Master's Thesis

TRACKING FEATURE-MODEL CHANGES OVER THE HISTORY OF SOFTWARE PROJECTS

PASCAL DUPRÉ

June 10, 2021

Advisor:
Florian Sattler — Chair of Software Engineering
Christian Kaltenecker — Chair of Software Engineering

Examiners:
Prof. Dr. Sven Apel — Chair of Software Engineering
Prof. Dr. Sebastian Hack — Compiler Design Lab

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

Pascal Dupré: *Tracking Feature-Model Changes over the History of Software Projects,* © June 2021

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

Black-box approaches in performance analyses of configurable systems are inferior to white-box approaches in regards to the extractable information about feature interactions. White-box approaches, however, may require a mapping from features to source code that needs to be updated for different source code versions. A control or data flow analysis could utilize the mapping to offer feature specific insights. Updating or creating such a mapping for many feature models, as an evaluation of the approach might require, is tedious and error prone. In this thesis, we explore an approach to approximate feature models for new source code versions on the basis of existing feature models. Our tool *FMstimator* enables users to create a feature model tailored to a source code revision with minimal manual labor.

CONTENTS

1 INTRODUCTION 1
1.1 Goal of this Thesis . . . 2
1.2 Overview . . . 2
2 BACKGROUND 3
2.1 Software Features and Feature Models . . . 3
2.2 Git . . . 4
2.3 Diff Unified Format . . . 5
2.4 Abstract Syntax Trees . . . 6
3 IMPLEMENTATION 9
3.1 Overview . . . 9
3.2 Preparation . . . 11
3.3 Tracking . . . 12
3.3.1 Tracking Heuristic . . . 14
3.4 Merging . . . 18
4 EXPERIMENTS 21
4.1 Synthetic Project . . . 22
4.2 lrzip . . . 22
4.3 vpxenc . . . 23
4.4 grep . . . 23
4.5 Curl . . . 23
5 EVALUATION 25
5.1 Operationalization . . . 25
5.2 Synthetic Repositories . . . 26
5.3 lrzip . . . 27
5.4 vpxenc . . . 27
5.5 grep . . . 28
5.6 Curl . . . 29
5.7 Results . . . 30
5.8 Discussion . . . 31
5.9 Threats to Validity . . . 32
5.9.1 Internal Validity . . . 32
5.9.2 External Validity . . . 33
6 RELATED WORK 35
7 CONCLUDING REMARKS 37
7.1 Conclusion . . . 37
7.2 Future Work . . . 37
A APPENDIX 39

LIST OF FIGURES

Figure 2.1 Feature model represented as feature diagram 4
Figure 2.2 Git history represented as graph 5
Figure 2.3 An AST of the for loop in the example code 7
Figure 3.1 Overview of *FMstimator*'s workflow 10

LIST OF TABLES

Table 5.1 Statistics of experiments on synthetic repository 26
Table 5.2 Statistics of experiments on lrzip 27
Table 5.3 Statistics of experiments on vpxenc 28
Table 5.4 Statistics of experiments on grep 29
Table 5.5 Statistics of experiments on Curl 30
Table A.1 Selected features for synthetic project at commit `e96e20d0` 39
Table A.2 Selected features for synthetic project at commit `17dffc1f` 39
Table A.3 Selected features for *lrzip* at commit `v0.600` 39
Table A.4 Selected features for *lrzip* at commit `v0.630` 40
Table A.5 Selected features for *vpxenc* at commit `v1.2.0` 40
Table A.6 Selected features for *vpxenc* at commit `v1.3.0` 41
Table A.7 Selected features for *grep* at commit `v2.20` 41
Table A.8 Selected features for *grep* at commit `v3.6` 42
Table A.9 Selected features for *Curl* at commit `curl-7_30_0` 42
Table A.10 Selected features for *Curl* at commit `curl-7_40_0` 43

LISTINGS

Listing 2.1 Example File A 6
Listing 2.2 Diff between File A and B in unified format 6
Listing 2.3 Example File B 6
Listing 2.4 Example C++ code 6

1 INTRODUCTION

Software has become ubiquitous in the modern world. With software entering more domains, the demand for more customized software products has risen. However, developing multiple customized solutions from the ground up, tailored specifically to every customer, is not feasible and too expensive. Therefore, it has become the norm to develop configurable software system that can be fitted to the customer's individual needs. Configurable software systems accumulate the common functionality, reducing needed development effort and reducing code duplication. The software system is divided into features, which represent user-visible functionality, which can be enabled or disabled by configuration options. Interactions and dependencies between features are represented and documented in feature models. One of the main advantages of this software design is that bug fixes and other improvements need to be implemented just once instead of multiple times. Afterwards, the software is configured and shipped along with the bugfix to the customers. The product for each customer is no longer customized software, but a customized configuration of the software system, which can be devised with a fraction of the original effort.

Building configurable software system creates a new challenge; there is not a single correct configuration for each customer but multiple. Of course, it is desirable to find the best configuration for each customer considering their requirements, e. g., some prioritize runtime, while other prioritize memory usage. Measuring each potential configuration and picking the best in a brute-force approach is not feasible in most cases, due to the sheer amount of configurations. The number of configurations grows exponentially with the configuration options. Instead, a representative subset of configurations is sampled and measured. The results are used to train a performance model to predict the performance of arbitrary configurations [11] [8] [10] [9]. The performance models are also of interest to developers to identify possible feature interaction with a bad performance. Approaches of this kind view the software system as a black box and do not consider the actual implementation. The performance prediction views the software as a function that receives the configuration as input and returns the performance metric. This ignores the true complexity of the actual software. While it is possible to extract the performance impact of some feature combinations, they might not reflect the actual cause, which limits the usefulness of this information.

Therefore, research is turning towards white-box approaches [17] [16]. Depending on the underlying analysis of these approaches it is necessary to provide an analyses with entry points for each feature. A mapping from features to source code is required. While such a mapping is easy to provide for a single version, updating or creating it for many is tedious and potentially error prone. Furthermore, during development features might be added or removed, which also requires to amend the feature model accordingly. To make white-box approaches that rely on a mapping from features to source code scalable, we must reduce the manual labor required to create a feature model.

1.1 GOAL OF THIS THESIS

In this thesis, we explore an approach to approximate feature models for selected source-code revisions. We analyze the problem of keeping feature-to-source-code mappings up to date, and, based on our insights, we design, implement, and evaluate our tool *FMstimator* as a first prototype. *FMstimator* approximates a feature model for a chosen revision based on two initial feature models, one for an earlier revision and one for a later revision. The given feature models are updated based on code changes between revision towards the chosen revision. A series of heuristics estimates the changes to the feature model and source-code mappings based on the source-code changes. In a final step, the updated feature models are merged into the final approximated feature model for the target revision. The final feature model also contains the updated feature-to-source-code mappings.

We discuss possible source-code changes and their handling in our heuristics. Furthermore, we conduct a case study to assess our approach and gain insights into necessary future improvements. The evaluation on different case studies also provides insight on changes that we should handle with future improvements to our tool.

1.2 OVERVIEW

The thesis is structured as follows. Chapter 2 introduces features and feature models, and introduces the core concepts of *Git*, the version control system, which is used by many open-source projects. Furthermore, we introduce the unified format for diffs and give a brief introduction to abstract syntax trees. Chapter 3 presents the design choices and details of *FMstimator*. The chapter discusses the individual phases separately in Section 3.2, Section 3.3, and Section 3.4. Chapter 4 states our research questions and presents the projects we selected for our case studies. In Chapter 5, we present the results of our experiments and answer the previously stated research questions. We discuss threads to the validity of our experiments in Section 5.9. In Chapter 6 we set our works in relation to other research. We summarize our findings in Chapter 7 and give an outlook on future work.

2 BACKGROUND

In this chapter, we introduce the conceptual abstraction of software features, which represent the user-visible functionality, and feature models, that group features hierarchically and represent dependencies between them, as they are the central objects of this thesis. Furthermore, we discuss the version control system Git, which we use to analyze the development history of software projects. Our tool uses diffs to analyze changes between version, we introduce the diff unified format to represent these changes. Lastly, we give a brief overview of abstract syntax trees, which we use in one of our tracking heuristics.

2.1 SOFTWARE FEATURES AND FEATURE MODELS

Modern software is usually configurable, in the sense that its behavior is not static but the user is able to configure it such that the behavior fits the use case. Enabling or disabling different configuration options add or remove certain functionality fro the software. We refer to these user-visible, configurable functionalities as *features*.

A feature is a distinct user-visible aspect or characteristic [12] in software, or a "characteristic or end-user-visible behavior of a software system" as Apel et al. [1] define it. For example, a database system might allow users to enable or disable data compression. Such a compression feature is, in this case, binary in nature as only two states exists: enabled or disabled. Furthermore, there are numeric features. In contrast to binary features, they are always assigned a numerical value in a specified range and cannot be disabled. For instance, compression algorithms often allow to configure how aggressive they should try to compress data, for example, with a value in a range between 0 and 9, where 0 stands for no compression and 9 stands for the best compression.

