Bachelor's Proposal

COMMIT FEATURE INTERACTIONS

SIMON STEUER
(2579492)

June 7, 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

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 in 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. 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 instructions uses variables that were changed or allocated by an earlier instruction in the program. Commit interactions can be determined through pairs of these 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.

Commits and Commit-Regions

Features and Feature-Regions

Feature- and Commit-Size

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.

Example Chapter

The detection of structural as well as dataflow-based commit-feature interactions is implemented in VaRA [3]. VaRA offers two main functionalities for this. The first one is the detection of feature- and commit-code, which is accomplished by being able to receive all commits and features an llvm-IR instruction belongs to. Thus we can collect all structural commit-feature interactions by iterating over all instructions in the code space. Inside an instruction we save every combination of commits and features as a CFI. It follows that, in order for an instruction to have a single interaction, it needs to be part of at least one commit as well as one feature. For each structural CFI we also save the amount of instructions said cfi occurs in. This is accomplished by incrementing the instruction counter if we happen to encounter a duplicate CFI. VaRA is also able to track taints of values along program flow, where taints essentially carry information on which commits have previously affected that specific value. Similarly to structural interactions, dataflow-based commit-feature interactions are iteratively collected on instruction level. In this case we speak of an dataflow interaction when an instruction both has a commit taint and belongs to a feature. Consequently this instruction uses a value that was changed by a commit earlier in the program while stemming from code constituting a feature. For our research we will 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 will aid us in examining several software projects to gain sufficient and sensible data about commit-feature interactions. The created reports will also be evaluated in the VaRA-Tool-Suite, which offers support to process and display statstics of the generated data.

METHODOLOGY

Research Questions

RQ1: What are the characteristics of structural commit-feature interactions?
We intend to research two main properties which will already provide a lot of insight into the development process of features and usage of commits therein. Firstly, we will examine the statistical distribution of how many commits, features interact with structurally. This will directly give an estimation on how many commits were used to create a feature. Our analysis also allows us to gauge feature-size, 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 will show how strictly this policy is enforced in the development of features.

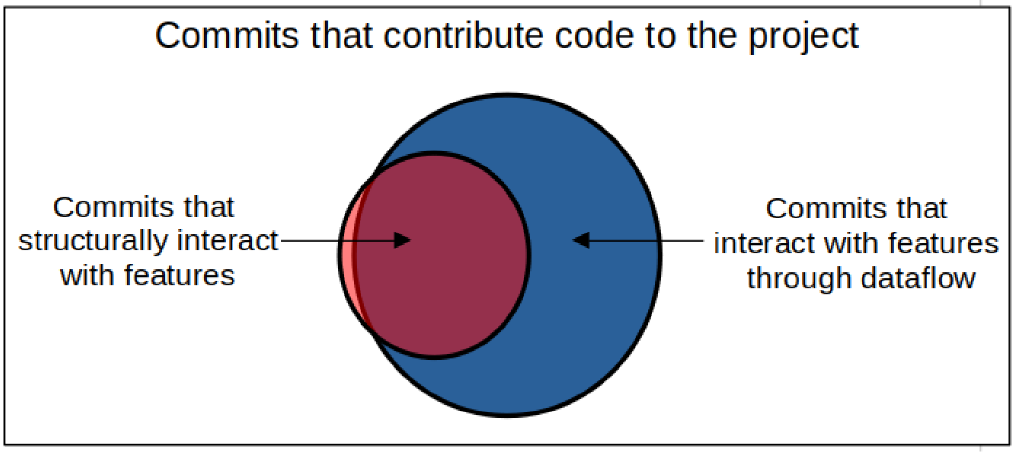
Commits that contribute code to the project
Commits that structurally interact with features
Commits that interact with features through dataflow

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

RQ2: How do commits interact with features through dataflow?
Previous research has focused solely on dataflow interactions between commits. We have already discussed the importance of research in the area and why further research is necessary. That's why we want to provide first insights on the properties of dataflow-based commit-feature interactions. Specifically, we will 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 will 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 the data of said feature. Since these datflow 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. Therefore we want to especially focus on commits that aren't part of a feature, which will give us more valuable and insightful information.

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-size could help put this data into perspective. Additionally we aim to gauge the amount of code each developer of a feature contributes. This way we can recognize 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 in the code space 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 unknown feature-size, it's not sensible to give an estimate about the amount of commits needed to implement a feature. We expect a positive correlation between size of a feature and its commits however. It was mentioned that we will examine small projects meaning that the pool of developers is limited in size. Features usually 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. As previously alluded to we expect the existence of a main developer that contributes most if not all of the code for a feature. 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. However we do know that commits structurally interacting with features most likely are part of dataflow interactions with them as well. Features implement additional and sometimes necessary functionality separate from the program. They do use and change specific data respective 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 - measuring feature-size with number of instructions

external validity: - investigated projects are small, findings might not be applicable to bigger projects

CONCLUSION

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.