Bachelor's Proposal

COMMIT FEATURE INTERACTIONS

SIMON STEUER
(2579492)

June 20, 2023

Advisor:
Sebastian Böhm   Chair of Software Engineering

Examiners:
Prof. Dr. Sven Apel   Chair of Software Engineering

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

PRESENTATION ABSTRACT

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

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

INTRODUCTION

Goal of this Thesis

The primary focus of this thesis is to gain an overview of how commits interact with fea- tures 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, strutctural and dataflow-based commit-feature Interactions. While using both types separately can already answer many research questions, we will also show applications utilizing a combination of both. We aim to reveal insights about the development process of features and usage of commits therein with the help of structural commit-feature interactions and high-level repository information. We will also investigate to what extend there exist commit-feature interactions throug dataflow that cannot be discovered with a purely syntactical analysis.

Overview

RELATED WORK

Interactions between Features and Interactions between Commits have already been used to answer many research questions surrounding software projects. However investigating Feature Interactions has been around for a long time whereas examining Commit Interactions is a more recent phenomenon.
In an article published in 2023, Sattler et al. [4] 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. 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 and thus, logically, accounting for most author interactions. 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. [2] first implemented functionality to automatically track load-time configuration options along program flow. Said configuration options can be viewed analogously to feature variables in our research. Their analysis tool Lotrack can detect which features, here configuration options, must be activated in order for certain code segments 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.
Referencing this paper Kolesnikov et al. [1] published a case study on the relation of external and internal feature interactions. Internal feature interactions are control-flow feature

interactions that can be detected through static program analysis as mentioned above. They concluded that considering internal feature interactions could potentially help predict external, performance feature interactions.

BACKGROUND

Interaction Analysis

Essential to the research conducted in this paper is the interaction analysis introduced by Sattler et al. [4]. Their interaction analysis tool SEAL is implemented on top of LLVM and PhASAR. Their approach 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 to the compiler's intermediate representation (IR). This information is added to the LLVM-IR instructions during the construction of the IR. SEAL focuses on commit-information, which encompasses the commit's hash and its respective repository. We say that the commit of an instruction is the commit that last changed the source-code line the instruction stems from. Determining the commit that last changed a source code line is achieved by accessing repository meta-data. The second analysis step involves the actual computation of the interactions. For this, Sattler et al. implemented a special, inter-procedural taint analysis. It's able to track data flows between the instructions of a given target program. Such dataflow interactions occur when an instruction uses data as its input that was changed or allocated by an instruction earlier in the program. Commit interactions can be determined through pairs of instructions that interact with eachother through dataflow. This is accomplished by lifting instructions according to their respective commits made possible by an instruction's mapped commit information.

Definitions

Definition 1. Commits are used within a version control system to introduce the latest source code changes to its respective repository. Inside a repository revision, a commit encompasses all source code lines that were last changed by it. The sum of all source code lines belonging to a commit within a revision is called a **commit region**.

Definition 2. Commit taints are used to track data along program flow essentially carrying information on which commit regions have previously affected this data. An instruction is tainted by a commit if it uses data, that has been changed or allocated inside a commit region, as input.

Definition 3. In general, **features** are parts of a program implementing specific functionality. In this work we focus on features that are modelled with the help of configuration variables. Configuration variables, also called feature variables, decide whether source code, performing a feature's intended functionality, gets executed. The sum of all source code lines whose execution depends on a feature variable is called a **feature region**.

We refine the computation of code regions, that was introduced by Settler et al., by defining the computation of commit and feature regions. This allows allows a precise definition of commit-feature interactions through interactions between commit and feature regions.

Definition 4. With computeCommitRegions(revs), we compute all commit regions for the specified revisions in revs $\subseteq$ R, by computing the revision-specific commit regions for each program revision p^{rev} with rev $\in$ revs.

$$\text{computeCommitRegions(revs)} = \bigcup_{rev \in revs} \bigcup_{p \in rev} \bigcup_{f \in p} \text{computeCR}_{Tag}(f, t{=}Commit)$$

Definition 5. With computeFeatureRegions(revs), we compute all feature regions for the specified revisions in revs $\subseteq$ R, by computing the revision-specific feature regions for each program revision p^{rev} with rev $\in$ revs.

$$\text{computeFeatureRegions(revs)} = \bigcup_{rev \in revs} \bigcup_{p \in rev} \bigcup_{f \in p} \text{computeCR}_{Tag}(f, t{=}Feature)$$

EXAMPLE CHAPTER

We adapt the definitions from Sattler et al. defining interactions between code regions to suit our use case of commit-feature interactions.