In many cases features depend on other features and cannot be configured independently. Imagine a database that offers different storage engines. Each storage engine is represented by a binary feature, however, only one can be used. To represent these dependencies, feature models are used to structure and express them.

In Figure 2.1 we show a feature model, based on our database examples, displayed as a feature diagram. It collects all features of a software and it displays dependencies and relationships between features. The feature diagram represents features in boxes; a solid border denotes binary features and a dashed border denotes numeric features. Furthermore, a child can only be enabled if the parent is enabled; a numeric child is ignored or configured with a default value, if the parent is disabled.

The *Root* feature on top represents the non-configurable code, also referred to as base, and cannot be disabled. It has two children, *Compression* and *Storage*. Compression is marked as optional by the hollow circle on top. *Level* is a numeric feature, $[0-9]$ represents the allowed value range; it is also possible to specify a finite set of possible values. The filled circle on top of *Storage* denotes that it is not optional, i. e., it is mandatory and must be enabled. Mandatory features are not configurable in the strict sense and are usually used

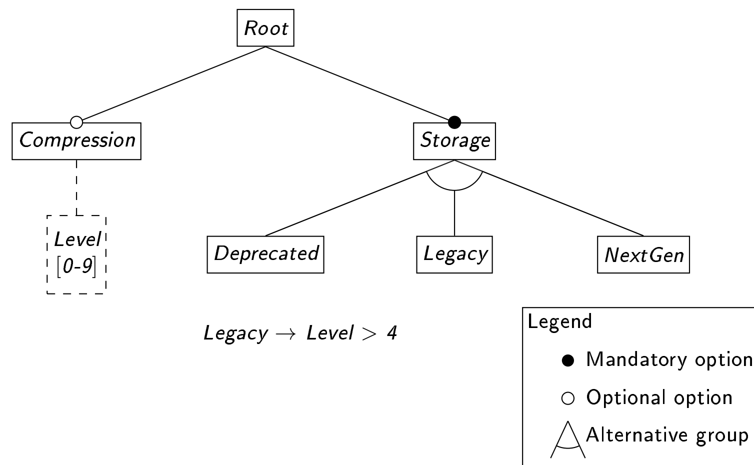

Figure 2.1: Feature model represented as feature diagram

to encode structure or groups of similar features. The arc between the connections to its children indicates that *Storage* is an alternative group, i. e., exactly one of its children must be enabled.

Further relationships, also known as cross-tree-constraints, are noted below the diagram as propositional formulas. In the example, the *Legacy* storage engine requires that *Level* is set to at least 4. Wherever possible, relationships should be encoded in the tree structure and cross-tree-constraints should only be used if necessary.

2.2 GIT

Git[1] is a version control system. Version control systems are used to manage repositories and track changes in its files such as source code. Many open-source projects host their source code via public git repositories, allowing virtually everyone to help development or inspect the project's code and history. During development developers create revisions, also known as commits, to archive and share their changes to files. Commits save the new version of the changed files and some meta information, such as author name and email, a message, and parent commits, i. e., commits on which the new version is based upon. Each commit can be uniquely identified by a hexadecimal hash. In order to simplify the handling of these, Git also allows to create tags to provide a meaningful name for important commits, e. g., version releases.

One of Git's benefits is its branching model. In most projects developers work in parallel on the same code basis. Git allows to easily create branches and merge them again when needed. In many projects development is performed exclusively on branches which are later merged into the primary branch (usually called *master* or *main*) of the project. This development style keeps the primary branch clean, prevents conflicts, and allows to have a working version at all times in this branch.

Figure 2.2 shows a git commit history as directed graph. Each circle represents one commit and the arrows point to parent commits, forming a chain. The commits are chronologically ordered from left to right. The leftmost commit is the first and initial commit and the

1 https://git-scm.com/

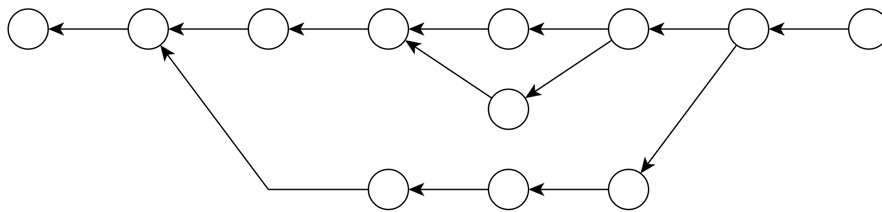

Figure 2.2: Git history represented as graph; circles represent commits and outgoing arrows point to parent commits.

rightmost is the latest commit. After the second commit the history branches off. This happens when multiple future commits are based on the same commit, e. g., when two developers are working in parallel. This branch merges back into the primary commit chain at the second to last commit. The merge commit has two parent commits: the last commit of the branch and the last commit of the original commit chain.

In this thesis, we also refer to a commit as older than another, if it is an ancestor of it. Furthermore, a commit's parents are sorted in the order given during creation. The first parent commit is usually the last commit on the same branch and the remaining parents stem from other branches. However, it is possible to change or circumvent this behavior. The first commit in the list of parents is also called the first parent.

2.3 DIFF UNIFIED FORMAT

In many scenarios it is useful to view the changes between two files. For example, in code reviews it is often sufficient to inspect and discuss changes to existing code. Difference algorithms create an edit script, also known as patch or diff, that contains instructions on the needed changes to transform one file into the other.

One of the most popular difference algorithms was invented by Myers [13]. It has a $\mathcal{O}(ND)$ time and space complexity, where N is the length of both files combined and D is the size of the resulting edit script. There are multiple formats to represent the differences, however, we use the unified format[2] for its better readability.

Consider Listing 2.1 as file A and Listing 2.3 as file B. The second paragraph of file A is deleted and the last paragraph of file B is added. Listing 2.2 shows an edit script to transform file A to file B, or vice versa, if the instructions are inverted accordingly. The unified format starts with a two line header which contains labels for the old (---) and new (+++) file. The header is followed by one or more hunks. A hunk is a collection of differences which are considered to be close to each other. Usually, they also contain context lines before and after changes for better readability. Hunks in unified format start with a single-line header, starting and ending with `@@` (Line 3 and line 13 in Listing 2.2). Hunk headers contain two number pairs, one for each file. The first number in a pair is the starting line and the second number is the number of lines the hunk spans in the respective file. If a hunk contains only a single line, the second number is omitted. The first number pair, marked by `-`, specifies the hunk's location in the old file, while the second pair, marked by

2 https://www.gnu.org/software/diffutils/manual/html_node/Detailed-Unified.html

```
This paragraph will not
change in the next version.

This paragraph will be
deleted in the next version.

This paragraph will not
change in the next version.
However, we can always add
more text below. For
demonstration purposes, this
text needs to be this long.
```

Listing 2.1: Example File A

```
--- file_a.txt
+++ file_b.txt
@@ -1,9 +1,6 @@
 This paragraph will not
 change in the next version.

-This paragraph will be
-deleted in the next version.
-
 This paragraph will not
 change in the next version.
 However, we can always add
@@ -11,3 +8,5 @@
 demonstration purposes, this
 text needs to be this long.

+This is a new paragraph, it
+was not here the last time.
```

Listing 2.2: Diff between File A and B in unified format

```
This paragraph will not
change in the next version.

This paragraph will not
change in the next version.
However, we can always add
more text below. For
demonstration purposes, this
text needs to be this long.

This is a new paragraph, it
was not here the last time.
```

Listing 2.3: Example File B

+. After the header, the added, removed or context lines are listed. The first character in each line indicates the kind of change.

- + indicates an added line.
- - indicates a removed line.
- a whitespace indicates no change.

2.4 ABSTRACT SYNTAX TREES

An *abstract syntax tree* (AST) is a representation of source code in the form of a tree. This data structure is primarily found in compilers and code analysis tools. An AST represents the source code's syntactical structure in a tree. The tree representation allows to encode certain source code details implicitly, such as delimiters, parentheses, etc. Depending on the AST implementation it contains additional information, e. g., a mapping of the tree nodes to source code locations.

Listing 2.4 shows some source code in C++, while Figure 2.3 shows a simplified AST representation of the for loop spanning lines 3-5.

```cpp
int nsum(int n) {
  int acc = 0;
  for (int i = 1; i <= n; ++i) {
    acc = acc + i;
  }
  return acc;
}
```

Listing 2.4: Example C++ code

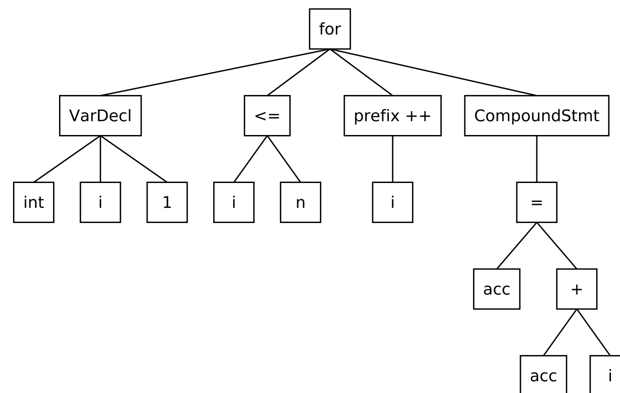

Figure 2.3: An AST of the for loop in the example code

The `for` statement is represented by the node `for`. The node has four child nodes:

1. `VarDecl` represents the declaration of the integer loop variable `i` with the initial value `1`.

