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 [11] 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. [4] was first to track load-time configuration options along program flow to implement the automatic detection of features inside programs. 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 [4]. Instead of using load-time configuration options, applications also use the C Preprocessor to implement variability. Numerous studies investigating configurable software systems have focused on such applications including the Linux kernel [3, 5, 8].

Feature interactions are a broad research topic in software systems encompassing numerous forms [1]. For example, interaction bugs are bugs caused by feature interactions that occur if and only if multiple features are activated [6]. Furthermore, performance interactions are defined as unexpected behavior when combining features that cannot be explained by their individual performance [12]. Using sampling techniques, performance interactions can be automatically detected, thus succesfully "improving perfomance prediction in configurable systems" [1]. Most related to our work, feature interactions can also occur at source-code level when the code implementing different features interacts structurally or through control- or data-flow. Since they are "realitively easy to identify"[1], they could be used to predict other kinds of interactions, although this is not necessarily the case [1]. Specifically, Kolesnikov et al. [3] published a case study on the relation of external, i.e. performance, 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 feature interactions [3].

In the last section of this chapter, we summarize prior research on feature development, as it is a major topic of this work. In contrast to our other research topic of evaluating the dataflow from commits to features, feature development and related topics have been the focus of several studies. Passos et al. [8] analysed feature scattering in the Linux kernel, specifically its device drivers, and found that most features (82%) are introduced without scattering. Overall, the percentage of driver features classified as scattered falls around or below 20% in the examined releases [8], meaning that most of them are implemented rather modularly. Michelon et al. [5] investigated the life cycle of features in four configurable software systems by examining how commits change the code of features over their entire development. They found that features are frequently changed and that these changes often include "substantial modifcations" [5]. Interestingly, Michelon et al. [5] noted that commits are more likely to change multiple features at once than they are to introduce changes only to a specific feature. This allowed them to determine that certain features evolve together, as commits changing code of one feature often change code of its co-evolving feature as well. Furthermore, the majority of commits affecting feature code change code of the base system as well [5], which means that commits are unlikely to only change feature specific code. While the retirement of a feature marks the end of its development, it is arguably still a part of it. Michelon et al. [5] also investigated this topic and found that features are rarely removed after their introduction. This is not the case for the Linux kernel however, as Passos et al. [7] discovered that "feature retirement is rather frequent" in it.

7 CONCLUDING REMARKS

In this thesis, we achieved an initial overview of commit-feature interactions in software projects, specifically structural and dataflow-based CFIs, their interplay, as well as other properties influencing them. We are able to answer or, at least gain some interesting insights, into numerous research topics we set out to investigate. In the following paragraphs, we briefly summarize our most important results and our thoughts regarding future work.

FEATURE DEVELOPMENT AND USAGE OF COMMITS THEREIN In RQ1, we find that the number of commits used during development varies strongly between the features of a project. Here, the size of a feature is an accurate predictor for the number of commits as the two metrics are strongly positively correlated. Regarding the usage of commits during feature development, we support the notion that commits usually deal with a single responsibility in relation to features as the majority ($> 69\%$) of them only have one feature-related concern. At the same time, commits structurally interacting with features might not only affect feature code, but introduce changes to the base system as well. Specifically, Michelon et al. [5] investigated this topic and finds that the majority of commits changing feature code also change code of the base system [5]. While the changed code of the base system might be unrelated to the changed features, meaning that the commit deals with multiple responsibilites both feature-related and feature-unrelated, we argue that a dataflow analysis could prove the opposite. Future studies could determine whether the respective code in the base system and the respective features interact through dataflow and, thus, show that the commit still deals with a single responsibility.

FEATURES AND THE COMMITS AFFECTING THEM THROUGH DATAFLOW Similarly to the number of implementing commits, features of a project also encompass a wide range in the number of commits affecting them through outside dataflow. Admittedly, our analysis potentially underestimates the number of commits whose dataflow stems from the outside, although this could be fixed in future research. We say that a commit affects a feature through inside dataflow, meaning that the dataflow occurs inside the regions of the feature, when it also structurally interacts with the feature. As the commit could also be partially located outside the feature, we could be dealing with an outside dataflow origin instead. One way to solve this issue would be to track whether the commit taint (see Section 2.2), belonging to data being used within a feature, originated inside the regions of said feature.

We find conflicting evidence regarding the linear correlation between the size of a feature and the number of commits interacting with it through outside dataflow. All of our four examined projects produce unique results ranging from positive, to no, and even strong negative correlations. Further research is necessary to determine whether there exists an accurate predictor for the number of commits affecting a feature through outside dataflow, where considering properties of a feature besides its size might be required.

PROPORTION AND DEPENDENCIES OF COMMITS AFFECTING FEATURES THROUGH DATAFLOW
The fraction of commits part of dataflow interactions are not unanimous among the projects and more common than we expected. They range from every ninth commit in xz to more than every third commit in gzip. Furthermore, we found that their fraction in a project is strongly influenced by the percentage of commits that structurally interact with features. Specifically, we propose that said percentage acts as a lower bound for them since we could confirm the notion that structural CFIs heavily coincide with dataflow-based CFIs discussed in Section 3.3. Still, our dataflow analysis revealed previously hidden interactions for the majority of commits interacting with features in all projects ($> 70\%$).