Definitions

Definition 6. The function interactingCommitRegions($\blacklozenge$,r_b,revs) computes the set of all commit regions that interact with $r_b \in CR$ with regard to a concrete interaction relation $\blacklozenge \in \lozenge$.

$$\text{interactingCommitRegions}(\blacklozenge, r_b, revs) = \\ \{\, r \mid r \in \text{computeCommitRegions(revs)} \wedge r_b \neq r \wedge r \blacklozenge r_b \,\}$$

Definition 7. The function CommitFeatureRegionInteractions computes the set of all CFRIs that are present in the software program at one of the specified revisions revs $\in$ R, regarding the concrete interaction relation $\blacklozenge \in \lozenge$.

$$\text{CommitFeatureRegionInteractions}(\blacklozenge, revs) = \\ \{\, (\blacklozenge, r_b, \text{interactingCommitRegions}(\blacklozenge, r_b, revs)) \mid \\ r_b \in \text{computeFeatureRegions(revs)} \\ \wedge |\text{interactingCommitRegions}(\blacklozenge, r_b, revs)| > 0 \,\}$$

In this paper we focus on two interaction relations, namely structural relations $\circledcirc$ and dataflow relations $\rightsquigarrow$ as defined by Sattler et al.
It should be noted that dataflow relations, unlike structural relations, are not symmetric. Since we want to focus on commit regions affecting feature regions through dataflow, we designed the definitions in an according way. Namely, we only consider interactions between a commit region r_c and a feature region r_f, s.t. $r_c \rightsquigarrow r_f$. This choice doesn't affect the definition of structural interactions, where we want to consider both cases, since $r_c \circledcirc r_f \Leftrightarrow r_f \circledcirc r_c$.

```
int calc(int val) {                        ⊳ d93dj4gr
    int ret = val + 5;                     ⊳ 7shd28dj
    if (FeatureDouble) {                   ⊳ fu3w17ds    ⊳ FeatureDouble
        ret = ret * 2;                     ⊳ fu3w17ds    ⊳ FeatureDouble
    }                                      ⊳ fu3w17ds    ⊳ FeatureDouble
    return ret;                            ⊳ d93dj4gr
}                                          ⊳ d93dj4gr
```

Listing 1: Commit Feature Interactions

The code example contains both structural as well as dataflow-based commit feature interactions. It's obvious that commit fu3w17ds implements the functionality of FeatureDouble for this function. It follows that a structural CFI can be found between them, as their respective commit and feature region overlap. Commit 7shd28dj introduces a new variable that is later used inside the feature region of the "Double"-Feature. This accounts for a CFI through dataflow, as data that was produced within a commit region is used as input inside an instruction of a feature region later on in the program.

Definition 8. The occurance of structural interactions between a commit and a feature region is defined as a **structural commit-feature interaction**.

Definition 9. The occurance of dataflow interactions from a commit region to a feature region is defined as a **dataflow-based commit-feature interaction**.

Implementation

The detection of structural as well as dataflow-based commit-feature interactions is implemented in VaRA [3]. Similarly to SEAL, VaRA maps information onto the compiler's IR during its construction. This makes it possible to extract two types of information on llvm-IR instruction level.
The first type are code-regions, more specifically commit and feature regions as defined in definition 1. and 3. Thus, structural commit-feature interactions can be collected by iterating over the compiled instructions of a program. According to definition 8., we can store a structural interaction between a commit and a feature, if an instruction is part of a respective commit and feature region. For each such interaction we also save the amount of instructions it occurs in. This is accomplished by incrementing its instruction counter if we happen to encounter a duplicate.
The second kind are taints, more specifically commit taints as defined in definition 2. Similarly to commit and feature regions, VaRA allows us to extract the information of its commit taints for each instruction. Thus, dataflow-based commit-feature interactions can also be iteratively collected on instruction level. According to definition 9., we can store a dataflow-based interaction between a commit and a feature, if an instruction has a respective commit taint while belonging to a respective feature region. Consequently said instruction uses data, that was changed by a commit region earlier in the program, as its input, while stemming from code belonging a feature region.
For our research we examine numerous software projects to get a wide range of reference

data, as commit-feature interactions could potentially vary greatly between different code spaces. Accordingly, the VaRA-Tool-Suite was extended making it possible to generate a report comprising all found CFIs of an according type in a software project. This aids us in examining several software projects to gain sufficient and sensible data about commit-feature interactions. The created reports are also evaluated in the VaRA-Tool-Suite, which offers support to process and display statstics of the generated data.