2. The binary operator `<=` represents the loop condition `i <= n`. The node has two children which represent the left and right operand.

3. The unary operator `prefix ++` represents the step function `++i`.

4. `CompoundStmt` represents the loop body.

The loop body consists of one line `acc = acc + i;`. This line is parsed in the AST according to the parentheses and operator precedence; `+` has precedence over `=`. Finally, the line is represented with the binary operator `=` on top, with the variable `acc` as left operand and the subtree for `acc + i` on the right.

3 IMPLEMENTATION

This chapter describes our approach of tracking feature model changes over the history of software projects and describes relevant implementation details that are crucial when implementing the approach. First of all, we present an overview over the different phases of our tool *FMstimator*. Furthermore, the tool's functionality is discussed in-depth.

The goal of this tool is to estimate a feature model for a given target commit. Manual creation of multiple feature models is tedious and *FMstimator* aims to reduce the manual work needed. At the core of our approach, we work with user-supplied feature locations. A feature location is a source code location which is essential for a features presence, i. e., if the location is removed, we can assume that the feature no longer exists.

Starting from a given feature model that is valid at a known commit, we incrementally analyze the code changes in the software's revisions towards the target commit and update feature code locations accordingly. If our analysis determines that code essential to a feature is removed, we remove the feature from the estimated feature model.

Of course, this technique only allows to detect feature deletions. To compensate for this limitation, we work with a second feature model in a similar manner. However, in this case we reverse the direction. Starting from a newer commit, we analyze the reverted changes to source code. If we observe a code deletion in this scenario, we, in fact, have observed the first introduction of the code, and in consequence the introduction of the connected feature.

Due to the approximation from two sides, we receive two approximated feature models for the target commit, which need to be unified. To achieve this, we have developed a merging algorithm that performs a union-like operation on feature models.

In the current version, the resulting feature model is intended to give an idea of a correct feature model and not to be a usage-ready version. We can accept this limitation, since the goal of this implementation is to explore the viability of this approach and not to deliver a perfect solution. We chose Python 3.8 for this implementation to allow a fast development of this prototype. It is available on Github[1].

3.1 OVERVIEW

Our tool aims to approximate a feature model at a given target commit based on a number of existing feature models. The core idea is that the tool observes source code locations that are associated with a feature and depending on changes to this source code it decides if a feature was removed, added, or if only the formatting changed.

Without domain specific knowledge it is not possible to detect when features are introduced. In order to work around this, we work with two feature models. The first one belongs to a commit that is an ancestor of the target commit and the second one belongs to a commit that is a descendant of the target commit. The second feature model contains all added features and by observing the commit history in reverse we can detect their introduction

1 https://github.com/padupr/FMstimator

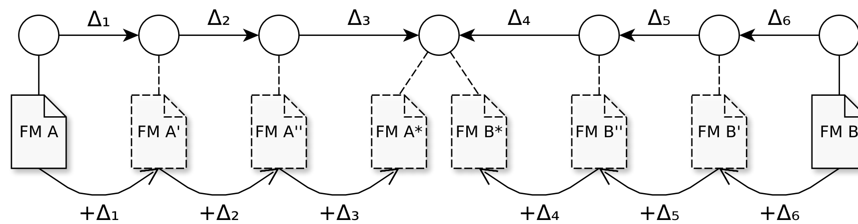

Figure 3.1: Overview of *FMstimator*'s workflow

which looks like a removal in the reverse perspective. Features that are introduced and removed in between cannot be tracked, since they are not known in either model. From here on, we refer to the commits belonging to the first and second feature model as first and second commit, respectively.

Furthermore, where possible, features should be annotated with source code locations, otherwise, they cannot be tracked. This task has to be done manually by the user or, preferably, a domain expert. In most projects it is easy to find source code related to features in command line option parsers or configuration file parsers. Also, these locations are usually the first code that is feature specific. Some features cannot be linked to any specific code location. Exemplary, the *Root* feature, which represents the non-configurable code base.

We implemented our approach in *FMstimator*. Its design splits the approach into three phases. This allows us to replace or update the functionality of each phase individually in the future. The three phases are:

1. Preparation
2. Feature and code location tracking
3. Merging of feature models

In the first phase, *FMstimator* traverses the project's commit history to find a commit sequence between two commits, corresponding to the given feature models that includes the target commit. We use this sequence to generate two commit sequences that start at the corresponding feature models and end on the target commit. Our tool offers two versions to find a commit sequence. The first creates sequences of commits that exist in the project's commit history, i. e., adjacent commits in the returned sequences are also adjacent in the commit history. The second creates two sequences that consist of one of the initial commits and the target commit. The first one is used for an *incremental* analysis of the history and the second is used for a *direct* analysis. An *incremental* analysis can observe code changes in a finer granularity than a *direct* analysis, possibly enabling better results. On the other hand, a *direct* analysis does not observe changes that are reverted again. This can be beneficial, if the removal of a feature's code location is reverted.

Figure 3.1 shows an overview of the workflow in the second phase. Initially, feature models `FM A` (left) and `FM B` (right), and the respective commit sequences towards the target commit are known. Here, the arrows between commits show the tracking direction. Starting from the commit of the first feature model `FM A` we calculate the diff (Δ_1) to the next commit in the commit sequence. We then apply the diff Δ_1 on `FM A` to estimate a feature model `FM`

`A'` for the next commit. To estimate the next feature model we use a series of heuristics to update the locations of each feature. The heuristics analyze the diff for changes that affect the tracked locations. For now, the implemented heuristics focus on updating the location and tolerate formatting changes. If none of the heuristics is able to provide an update to the location, we must assume that the code was removed and delete the location from its feature. If a feature loses its last location, we also remove the feature from the updated model. This process is repeated for every pair of consecutive commits in the commit sequence to incrementally update the feature model until we receive the final model `FM A*`.

The same process is performed on the second feature model `FM B` with corresponding commit sequence to receive the model `FM B*`. This case does not require any special handling, since the commit sequence is already given in the correct order from the first phase. This phase results in two feature models that contain all features that persisted from their original commit to the target commit. The final two feature models are passed to the third phase for merging.

In the third phase, both feature models are merged into one. This process requires both feature models to have a similar structure, i. e., features have the same parent feature and properties in both models. While this requirement seems restrictive, it can be fulfilled by creating one of the two feature models and adapting a copy to fit to the other commit.

We discuss each of these phases in the following sections in greater depth. This also includes the presentation of implementation details and the design decision for the heuristics.

3.2 PREPARATION

In the preparation phase *FMstimator* parses inputs and collects data for the next phase. At first, *FMstimator* parses the given feature model files into an internal representation. For this, we use the python bindings of the VaRA feature library[2]. As part of this thesis, we also participated in the development of this library. The feature model is given in XML; the used format is based on the format that is also used by *SPL Conqueror*[3]. In order to include more information, we extended the format; most importantly a commit reference can be included as XML attribute and features can be annotated with code locations. Each code locations consist of five values: a file path, a start line and column, an end line and column, which uniquely identify the location of a feature variable in the source code.

In the next step, we collect the commit sequence, which we analyze in the second phase. For this, we use *pygit2*[4] (python bindings for *libgit2*). Starting at the commit belonging to the second feature model, we always move to the first-parent commit and record it until we find the commit belonging to the first feature model. At last, we need to verify that the target commit is part of the found sequence.

This method of obtaining the commit sequence limits the selection of initial and target commits depending on the second commit. If a commit is not in the line of first-parent of the second commit, it cannot be found with this method. However, in most repositories the interesting commits, e. g., releases, are compatible with our method. In this prototype, we wanted to evaluate the feature model approximation and location tracking, which is not

2 https://github.com/se-sic/vara-feature
3 https://www.se.cs.uni-saarland.de/projects/splconqueror/
4 https://www.pygit2.org/ and https://libgit2.org/

limited by this approach. In order to solve this shortcoming, we could implement a more sophisticated commit search. The commit history of Git is essentially a directed acyclic graph, so we could use a graph search algorithm, such as Dijkstra's algorithm [4], to find a path from the second commit to target commit and then continue the search to the first commit.

Depending on the chosen analysis type, *incremental* or *direct*, we create the sequences for the second phase. For an *incremental* analysis, we slice the found sequence at the target commit such that we have two sequences: one starting at the first commit and one starting at the second commit, both ending on the target commit. For a *direct* analysis we create two sequences containing two elements each. The first contains the commit corresponding to the first feature model and the target commit. The second contains the commit corresponding to the second feature model and the target commit.