Although this was not a topic we initially intended to examine, we determined that commits used to implement features have an increased likelihood to affect *other* features via outside dataflow. This implies that different features tend to be connected via dataflow, which could serve as motivation to investigate dataflow interactions between features. Discovering which types of features often share data and, thus, depend on each other could potentially have implications on other, for example bug-related [6], interactions between them. In relation to this, future studies might want to determine commits located outside of features whose code serves as an intersection for the dataflow between certain features. The respective commits can additionally be considered when trying to find the cause of bugs or other issues occuring in relation to feature interactions [1]. To determine these commits, our analysis in VaRA needs to be extended allowing us to examine dataflow from features to commits as well. Commits that are affected by and affect certain features through outside dataflow might be commits that are central to the interplay of the involved features.

INITIAL INSIGHTS INTO AUTHOR-FEATURE INTERACTIONS AND FUTURE PROJECT CHOICE
One major problem we faced when evaluating RQ3 was that xz, which played an important role in answering the two previous research questions, was essentially of no use here. That is, because xz was mostly developed by a single author, which means that its features only interact with a single author as well. We therefore consider our results for RQ3 to be preliminary and advocate for the examination of more projects in this regard. Still, the remaining projects have given us first insights on how authors interact with features. We generally find that a feature has more authors affecting it through outside dataflow than structurally interacting authors, i.e. those directly involved in its implementation. Out of the authors implementing a feature, only around half of them also introduce changes influencing them via outside dataflow. Considering this, it follows that a substantial number of authors interacting with a feature exclusively do so through dataflow. Thus, our dataflow analysis is able to discover many additional interacting authors that we cannot reveal with our structural analysis.

Across all RQs, we have seen that projects encompassing a higher number of features produce statistically more significant correlations than projects encompassing less features. Particularly, xz and gzip encompass at least 10 features and generally produce p-values falling out of our pre-defined rejection interval, whereas this is not the case for bzip2 and lrzip encompassing 8 and 3 features respectively. We therefore propose that future research should preferably investigate projects with a minimum of 10 features if they intent to compute pearson correlation coefficients or other metrics relying on the number of provided datapoints.

THE CONSEQUENCES OF FEATURE OVERLAP GZIP has shown initial evidence for the impact and consequences a high degree of feature overlap can have on our generated data. As sets of features share the majority of instructions constituting their size, it follows that they largely interact with the same commits structurally and through dataflow. Perhaps more surprsingly, we even find that features not sharing much code are often affected by the same commits through outside dataflow. Naturally, this phenonemon also manifests itself in clusters of features interacting with the same authors. We are aware that a high degree of feature overlap in GZIP casts some doubt on the implied meaning, for example structurally interacting commits being implementing commits, of the collected CFIs. For the respective features, we cannot be sure what fraction of their interactions occur in instructions actually implementing their specific functionality. While solving this issue goes beyond the scope of our work, it is important for us to adequatly adress and inform the reader about it. We adivce future studies, dealing with interactions of features at source-code level, to take the consequences of feature overlap into account and find ways to mitigate them. Here, it would be important to distinguish between feature overlap caused by structurally interacting features, which does not pose threats to internal validity, and feature nesting, which does (see Section 5.3.1). Additionally, being able to decipher feature nesting, i.e. determine which features are being implemented in code containing nested features, would solve this issue entirely.

BIBLIOGRAPHY

[1] Sven Apel, Joanne M Atlee, Luciano Baresi, and Pamela Zave. "Feature interactions: the next generation (dagstuhl seminar 14281)." In: *Dagstuhl Reports*. Vol. 4. 7. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2014.

[2] 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.

[3] 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).

[4] 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.

[5] Gabriela K. Michelon, Wesley K. G. Assunção, David Obermann, Lukas Linsbauer, Paul Grünbacher, and Alexander Egyed. "The Life Cycle of Features in Highly-Configurable Software Systems Evolving in Space and Time." In: *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2021. Chicago, IL, USA: Association for Computing Machinery, 2021, pp. 2–15. ISBN: 9781450391122. DOI: 10.1145/3486609.3487195. URL: https://doi.org/10.1145/3486609.3487195.

[6] Changhai Nie and Hareton Leung. "A survey of combinatorial testing." In: *ACM Computing Surveys (CSUR)* 43.2 (2011), pp. 1–29.

[7] Leonardo Passos, Jianmei Guo, Leopoldo Teixeira, Krzysztof Czarnecki, Andrzej Wąsowski, and Paulo Borba. "Coevolution of variability models and related artifacts: a case study from the Linux kernel." In: *Proceedings of the 17th International Software Product Line Conference*. 2013, pp. 91–100.

[8] Leonardo Passos, Rodrigo Queiroz, Mukelabai Mukelabai, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Jesus Alejandro Padilla. "A Study of Feature Scattering in the Linux Kernel." In: *IEEE Transactions on Software Engineering* 47.1 (2021), pp. 146–164. DOI: 10.1109/TSE.2018.2884911.

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

[10] 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.

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

[12] Norbert Siegmund, Sergiy S Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. "Predicting performance via automated feature-interaction detection." In: *2012 34th International Conference on Software Engineering (ICSE)*. IEEE. 2012, pp. 167–177.