Conceptualization

Here, we introduce concepts and meanings of structural and dataflow-based commit-feature interactions as well as a combination therof.
A structural interaction between a commit and a feature implies that the instruction said interaction occurs in, stems from code that was last changed by said commit and implements functionality of said feature. It follows that the commit of the interaction contributed code to, i.e. was used to implement, the feature. At the same time we know that each LOC inside a git project was added and last changed by a commit. Thus, every instruction belonging to a feature region is also part of a commit region. By definition, feature regions encompass the entire code space implementing the functionality of their respective features. This means that the commits structurally interacting with a feature, are the commits constituting the code space, i.e. implementing the functionality, of said feature. Following this, we can also say that a commit implements the features it structurally interacts with. By accessing high-repository information, we can determine the author of each commit. Extracting the authors of all commits a feature has structural interactions with, determines the authors that implemented it.

We also want to introduce the concept of **feature scope** as the amount of instructions stemming from a feature's region. The amount of instructions can simply be added up from the amount of instructions saved inside each structural interaction of a feature. The scope of a feature is meant to be a useful measure to assess the extend to which a feature interacts with commits. Features with very little scope, but many dataflow interactions with commits, might be more interesting than features with large scope and the same amount of interactions.
For a feature, it's also possible to estimate the extend to which each of its developers contributed to its implementation. This can be accomplished by tracing back each structural interaction of the feature with a commit, to the commit's author. Now, adding up the amount of instructions for each author and putting it into contrast with the scope of a feature can show how much a developer contributed to the implementation of a feature.

Instructions with structural commit-feature interactions do likely introduce or change data that will be used within the feature of that interaction. Thus, structural interactions heavily coincide with dataflow-based interactions. Unlike dataflow interactions that don't coincide with structural interactions, programmers are much more aware of them. That's why we want to be able to separate these two kinds of dataflow interactions. This can be accomplished by simply checking whether a commit that influences a feature through dataflow, also structurally interacts with it.

METHODOLOGY

Research Questions

RQ1: How do commits interact with features structurally?
We intend to research two main properties which already provide a lot of insight into the development process of features and usage of commits therein. Firstly, we examine the amount of commits, features interact with structurally. This gives us a direct estimate on how many commits were used in the development of a feature. Our analysis also allows us to meausre the scope of feature , which can put the amount of commits used to implement a feature into perspective. Secondly, we want to examine how many features a commit interacts with structurally, e.g. how many features a commit usually changes. This is especially interesting when considering best practices surrounding the usage of commits. It is preferred to keep commits granular meaning they should only fix a single bug or, in our case, change a single feature. Acquiring data on this issue might show how strictly this policy is enforced in the development of features.

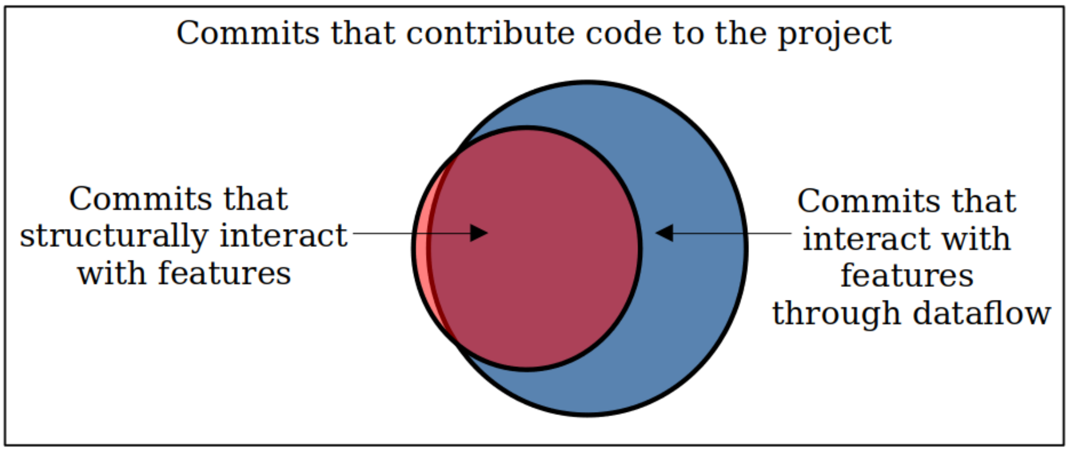
Commits that contribute code to the project
Commits that structurally interact with features
Commits that interact with features through dataflow