3.3 TRACKING

In the tracking phase, we update the given code locations based on code differences and remove features from their feature model if their locations are lost. Algorithm 1 shows the tracking algorithm.

Algorithm 1: Feature tracking over commit chain

```
   Input: Feature Model FM, Commit List Commits
 1 OldCommit = Commits.PopFront()
 2 for each NewCommit in Commits do
 3     Diff = ComputeDifference(OldCommit, NewCommit)
 4     for each Feature in FM do
 5         if Feature.hasNoLocations() then
 6             continue
 7         for each Location in Feature do
 8             NewLocation = EstimateNewLocation(Location, Diff, OldCommit,
                NewCommit)
 9             if NewLocation found then
10                 Feature.UpdateLocation(Location, NewLocation)
11             else
12                 Feature.RemoveLocation(Location)
13         if Feature lost all locations then
14             Feature.MarkRemoved()
15     OldCommit = NewCommit
```

The loop in line 2 walks over all commits in the given commit list. It remembers the last one and computes the the difference to the current one, i. e., which changes were made. In our implementation, we use the default difference algorithm provided by *pygit2*[5]. We

5 https://www.pygit2.org/diff.html#pygit2.Repository.diff

also apply the `find_similar`[6] function to the diff to detect file renaming and moving. In the following, all features that are annotated with code locations are processed. Based on the known feature location and computed difference, a new feature location is estimated (line 8) with heuristics. If a new location could be estimated, the old location is updated, otherwise, it is removed. Should a feature lose all of its locations, it is marked as removed.

There are multiple cases of a feature's removal (or introduction) in the observed commit range which we need to consider. As long as one of the two remaining feature models contains a feature, it needs to be included in the final feature model that is built during the merging phase. The simplest case is a feature that persists throughout the tracked range and the feature should exist in both feature models after tracking. In the following, we list cases in which features are added or removed at different moments in the observed commit range and how they are handled.

- In the first case, consider a feature that is removed between the first and target commit. It is removed from the first feature model during tracking and is not part of the second one from the beginning. Neither of the resulting feature model have this feature.
- In the second case, consider a feature that is added between the first and target commit. It is not contained in the first feature model, but in the second one. While tracking the second feature model, the feature is not removed, hence, the second resulting feature model contains it.
- In the third case, consider a feature that is added between the target and second commit. It is not contained in the first model and during tracking of the second model the addition is detected, hence, none of the resulting models contain this feature.
- In the last case, consider a feature that is removed between the target and second commit. The feature is only contained in the first model and not in the second. During tracking it remains part of the first model.

Note that the last two cases mirror the first two, respectively. Removing a feature before the target commit is similar to adding a feature after it. They can be transformed by swapping both feature models and reversing the commit sequence, effectively changing the removal to an addition and moving it to the other side of the target commit.

We created a heuristic pipeline to estimate new feature locations. It consists of multiple heuristics that are queried sequentially. If one heuristic returns a new location, the pipeline stops and returns that location. This design decision also allows to easily append more heuristics if needed. We have designed three heuristics that compliment each other. While the first heuristic H_1 is detecting line movements by analyzing the given diff, the second heuristic H_2 tries to find new code that matches the old code. The third heuristic H_3 compares the AST of the old and new file version to find a new location. This design saves time by only executing the lowest heuristics possible to get a result.

6 https://www.pygit2.org/diff.html#pygit2.Diff.find_similar

3.3.1 *Tracking Heuristic*

Our tracking heuristics operate on the given difference between two commits and a location consisting of five values: path, start line, start column, end line, and end column. We write a location as <path(startline:startcolumn-endline:endcolumn)>.

```
40  // do something...
41  if (opt == 'c') {
42    compression = true;
43  }
44  // do more...
```

For example, imagine that we want to track the statement `compression = true;` in the code above. The code's location is <Foo.cpp(42:5-42:24)>[7].

Before we start with a deeper analyses, we check if the file, specified in the tracked location, was moved or changed. This information is provided by the computed diff and we update the old path to the new one, if the file was moved, or do nothing if the file was not changed.

If the file was changed, there are several types of code changes, which we need to cover. At first, we list our categorization of code changes. Then, we present the handling of these cases by our heuristics.

Case 1: Lines added/removed before:
When code lines are added or removed all of the following code is moved down or up, respectively.

Case 2: Lines added/removed after:
The tracked code is not moved by added or removed lines below.

Case 3: Tracked code is removed:
We can only observe that the code is removed, so we cannot provide a new code location.

Case 4: Tracked code is moved:
This is an extension of the third case. We can observe that the tracked code is removed, but we can also observe that it is added at a different location.

Case 5: Tracked code is modified
This case is objectively the hardest. We can observe that the tracked code is removed, but we do not observe that the same code is added at a different location.

The first two cases are handled by heuristic H_1. It is entirely based on counting added and removed lines for the file in the given location. Consider this example for the first case:

```
@@ -28,2 +28,5 @@
   // code
+  if (opt == 'h') {
+    print_help();
+  }
   // more code
```

7 We start counting at 1 for lines and columns; The end is marked by the first character that is not included.

In this example, three lines are added above, so the tracked code is also moved down three lines. We can either count added and removed lines or subtract the length of the hunk in the old version from the new version, if the tracked code is not part of the hunk. Of course, only lines are need to be considered that are before the beginning of tracked code.

Consider this example for the second case:

```
@@ -75,2 +75,5 @@
   // code
+  if (opt == 'h') {
+    print_help();
+  }
   // more code
```

In this example, three lines are added below, so the tracked code is not moved. In fact, we do not have to consider code changes after the tracked location at all. H_1 is the first heuristic in the pipeline and can handle both cases without failure, therefore, the remaining heuristic do not need to handle this case.

Consider the following code for the third case:

```
@@ -40,5 +40,2 @@
   // do something...
-  if (opt == 'c') {
-    compression = true;
-  }
   // do more...
```

In this example, three lines are deleted, one of them contains the tracked code. If these are the only lines, we have to assume that the code is removed indefinitely. In this case, no heuristic should find a correct new location. H_1 does not return a location if part of the given location is removed. H_2 and H_3 will search for a new location but fail, since the removed code is not added again.

To summarize the logic of H_1: If lines are added/removed before the tracked location, the code location shifted down/up accordingly. If lines are added/removed after the tracked location, the location is unaffected. If the tracked line is removed, the heuristic does not estimate a new location but reports it as removed.

The following cases are handled by H_2 and H_3. Both of heuristics try to relocate removed code. The heuristic H_2 tries to recover the tracked code from the added lines. Our implementation assumes that it is placed after H_1 in the heuristic pipeline, so the original code is assumed to be removed. It searches in the added lines of the diff for equivalent code and, if found, returns its location as new estimated location. We only consider added lines as potential new locations to prevent false mappings to existing duplicated code. H_3 works similarly, but on a semantical level. At first, it parses the old code to an AST and searches for a subtree that represents the tracked code. Then it parses the new version of the same file and tries to find an equivalent subtree. If successful, it returns the location of the code that the new subtree represents. Similar to H_2, the new location must be partially added in the given diff.

Consider the following diff for the fourth case:

```
@@ -40,5 +40,2 @@
   // do something...
-  if (opt == 'c') {
-    compression = true;
-  }
   // do more...
@@ -61,2 +58,5 @@
   // did more...
+  if (opt == 'c') {
+    compression = true;
+  }
   // still more to do...
```

Here, the tracked code was moved from line 42 to line 60. By searching the added lines for the tracked code we can find a correct new code location in line 60.

FMstimator also supports the tracking of code that spans more than a single line. In this case we have to consider that sometimes only part of the original code is added and handle it correctly. Consider the following code:

```
40  // do something...
41  if (opt == 'c') {
42    compression =
43      true;
44  }
45  if (opt == 'f') {
46    foo =
47      true;
48  }
49  if (opt == 'b') {
50    bar =
51      true;
52  }
53  // do more...
```

Here, we want to track the location `<Foo.cpp(42:5-43:12)>`. If the last conditional is removed and the first is moved to the end, we can receive the following diff:

```
@@ -40,6 +40,2 @@
   // do something...
-  if (opt == 'c') {
-    compression =
-      true;
-  }
   if (opt == 'f') {
@@ -48,4 +44,4 @@
   }
-  if (opt == 'b') {
-    bar =
+  if (opt == 'c') {
+    compression =
       true;
```

While the tracked code is removed completely in the first hunk, only a part of the tracked code is shown as added in the end. This diff gives instructions to remove only the first two lines of the last conditional and replace it with the first two of the first conditional; this allows for a shorter diff.

In order to successfully recover the code in this case, we have to be able to recover only parts of the code from the added lines but also find the remaining code. H_2 approaches this by collecting all added lines and splitting the tracked code into lines. In the next step, we search the content of split tracked code lines in the added lines, and if we find a matching pair, we can compute a possible new location based on the matched code. In this example, the tracked code spans two lines and we find a match for the first line, so we have to also check the next line for the remaining code. Lastly, we must check if the found location is actually equal to the original code; if it is not we have to continue searching. This solution does not allow the changes to reduce the number of tracked lines.

