Bachelor's Proposal

COMMIT FEATURE INTERACTIONS

SIMON STEUER
(2579492)

May 31, 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 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, 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 on the development process of features 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 insdie a program that cannot be discovered with a purely syntactical analysis.

Overview

BACKGROUND

Commit-Feature Interactions

Broadly speaking commits interact with features, when one affects the other in any way. Obviously there are many possibilities for such interactions in a Software Project. Since looking at all possible types would go beyond the scope of this work, we will instead concentrate on two of the most important ones. To properly define both types we will first introduce some necessary definitions.
Sattler et al. [4] aptly explains that a program P can be viewed as a composition of instructions $i_1, ..., i_n \in$ P. Every such instruction stems from code that was contributed by a sequence of commits $c_1, ..., c_m \in$ C, where C is the set of all commits. Therefore, we say that each instruction belongs to a commit, namely the commit that last changed the code this instruction stems from.

Definition 1. Let $Commit(i)$ be a function that maps an IR instruction i $\in$ P to the commit it belongs to.

Secondly, we say that an instruction is part of a feature when its execution depends on said feature being activated or directly uses said feature variable. Naturally, there can exist more than one such feature.

Definition 2. Let $Features(i)$ be a function that maps an IR instruction i $\in$ P to all the features it is part of.

An instruction is tainted by a commit when it uses values that were changed or introduced by an instruction belonging to said commit. Still, taints of values are not permanent and only last as long as their value is not freshly set.

Definition 3. Let $Taints(i)$ be a function that maps an IR instruction i $\in$ P to all the commits that taint it.

```
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. A structural CFI can be found between the commit with hash ”fu3w17ds” and the ”Double”-Feature, as commit fu3w17ds implements the functionality of FeatureDouble for this function. Commit 7shd28dj introduces a new variable that is later used inside the ”Double”-Feature. This accounts for a CFI through dataflow, as data that was changed by the aforementioned commit is used by a feature later on in the program.

Structural Interactions

We speak of a structural commit-feature interaction when there is an occurence of syntactical overlap between commit- and feature-code.

Definition 4. Let $StructuralInteractions(c,f)$ be a function that returns all instructions i ∈ P containing a structural interaction between commit c and feature f:

$$StructuralInteractions(c, f) = \{\ i \mid c \in Commit(i) \land f \in Features(i)\ \}$$

Consequently we say that a commit c and a feature f interact structurally, if and only if $|StructuralInteractions(c, f)| \geq 1$.

Dataflow-based Interactions

We speak of dataflow-based commit-feature interactions when an instruction, that is part of a feature, uses values that were changed by an instruction belonging to a commit earlier in the program.

Definition 5. Let $DataflowInteractions(c,f)$ be a function that returns all instructions i ∈ P containing a dataflow-based interaction between commit c and feature f:

$$DataflowInteractions(c, f) = \{\ i \mid c \in Taint(i) \land f \in Features(i)\ \}$$

Similarly to structural interactions we say that a commit c and a feature f interact through dataflow, if and only if $|DataflowInteractions(c, f)| \geq 1$.
For programmers it can be difficult to be aware of these dependencies as they might be hidden and span over several files. We provide software that can automatically discover these, which can aid a programmer’s ability to find errors and remove bugs. For instance looking at a features’s dataflow-based interactions with recent commits can help detect the commit that might have caused a bug in said feature.

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.

METHODOLOGY

Research Questions

RQ1: What are the characteristics of structural commit-feature interactions?
We intend to research two main properties that will already give a lot of insight into the development process of features and usage of commits. Firstly, we will examine the statistical distribution of how many commits, features interact with structurally. This will directly give a rough estimation on how many commits were needed to create a feature. Secondly, we want to examine how many features a commit affects structurally, e.g. how many features a commit usually changes. This is especially interesting when considering best practices surrounding the usage of commits.

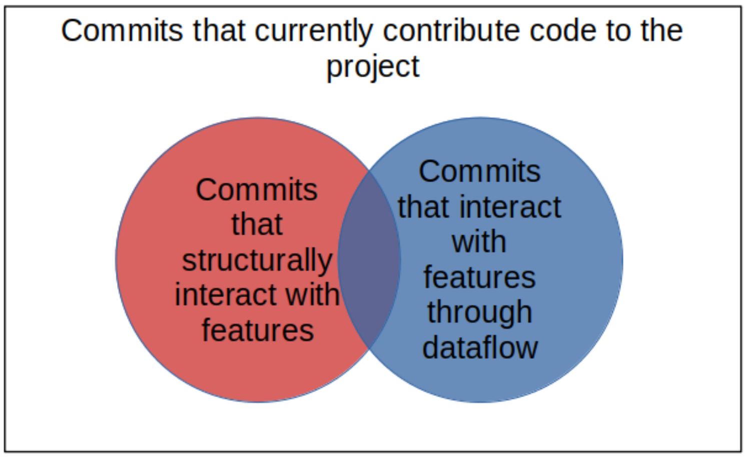
Commits that currently contribute code to the project
Commits that structurally interact with features
Commits that interact with features through dataflow

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

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 software development.

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. One could form several assumptions about them based on a basic understanding of commits and features in software engineering. Showing wether these can be based on statiscal evidence will either secure or change the way we view them. Specifically, we want to investigate how interconnected commits and features are by analyzing the amount of features a commit usually affects through dataflow. Showing what fraction of all commits contributing code to a project affect data of a feature will provide further insight on this. Since it's plausible that commits constituting code of a feature might be more likely to influence the data used in said feature, we want to differentiate between these types of commits here. This can be accomplished with the help of our structural commit-feature interaction analysis. This will give us important information on how likely commits, that aren't part of feature, are to influence the data of a feature.

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 using structural commit-feature interactions and high-level repository information. One major question is how many authors implement a feature on average, where considering feature-size could help put this data into perspective. The collected results could serve as advice for software companies on how to balance workload on to-be implemented features.

Operationalization

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.

RQ1: What are the characteristics of structural commit-feature interactions?
The needed data will be collected by creating the aforementioned 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 and add up the amount of instructions where a structural interaction with said feature occurs. 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 belonging to a feature also belongs to a commit. Therefore all the commits of a feature saved during evaluation 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.

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 analyses as will be discussed below. In this RQ 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 yes 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?
For this RQ 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 measure the correlation between feature-size and the amount of developers needed to implement it.

Expectations

In this section, discuss the results you expect to get from your evaluation.

Threats to Validity

In this section, discuss the threats to internal and external validity you have to be aware of during the evaluation.

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.