Figure 1: Kinds of Commits in a Software Project investigated in this work

RQ2: How do commits interact with features through dataflow?
Investigating dataflow can unveil interactions between parts of a program that were previously hidden from programmers. This can help a programmer understand the extend to which new changes affect other parts of a program. Deploying the introduced analysis in an ad-hoc, functional manner could even aid a programmer when fixing bugs. Bugs occuring in certain parts of a program could be traced back to their cause by factoring in recent changes affecting said parts through dataflow.
Previous research has laid the groundwork for researching dataflow interactions between different parts of a program. However, it has focused solely on dataflow interactions between commits. That's why we want to provide first insights on the properties of dataflow-based commit-feature interactions. Specifically, we investigate how connected commits and features are by analyzing the amount of features a commit usually affects through dataflow. Knowing what fraction of all commits contributing code to a project are part of dataflow-based interactions can show how often new commits affect the data of a feature. Regarding this, it is worth

considering that commits constituting code of a feature are very likely to influence said feature through dataflow. Since dataflow interactions coinciding with structural interactions are so obvious, programmers are also much more aware of them. Depending on the prevalence of feature-regions in a project's code space, this could heavily skew the data in one direction, as a large portion of all dataflow interactions would stem from these obvious interactions. Therefore we want to especially focus on commits that aren't part of a feature giving us more valuable and insightful information.

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

RQ3: How do authors implement features?
Usually there are many programmers working on the same software project, implementing different features, sometimes alone, sometimes with the help of colleagues. We want to shine some light on the exact statistics of this by combining structural commit-feature interactions with high-level repository information. One major question we want to answer is how many authors implement a feature on average, where considering feature-scope could help put this data into perspective. Additionally we aim to investigate the extend to which each developer contributes to the implementation of a feature. This way we can detect development patterns like the existence of a main developer as discussed by Sattler et al. [4]. The collected results could serve as advice for software companies on how to allocate programmers on to-be implemented features.

Operationalization

RQ1: What are the characteristics of structural commit-feature interactions?
The needed data will be collected by creating reports comprimising all structural commit-feature interactions of a chosen software project. The collected data is then evaluated by iterating over all found commit-feature interactions. For each encountered feature we save the commits it interacts with. As the analysis has also calculated the amount of instructions a structural interaction occurs in we save the entire amount of such instructions for every feature. We have previously noted that a feature only structurally interacts with a commit that has contributed code to said feature. Besides that each line of code in a git project was added or changed by a commit, which means that each line of code that belongs to a feature also belongs to a commit. Therefore the commits saved inside the interactions of a feature are the commits that constitue the code space of a feature. With this data, it's possible to calculate the average amount of commits used to create a feature, while being able to correlate this with its size, in our case, its number of instructions. For the evaluation of the second point, we iterate over all found structural commit-feature interactions again, but instead save the features each encountered commit interacts with. From aforementioned explanations, it follows that a commit contributes code, e.g. implements, the feature it structurally interacts with.

RQ2: How do commits interact with features through dataflow?

The projects investigated for dataflow-based commit-feature interactions will be the same projects investigated for structural commit-feature interactions. This choice will allow more insight into a single project and allow us to combine both analysis results as will be discussed below. In **RQ2** we consider all commits that currently contribute code to the project, which we can extract from high level repository information of the project. For each commit we will save whether and if true which features they interact with through dataflow. Similarly to **RQ1**, this is carried out by iterating over the dataflow-based commit-feature interactions in the created reports. The acquired information makes it possible to calculate what fraction of commits interact with features through dataflow. For commits that do have dataflow-based interactions with features, we will examine how many features they interact with on average. The dicussed separation for said commits into those that are part of a feature and those that aren't is accomplished with the usage of already created structural reports. We can find out whether a commit is part of a feature by checking if it is part of a structural commit-feature interaction in the according report.