Next, consider the following diff for the same code example as before, however, in this case the second conditional is removed.

```
@@ -42,6 +42,2 @@
     compression =
-       true;
-   }
-   if (opt == 'f') {
-     foo =
        true;
```

This diff does delete part of the tracked code, but does not add any lines. We did not consider this case during implementation and are not able to handle it, albeit the new location is equal to the old new. There are two solutions to handle this.

H_1 returns no new location if part of the tracked location is in a removed line. Instead of aborting the tracking and returning no location, we could perform a comparison first between the original code and the code after tracking and return the new location if it is equal. Of course we would still need to consider line changes before the tracked code. H_2 and H_3 currently only considers added lines to recover code. If we also considered the immediate context lines for recovery, we could also find the correct location here.

The fifth case must be treated carefully. Our first two heuristics do not use any syntactic nor semantic knowledge, so we cannot decide if a change to the tracked code does not delete the tracked feature. The second heuristic allows changes to leading and trailing whitespaces in the tracked code, in order to detect changes in indentation.

To allow at least the tracking of syntactic changes, we implemented a heuristic that is based on the abstract syntax tree of the tracked code. A detailed description can be found below. Similarly to the second heuristic, it only considers added lines as new locations, so it also does not handle changes correctly if no part of the original code is added. The solution would be similar.

H_3 complements the previous heuristics. As noted before, the heuristics that are only based on line content do not use any deeper understanding of the code. This heuristic tries to find an AST subtree of the tracked code and rediscover it in the next commit. It is

implemented as a recovery strategy and is only used to find a new location if the tracked code is removed. Therefore, we only consider code if parts of it were added. Furthermore, this heuristic expects that there is an AST subtree that represents the tracked code.

At first, the file of the tracked code is parsed with *Clang* [8] to receive an AST. In the next step, we try to find the AST subtree that represents the tracked code. Starting at the AST's root node, we traverse the tree until we find a node, whose source location is the same as or inside of the tracked location. Since we move down the tree and a node's children's source location can only become more precise, we will eventually find the most coarse AST subtree, if the AST contains it.

Unfortunately, the python bindings of Clang are limited and do not expose all code location information that is known to the AST. This causes our implementation to sometimes miss the subtree, specifically, if the start or end of the tracked code contains pre-processor macros. In this case, the reported code location refers to the code that was inserted in place of the macro. We believe that the C++ API of Clang also offers the source locations before macros resolution, so this problem should be solvable with better API access.

If a correct subtree is found, we continue by also parsing the new version of the file to receive a second AST. In this second AST, we search for a subtree that is similar to the subtree that we found before. If we find one, we can use its source location as new location. As before, we still check if at least part of the new location was added.

This heuristic has several drawbacks. Firstly, it is language specific. The previous heuristics work on all text-based files; this one does only work on languages that are supported by Clang. Secondly, parsing source code is slow. This can be mitigated by using it only when the need arises, which our implementation does; this heuristics is only used if both previous heuristics fail. Furthermore, *Clang* must be able to find files included by the parsed files, otherwise the AST might be incomplete. *FMstimator* allows the user to pass arguments that are forwarded to the parsing process to allow the specification of non-standard include files. Ideally, this information should be obtained in a more autonomous manner.

Lastly, we acknowledge that these heuristics can fail to recognize that code is moved into a comment, or is no longer reachable. However, it is considered as bad coding style to move code into comments and most projects will not accept code contributions that do so. It is usually advised to remove code and retrieve it from the version control system if it is needed again. Furthermore, we do not expect the selected code to become unreachable, however, this also depends on the selected code locations, e. g., the command line parser is usually always reachable during a normal development history.

3.4 MERGING

For the last phase, we receive both final feature models from the previous phase. The goal of this phase is to merge both feature models into one. If a feature is contained in either of the given feature models, it should also be contained in the merged feature model; this can also be viewed as a union of both.

Algorithm 2 shows the merging algorithm. It receives a feature model and a feature model node as input. Initially, it should be called with the root feature of the other model that is merged.

8 https://clang.llvm.org/

Algorithm 2: Merging of two Feature Models

```
   Input: Feature Model FM, Feature SubtreeRoot
 1 Function mergeSubtree(FM, SubtreeRoot):
 2     if SubtreeRoot in FM then
 3         if not similar(SubtreeRoot, FM.get(SubtreeRoot)) then
 4             raise error
 5         else
 6             MergeGroupProperties()
 7             MergeLocations()
 8     else
 9         FM.addFeature(SubtreeRoot.copy(), SubtreeRoot.parent())
10     for each Child in SubtreeRoot.children() do
11         mergeSubtree(FM, Child)
```

In a first step, the algorithm checks if there is a feature in the feature model with the same name as SubtreeRoot[9]. If a matching feature is found, it continues to check if other properties are also similar, such as parent feature, feature kind (binary or numeric), optional or mandatory. This similarity check is implemented in a very strict manner. It requires the parents to have the same name and to be of the same kind. Furthermore, the features themselves also must have the same kind, binary features must both be optional or both mandatory, and numeric features must allow the same values. Should the matched features differ to strongly, the merging process is aborted. In this case, the user has to perform the merge manually.

If the features match, we try to merge potential potential group properties and locations, i. e., alternative groups. The used XML format does not allow to represent alternative groups with fewer than two children. If new children are added to a group it is possible that only one of the models is aware of the alternative group. Therefore, we allow to merge a feature with another that is the parent of an alternative group, *iff* the feature has less than two children, otherwise an error is raised. Merging locations is a union of the feature's locations in both models.

We decided to use a strict approach to ensure that the final model does not create new and possibly false constraints. For example, attaching a feature to a different parent also creates a different implicit implication. If changes to the structure of the feature model or to the properties of single features are needed, we want the user to resolve them instead of returning a model with false properties that is potentially used.

In the case that there is no matching feature to SubtreeRoot, we create a copy of the feature and add it. It is added as a child to a feature that shares the name of SubtreeRoot's parent. Such a parent must exist; either it existed from the beginning of the merging process or was added previously.

Lastly, the merging process is performed recursively with each child node of SubtreeRoot. This merging process does not consider cross-tree-constraints. Feature models do not carry

9 A feature's name is assumed to be unique.

sufficient information on the origin of constraints to safely merge them. However, we do preserve constraints that are embedded in the tree structure itself. While the similarity check seems very restrictive, it should not abort the merging process if it is considered during the creation of the initial feature models.

Finally, the merged model is written in XML-format. If the merging failed, we output the estimated feature models at the target commit in XML-Format, such that the user can merge them manually.

4 EXPERIMENTS

This chapter describes the experiments we conducted to analyze the precision and applicability of our approach, when applied to real world software projects.

Initially, we created a small project ourselves to test our approach in a familiar setting and test specific edge cases. Furthermore, we chose four open-source projects: *lrzip*, *vpxenc*, *Curl*, and *grep* to evaluate our precision on real world software projects.

All of the selected open-source projects are implemented in C and/or C++ and have tagged commits that belong to releases, allowing us to easily identify them. Furthermore, all projects offer a command-line interface, allowing us to map features to the corresponding command-line parser code. The experiments on open-source projects allow us to identify previously unconsidered strengths and weaknesses of our approach on a real development history.

For each project, except our own, we chose a commit range that covers several hundred commits. In the next step, we reviewed the present features at the beginning and the end. After selecting an initial set of features, we created a feature model for the later commit first and then adopted it to the earlier commit. We assume that software and, as consequence, feature models become increasingly complex during development, therefore, creating the later model first is reasonable to include the tree structures for, possible future, feature groups in the earlier model. The created feature models are not aimed to be complete but to include a representative set of features for the project. Where possible, features are annotated with a code location in the command line parser. Whenever possible, we included features that are added or removed in the selected range. A full list of features included in the initial feature models is included in Appendix A.

In each project, we selected three target commits in the commit range. If present, we selected tagged release commits. While selecting commits, we tried to space them evenly over the whole range. For each selected commit, we run our tool twice; once with incremental commit analysis and once with direct commit comparison. This experiment setup requires six tool executions, three target commits with two settings, for each project, so 30 in total.

For our experiments, we modified the behavior of the heuristic pipeline; If the first heuristic fails to update the location, we estimate a new location with both recovery heuristics. This allows us to compare estimated locations and provides more opportunities to evaluate the behavior of H_3. The result of H_2, if present, is preferred such that he output of the heuristic pipeline does not change.

The goal of these experiments is to answer the following research questions:

RQ 1: How common are code changes, which we cannot handle with the current heuristics?

RQ 2: How often is user intervention required to merge or correct the final feature model?

RQ 3: Is the incremental commit analysis more precise than the direct commit comparison?

The research questions are answered in the discussion of our evaluation.

In the following, we present each project and the selected commits. Furthermore, we present a simplified commit history of each project. The commit history is reduced to the start and end commit of the commit range and the three selected target commits. Additionally, we list relevant added and removed features between commits.

4.1 SYNTHETIC PROJECT

We implemented a small command line tool[1] that implements the context-sensitive image cropping algorithm *seam carving*[2]. The project's history deliberately contains changes to test our heuristics; This includes file renaming, feature deletion and addition, reordering of the command line parser, and changing the indentation style from two spaces to four spaces. The project contains 16 commits. We selected every fourth commit as our target commits.

The following list shows the hashes of selected commits and the feature changes until the next commit. Note that we interpret the replacement of a binary feature with a numeric feature as removal and addition of features. Selected commits:

- `e96e20d0`
- `5f6e17ea`
  - Add Sobel feature
- `103c787b`
  - Remove binary logging feature
  - Add numeric logging feature
- `fbf2c148`
  - Remove dualGradient feature
- `17dffc1f`

4.2 LRZIP

lrzip is a compression utility. Its name is an abbreviation for *Long Range ZIP* or *LZMA RZIP*. We chose `v0.600` as the first commit and `v0.630` as the second. Both feature model contain 12 features of which nine are annotated with a location. None of the features is removed or added in the observed commit range. This case study allows us to evaluate our approach with feature models that should remain unchanged except for the feature locations. All selected commits are listed below:

- `v0.600`
- `v0.610`
- `v0.614`
- `v0.620`
- `v0.630`

1 https://github.com/padupr/seamcarving
2 https://en.wikipedia.org/wiki/Seam_carving

4.3 vpxenc

vpxenc is a command-line video encoding tool for the codecs VP8 and VP9. It is part of *libvpx* that serves as reference implementation of both codecs. We chose to run our experiments in the commit range between `v1.2.0` and `v1.3.0`. The VP9 codec was officially introduced in release `v1.3.0`, so we expect to changes to the argument parser in this range.

The command-line option parser in this project is not completely straightforward, so we will discuss it briefly. The source code specifies possible options in structs. These include a short and long name, if it has a value, and a description. The command-line parser tries to match the given options against the structs. For general options, valid for both codecs, it uses a standard cascade of if-conditionals, however, options that are specific to a codec are matched in a loop, that does not have any code specific to the options. Therefore, if the option is not explicitly parsed in the argument parser, we selected the definition of the appropriate struct. The selected commits and feature changes are listed below:

- `v1.2.0`
  - Add `lossless` feature
  - Add `frameparalleldecoding` feature
- `82c415328`
- `adfc54a46`
- `660dcfe6a`
- `v1.3.0`

4.4 grep

grep is a utility to search files for lines matching specified pattern.

We conduct our experiment between the release commits `v2.20` and `v3.6`. Both feature models have a total of 12 features of which we annotated 10 with a location. In the observed range none of the features are added or removed. All selected commits are listed below:

- `v2.20`
- `v2.25`
- `v3.0`
- `v3.3`
- `v3.6`

4.5 curl

Curl is a utility to transfer files. It supports a variety of web protocols. The project documents the first version in which an option is introduced[3], which makes it a great case study, because we know every feature addition. Unfortunately, the document does not list when features were removed. We conduct our experiment between the release commits `curl-7_30_0` and `curl-7_40_0`. The feature model for the first commit has a total of 17 features of which we

3 https://github.com/curl/curl/blob/curl-7_77_0/docs/options-in-versions

annotated 11 with a location. The second feature model has a total of 26 features of which we annotated 20 with a location.

The following list shows the tags of selected commits and the feature changes until the next commit.

- `curl-7_30_0`
  - Add `sasl-ir` feature
- `curl-7_32_0`
  - Add `http1.1` feature
  - Add `http2` feature
  - Add `tlsv1.0` feature
  - Add `tlsv1.1` feature
  - Add `tlsv1.2` feature
- `curl-7_35_0`
  - Add `no-alpn` feature
  - Add `no-npn` feature
- `curl-7_38_0`
  - Add `pinned-pubkey` feature
- `curl-7_40_0`

5 EVALUATION

In this chapter, we present the results of our experiments. First, we explain how we evaluate the experiments to answer each research question. We discuss the results for each case study separately and highlight specific problems and errors of our heuristics. In Section 5.7, we provide an overview over all case studies. In the end, we discuss the results and possible threats of validity to our evaluation.

We present the statistics for each case study separately and show them in a table. The table shows the target commit, if analysis was done incrementally (inc.) or direct (dir.), analyzed diffs in each tracking direction, expected failures of a perfect heuristic H^*, and for each heuristic successes (✓) and failures (✗). The perfect heuristic H^* serves as ground truth to which we compare our heuristics. A perfect heuristic is unable to estimate a new location, *iff* the feature was removed. If a heuristic returns a new location, it is counted as a success. The amount of failures of H^* is the sum of features removed before the target commit and features added after the target commit. Keep in mind that more failures than expected do not necessarily result in missing features in the final model due to possibly redundant features in both initial features. The final feature models and log files of the experiments are available on Github[1]

5.1 OPERATIONALIZATION

In the following, we describe our methodology to answer the research questions.

For **RQ 1**, we investigate the unexpected failures of heuristics H_2 and H_3. Before the experiments, we determined which features should be removed in each experiment. If a heuristic is unable to estimate a new location for a feature that should not be removed, we count it as an unexpected failure. Furthermore, we also check the final feature locations. We chose the feature locations in both initial feature models such that they should be equal in the final models. If they differ, this hints at an error during tracking. We also present the problematic code changes in each project and discuss possible solutions in Section 5.8.

For **RQ 2**, we check if the final feature model is correct. We check if all features are included as expected and if the tree structure, i. e., parent-child relationships and groups, is correct.

For **RQ 3**, we compare the results of the experiments with incremental analysis against direct analysis for the same target commit. We are especially interested in cases where one of the analysis types is able to retain a feature and the other loses it incorrectly. We anticipate that incremental analysis calls the heuristics more often since it also process more changes. Here, we are only interested in a difference of heuristics failures.

1 https://github.com/padupr/FMstimator

5.2 SYNTHETIC REPOSITORIES

The statistics for this case study are shown in Table 5.1. In the first two experiments, we expect two features to be removed: the features *Sobel* and the numeric *logging* feature should be removed from the second feature model during tracking from `fbf2c148` to `5f6e17ea`. In the experiments for target commit `103c787b`, we only expect the numeric *logging* feature to be removed. In the last two experiment, the binary *logging* feature should be remove during forward tracking.

Table 5.1: Statistics of experiments on synthetic repository. The columns show the target commit, if type of analysis was incremental (inc.) or direct (dir.), analyzed diffs in forward (fw.) and backwards (bw.) tracking direction, and successes and failures of each heuristic

Commit	Type	Diffs		Heuristics						
		fw.	bw.	H^*	H_1		H_2		H_3	
				✗	✓	✗	✓	✗	✓	✗
`5f6e17ea`	inc.	3	12	2	54	7	5	2	5	2
`5f6e17ea`	dir.	1	1	2	5	5	3	2	3	2
`103c787b`	inc.	7	8	1	58	6	5	1	5	1
`103c787b`	dir.	1	1	1	5	5	4	1	4	1
`fbf2c148`	inc.	11	4	1	57	6	5	1	5	1
`fbf2c148`	dir.	1	1	1	4	6	5	1	5	1

Concerning **RQ 1**, we have found one change example, that H_2 is not handling correctly. While H_2 and H_3 are both able to find locations when expected, the returned location is different for two features. We discuss only one of these cases, because they are similar.

The varying locations occurs during the analysis of a diff that changes the indentation style of the whole project. This diff removes most lines to add them again but adapted to the new indentation style. The argument parser declares several variables to store the parsed option values but also assigns a default value, i. e., `bool vertical = true;`. However, it also offers to explicitly specify the default value over command line options, i. e., `vertical = true;`. We are tracking the latter, so when the line is removed, H_2 searches the added lines for the old code. Both lines are removed and added again with different indentation, and are viable candidates to track for H_2, however, H_2 always chooses the first match, which is the wrong decision in this case. H_3 does not consider the variable declaration due to the different AST structure and chooses the correct location.

Concerning **RQ 2**, in this case study, features were removed as expected and the resulting feature models are also correct.

Concerning **RQ 3**, the results of incremental and direct analysis does not differ. H_2 and H_3 reported the same amount of failures for both analysis types. Both analysis types also produce the wrong location described before.

5.3 LRZIP

The statistics for this case study are shown in Table 5.2. H^* has no failures since no features are removed in the observed history.