RQ3: How do authors implement features?
Here, we will examine the same projects as the previous **RQs**. That way we can reuse data produced in **RQ1** to map each feature to the authors that implemented it. In **RQ1** we have already mapped each feature to the commits it interacts with, e.g. that contribute code to it. It's possible to retract the authors of these commits by searching through high-level repository information with their hashes. This will directly give us the authors that implemented a feature. The amount of instructions that stem from code belonging to a feature has also been calculated for **RQ1**. With this information we can correlate the size of a feature with the amount of developers that implemented it. Furthermore we want to estimate the amount of code a developer contributes to a feature. To accomplish this, we adapt the analysis used to map each feature to the authors that implement them. When iterating over the commit-feature interactions we not only save the commit, but addtionally save the amount of instructions of that interaction. When extracting the authors from the commits that were mapped to a feature, we also add up the amount of instructions for each commit. Now we can estimate the amount of code an author contributes to a feature with the amount of instructions stemming from said code.

Expectations

How and to which extend features are used in the projects to be examined is not known and could potentially vary from project to project. Thus, some results of the dicussed research topics are difficult to predict. For example, nesting of feature regions inside each other could lead to an increase in the amount of features a commit usually changes. Due to the discussed best practices of commits, we expect commits to change at most one feature on average if there happens to be little nesting. Because of the unknown size of features, it's not sensible to give an estimate about the amount of commits needed to implement a feature. We expect a rather strong positive correlation between the scope of a feature and its commits however. It was mentioned that we examine small projects meaning that the pool of developers is limited in size. Normally, features only encompass a tiny share of a project's overall code. Besides that they implement specific functionality that some programmer's might have a better understanding of than others. This leads us to the expectation that a feature is implemented by a small share of all developers contributing to a project. Because of the small pool of developers and prior research findings, we expect the existence of a main developer that contributes most if not all of the functionality of a feature. We know that commits structurally interacting with features most likely are part of dataflow interactions with them as well. Excluding such commits, the extend to which commits interact with features through dataflow depends heavily on what fraction of the code space is made up of feature regions. The purpose of features is to implement additional and sometimes necessary functionality separate from the main program. For this they access and change specific data according to their intended functionality. Provided that feature regions only make up a small portion of the program, we do expect relatively few, albeit important dataflow interactions between commits and features.

Threats to Validity

internal validity: - feature regions overapproximate: instructions belonging to a feature region might not implement its functionality Ex.: if (!FeatureDouble) `x = x + 2;` else `x = x * 2;` entire code is part of feature region of FeatureA => more structural and dataflow interactions than actually present - commit changing a source code line doesn't necessary change its meaning, for example refactoring => some strucutral and dataflow interactions contain commits that did not change the state of a program - taints underapproximate, analysis does not detect all dataflow interactions necessarily (SEAL Paper: page 9) => some dataflow interactions are missed potentially

external validity: - investigated projects are small, => findings might not be applicable to larger projects or might not be scaled accordingly

CONCLUSION

In this work we want to research the main properties of structural and dataflow-based commit-feature interactions. Using high-repository information and a combination of both types allows us to gain additional knowledge on their properties. Following this, we aim to interpret the findings in a sensible way. Structural interactions and the injection of author information can be utilized to provide insights into feature development and usage of commits therin. Dataflow interactions can unveil interactions between features and commits that cannot be discovered through a purely structural analysis. Seeing how common they really are could encourage programmers to be more aware of them. Furthermore, they can improve our understanding on which impact new commits have on features.

Research involving interactions inbetween features and commits has shown that investigating interactions between program entities is a topic worthy of study. For example, Sattler et al used dataflow commit interactions to allow for a more detailed understanding of author interactions in software projects. These interactions also make it possible to identify seemingly insignificant changes that have a central impact on the program. Kolesnikov et al provides further indication for the wide range of subjects interactions can be used for. Particularly, they argued that control-flow interactions between features can help predict performance interactions between them.

How are you going to do it? Be sure that what you propose is doable.

- extend VaRA implementing the detection of structural and dataflow-based commit-feature interaction
- VaRA offers three main properties for this: the extraction of commit and feature regions, as well as commits taint on instruction level
- extend VaRA-Tool-Suite with the creation of separate reports containg both types of interactions
- reports can be combined and enriched with high-repository information, such as info to which author a commit belongs

BIBLIOGRAPHY

[1] 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).

[2] 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.

[3] 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.

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