Table 5.2: Statistics of experiments on lrzip. The columns show the target commit, if type of analysis was incremental (inc.) or direct (dir.), analyzed diffs in forward (fw.) and backwards (bw.) tracking direction, and successes and failures of each heuristic

Commit	Type	Diffs		Heuristics						
		fw.	bw.	H^*	H_1		H_2		H_3	
				✗	✓	✗	✓	✗	✓	✗
v0.610	inc.	100	162	0	198	9	0	9	0	9
v0.610	dir.	1	1	0	9	9	0	9	0	9
v0.614	inc.	155	107	0	153	9	0	9	0	9
v0.614	dir.	1	1	0	9	9	0	9	0	9
v0.620	inc.	205	57	0	117	9	0	9	0	9
v0.620	dir.	1	1	0	9	9	0	9	0	9

Concerning **RQ 1**, all tracked features are lost during forward tracking in the same step. *lrzip* stores the results of the command line parser in a dedicated struct. Initially, the command line parser writes to the struct with a structure reference, i. e., `a.b`. However, the structure reference is replaced with a structure dereference, i. e., `a->b`. H_2 is unable to find a new location, because the source code has changed and H_3 is unable to find a new location, because there is no matching AST subtree.

Concerning **RQ 2**, while the features were incorrectly removed in one feature model, they were not removed in the other feature model. This allowed the merging algorithm to create a correct feature model for all experiments of this case study.

Concerning **RQ 3**, the results of incremental and direct analysis does not differ. H_2 and H_3 reported the same amount of failures for both analysis types.

5.4 VPXENC

The statistics for this case study are shown in Table 5.4. H^* has no failures since no features are removed in the analyzed range. We would observe the removal of *lossless* and *frameparalleldecoding* during backwards tracking from `82c415328` to `v1.2.0`.

Concerning **RQ 1**, H_2 works as expected and is always able to provide a new location. H_3, on the other hand, repeatedly fails to provide a new location. In all of these cases the heuristic already fails to find an AST subtree in the old version. While parsing the source code, *clang* is unable to find `vpx_config.h`. This file is created by the project's build system and is not present in the repository. `vpx_config.h` defines a series of preprocessor directives to configure the compilation process. If it is missing, the preprocessor does not include certain source code, among others, this also removes the source code mapped to some of our features, which therefore is not included in the AST.

Table 5.3: Statistics of experiments on vpxenc. The columns show the target commit, if type of analysis was incremental (inc.) or direct (dir.), analyzed diffs in forward (fw.) and backwards (bw.) tracking direction, and successes and failures of each heuristic

Commit	Type	Diffs		Heuristics						
		fw.	bw.	H^*	H_1		H_2		H_3	
				✗	✓	✗	✓	✗	✓	✗
82c415328	inc.	306	912	0	275	11	11	0	7	4
82c415328	dir.	1	1	0	13	9	9	0	6	3
adfc54a46	inc.	610	608	0	275	11	11	0	7	4
adfc54a46	dir.	1	1	0	13	9	9	0	6	3
660dcfe6a	inc.	914	304	0	263	11	11	0	7	4
660dcfe6a	dir.	1	1	0	13	9	9	0	6	3

Concerning **RQ 2**, all final feature models are correct.

Concerning **RQ 3**, while all feature models are correct, we found a different result in the estimated locations. In the experiment runs for target commit `660dcfe6a` the incremental analysis is able to keep track of the correct code for *goodQuality* in the argument parser. However, the direct analysis chooses the wrong code location during recovery. In this case, H_2 and H_3 both pick the same new location. The new location is a new assignment that enables *goodQuality* by default. While this new location is also evidence of the features existence, the removal of this code is not a sign of the features removal, only a sign that it is no longer enabled by default, therefore we consider the new location as wrong. The incremental analysis handles this situation correctly, since the diff that introduces the new default assignment does not remove the correct location and H_1 is able to provide a correct update.

5.5 GREP

The statistics for this case study are shown in Table 5.4. H^* has no failures since no features are removed in the observed history.

Concerning **RQ 1**, the results of this case study are very similar to those of *lrzip*. While no features and, in consequence, no locations should be lost, the heuristics fail several times. Here, the errors occur in two different steps. Between `v2.20` and `v2.25` the argument parser is partially updated. The type of some variables is changed from `integer` to `bool` and, in consequence, the assigned values are also updated accordingly, e. g., `foo = 1` to `foo = true`. Between `v2.25` and `v3.0` the argument parser is updated further. Previously, the pattern type was set by

```
case 'E':
  setmatcher ("egrep");
```

Table 5.4: Statistics of experiments on grep. The columns show the target commit, if type of analysis was incremental (inc.) or direct (dir.), analyzed diffs in forward (fw.) and backwards (bw.) tracking direction, and successes and failures of each heuristic

Commit	**Type**	**Diffs**		**Heuristics**						
		fw.	**bw.**	H^*	H_1		H_2		H_3	
				✗	✓	✗	✓	✗	✓	✗
`v2.25`	inc.	238	436	0	1,126	9	0	9	0	9
`v2.25`	dir.	1	1	0	11	9	0	9	0	9
`v3.0`	inc.	444	230	0	969	9	0	9	0	9
`v3.0`	dir.	1	1	0	11	9	0	9	0	9
`v3.3`	inc.	531	143	0	798	9	0	9	0	9
`v3.3`	dir.	1	1	0	11	9	0	9	0	9

but in the newer version it is changed to

```
case 'E':
  matcher = setmatcher ("egrep", matcher);
```

This is one of the most complex case, which we encountered. We can observer two significant changes. The signature of the used function has changed, and the return value is saved.

Concerning **RQ 2**, although several features are incorrectly lost before merging, they were still present in one of the feature models that were passed to merging algorithm. The final feature models are all correct.

Concerning **RQ 3**, the final feature models were equal and the heuristics H_2 and H_3 also fail equally often.

5.6 CURL

The statistics for this case study are shown in Table 5.5. In the selected commit range, features are only added but not removed, so we only have feature deletions in the backwards analysis. While tracking changes from `curl-7_40_0` to `curl-7_38_0` the feature *pinned-pubkey* should be removed. Continuing to `curl-7_35_0` the features *no-alpn* and *no-npn* should be removed. In the last section, five features should be removed: *http1.1,2*, *tlsv1.0,1,2*.

Concerning **RQ 1**, we have found two cases that are handled incorrectly. In the first, the features *trace* and *verbose* are removed incorrectly during tracking changes between `curl-7_35_0` and `curl-7_38_0`. They are mutually exclusive and which option is selected is saved in the same variable:

```
1 case 'g':
2   config->tracetype = TRACE_BIN;
3 // other cases
4 case 'v':
5   config->tracetype = TRACE_PLAIN;
```

The feature is lost when the struct member `tracetype` is moved from the `config` into `global` or vice versa. Due to the code change, H_2 is unable to provide a new location.

Table 5.5: Statistics of experiments on Curl. The columns show the target commit, if type of analysis was incremental (inc.) or direct (dir.), analyzed diffs in forward (fw.) and backwards (bw.) tracking direction, and successes and failures of each heuristic

Commit	Type	Diffs		Heuristics						
		fw.	bw.	H^*	H_1		H_2		H_3	
				✗	✓	✗	✓	✗	✓	✗
curl-7_32_0	inc.	345	2,376	8	753	13	2	11	0	13
curl-7_32_0	dir.	1	1	8	18	13	2	11	0	13
curl-7_35_0	inc.	1,088	1,633	3	751	7	2	5	0	7
curl-7_35_0	dir.	1	1	3	24	7	2	5	0	7
curl-7_38_0	inc.	2,022	699	1	563	5	2	3	0	5
curl-7_38_0	dir.	1	1	1	26	5	2	3	0	5

In the second case, *sasl-ir* is incorrectly removed. The command line parser of *Curl* allows to explicitly enable or disable each option, so every `--option` has a respective `--no-option`. Before looking up which option it is currently processing the parser check if it starts with `--no-` and saves is in the variable `toggle`. The value of `toggle` is later copied to the config-variable to enable or disable it. The command line parser of version `curl-7_32_0` does not allow to explicitly disable *sasl-ir*.

```
1 case 'K': /* --sasl-ir */
2 config->sasl_ir = TRUE;
```

A few commits later, this is updated to

```
1 case 'K': /* --sasl-ir */
2 config->sasl_ir = toggle;
```

According to the commit message, this change was done to several options. Due to the change of the assigned value, H_2 is unable to provide a new location.

Note that H_3 is not able to find an AST subtree that fits to the original code. While parsing the given file, *Clang*'s preprocessor is unable to include `curlbuild.h`. This file is not present in the repository, but is created by the build system.

Concerning **RQ 2**, the resulting feature models are correct except for the feature models of `curl-7_32_0`, which are missing *sasl-ir*. The responsibility to notice and repair this error is on the user. We cannot warn the user of this error, since that would require us to be aware of the false feature removal before.

Concerning **RQ 3**, there is no difference between incremental and direct analysis in this case study.

5.7 RESULTS

In this section, we provide a brief summary of the results over all case studies.

RQ 1: How common are code changes, which we cannot handle with the current heuristics? Each case study has shown changes that none of our heuristics could han-

dle or were handled incorrectly resulting in a false location. We have also seen that H_3 is often unable to function properly. H_2 could not handle all code changes in three out of five case studies. H_3 could not handle all code changes in four out of five case studies.

RQ 2: How often is user intervention required to merge or correct the final feature model? Our tool was able to create a correct feature model in all but two cases. The two remaining cases in the *Curl* case study only require the addition of one feature.

RQ 3: Is the incremental commit analysis more precise than the direct commit comparison? The incremental analysis and direct comparison produced a different result in one case. In the case study on *vpxenc* the direct analysis produced a false location. This location is still related to the feature, however, not as strongly as the correct one. Overall, heuristics H_2 and H_3 have shown the same amount of failures for the same target commits and the merged feature models are also equal.

5.8 DISCUSSION

We discuss the implications to our approach of the results for RQ 2 and RQ 3 first, and then discuss the results for RQ 1 and the found problematic changes and errors we found in the case studies.

The evaluation to RQ 2 shows that our merging approach is working as intended. The only case in which it failed is caused by an error in the tracking phase. The case study on *lrzip* and *grep* highlights the value of redundant information in two feature models to our approach. Even if a feature is incorrectly removed in one of the tracking directions it can be included in the final results if the other feature model retains it. This mechanic makes the final feature model more robust against errors in the tracking phase.

For RQ 3, we evaluated if an incremental commit analysis is useful or if a we can skip to the end immediately with a direct analysis. We found one case in which the heuristics produced different results. Our heuristics are currently too strict to allow a large difference in behavior. If we create a heuristic to accept small changes to the tracked code, these changes could accumulate over multiple steps. However, we usually encountered only a single change to the tracked code, so the the incremental analysis might be unnecessary if the user is only interested in the final feature model. The incremental analysis offers a better understanding of the time and reason a feature was removed.

In the evaluation for RQ 1, we have seen multiple problems in our heuristics. In our synthetic project H_2 found a false location, and in *vpxenc* H_2 and H_3 both found the same false location. Two solutions come to mind: Heuristics could take the surrounding code into account to narrow down possible new locations, or they could calculate an assumed position and prefer locations in its vicinity, e. g., we prefer a location if its close to the old one. In general, the next revision of heuristics should make a more elaborate choice when they find multiple possible candidates instead of taken the first.

The errors in the case study on *lrzip* have shown that our heuristics must allow some changes to the tracked code.

However, the allowed changes must be selected carefully and the accumulative effect of incremental analysis must be considered.

grep has presented two types of changes, we could not handle. First, it changed variable's types and in the following also the default values. While this could be solved with certain

allowed changes, a more general solution is desirable. The second case involves the introduction of a new variable and a modified function signature. A safe and general solution to this case is not possible without significantly altering the heuristics. Generally, we do not think that this case should be handled with high priority without further assessment to the prevalence of changes of this extent.

Curl has also presented two types of unhandled changes. First, the code is slightly changed, such that a different value is assigned. Second, the variable associated to a feature is moved to a different struct. In our current heuristics, both are not easy to solve. However, these changes suggest that instead of tracking statements, we could possibly track variable declarations and definitions that are connected to features.

We categorize the unhandled change in three groups: *syntactic modifications*, *semantic modifications*, and *code refactoring*. The *syntactic modifications* includes formatting changes. Formatting changes can cause our heuristics to recover incorrect source code, i. e., the estimated location is not the intended one. This type change occurs if the tracked code is moved to a new location and equivalent code is also added before the new location of the added code. The challenge to solve in this category is the selection of the correct source code location our of multiple. A more informed decision process is required by the heuristics. The *semantic modifications* comprises small changes to the source code that extend past formatting changes. This includes most of the previously described changes, like type changes as seen in *grep*, switches between a structure reference and dereference, or moving a variable that stores the features state to a different struct. The *code refactoring* includes larger changes to the code, e. g., changes to function signatures. We have only encountered this in *grep* when the parsing of the pattern options was updated, i. e., a function call got changed to a variable assignment of the return value of the call.

5.9 THREATS TO VALIDITY

In this chapter we discuss threads to validity that arise for our evaluation. First, we discuss internal validity, and, afterwards, focus on external validity.

5.9.1 *Internal Validity*

A threat to validity are possible bugs in our code. We created several tiny projects to test our tool. Each project consists of exactly two commits and we created a feature model for each that also contains the expected locations.

Another threat to validity are bugs in the used libraries. The VaRA feature library is still in early development and not widely used. While the library has tests, it is not bug free. For the evaluation, we implemented bug fixes where needed and will contribute them to the project with tests later. *libgit2* and *libclang* are widely used and well tested. We consider it unlikely to be affected by any bugs in them.

Lastly, the tool is unable to verify if a given location is correct and cannot detect false locations. For now, the locations must be entered manually in the feature model files and it is possible that the users misspells them. To mitigate this problem, our tool displays the features with location and the code at that location for the feature model's commit. This allowed to check if the location was entered correctly.

5.9.2 *External Validity*

One threat to external validity is the selection of projects for our case studies. We tried to mitigate this by selecting projects from different domains, i. e., image processing in our synthetic project, file compression in *lrzip*, video encoding in *vpxenc*, plain-text filtering in *grep*, and a network utility in *Curl*.

Furthermore, all selected projects are implemented in C or C++. Different programming languages might require different heuristics. H_3, for example, only works on languages in the C-family. We expect similar results for other languages, however, expanding the tool to handle more languages was out of scope of this thesis.

Another threat to validity is the selection of code locations. In order to have comparable results, we decided to choose locations in the argument parser. This limits our insights in possible code changes that might occur in other parts of the projects. However, while a domain expert might be able to quickly locate other parts of the projects that are connected to the feature, an average user usually is not. The argument parser is often easy and fast to locate in a new project.

6 RELATED WORK

Nieke et al. [14] proposed a new approach to feature models. They suggested the use of a meta model to describe the planned evolution or document past evolutions. Our tool can assist the documentation of past evolution by providing an approximate feature model. This allows easier adoption of the proposed meta models if past evolution is not documented.

Feichtinger et al. [6] developed an approach to extract feature dependencies from source code in order to aid development. Their approach facilitates the mapping of features to code locations as well. They successfully demonstrated their tool in different case studies. We believe that our tool could benefit from their research by checking the final feature models with their approach and implementing the given suggestions.

Our approach relies heavily on the quality of the calculated differences and the information we can mine from their results. A study on different diff algorithms was conducted by Nugroho et al. [15]. In their work, they compare Git's default diff algorithm *Myers* against *histogram*. While they advise developers to use *historgram* for code comparisons, there is no explicit recommendation for analysis tools using diff algorithms. However, they also report that the choice of diff algorithms has a noticeable impact on results. Their research inspires an evaluation of the effect of the chosen diff algorithm for our work.

There are also diff algorithms that target source code instead of general text. The change distilling approach by Fluri et al. [7] use *bigram string similarity* to match statements and *subtree similarity* by Chawathe et al. [3] to match syntactic structures in ASTs. Falleri et al. [5] build an algorithm that works at an AST granularity and also detects node movement instead of representing it as a deletion and addition at its new location.

Baxter et al. [2] presented an approach to detect and remove code duplication using ASTs. In their work they present a metric to measure subtree similarity, which we could utilize in a heuristic that allows small syntactic and/or semantic changes.

7 CONCLUDING REMARKS

In this chapter, we summarize the contributions of this thesis and discuss possible improvements to *FMstimator*.

7.1 CONCLUSION

Performance analysis and prediction of configurable software view the subject as black box in most cases. White box approaches require a connection between a feature and associated source code. While it is possible to supply manual mappings from feature to source code in small studies, this technique does not scale for larger studies.

Our approach tackles this issue by approximating a feature model for a target commit, reducing the need of manual labor. Starting from two feature models for a project annotated with feature source locations, we perform a stepwise analysis over the project's Git history towards the target commit. Based on the feature source locations, we update the feature models to reflect the changes to the source code. The analysis utilizes three heuristics that analyze the diff between two Git commits to estimate an updated feature source locations for the next step. The used heuristics are very strict and only allow changes to the formatting of the source code. In a final step, we merge the two feature models, approximated for the target commit, into a single approximated feature model for the target commit.

We implemented our approach in *FMstimator* and evaluated it in experiments on five projects. The evaluation revealed source code changes that need to be handle beyond formatting changes. Furthermore, we learned that the use of two feature models hardens the approach against incorrectly missing features in the final feature model. As long as a feature is present in one of the two feature models it is also included in the merged feature model. Lastly, we found evidence that an incremental analysis, applying changes step wise to the feature model, is beneficial over a direct analysis, that applies all changes at once on the feature model.

Overall, our approach is promising but requires some improvements. Our approach approximates correct feature models in a majority of our experiments, successfully reducing the required manual labor. In its current state, the final feature models should be used with caution, since they can be slightly inaccurate. Furthermore, we identified the problematic code changes that caused our heuristics to fail in the experiments and most of them can be handled with more sophisticated heuristics.

7.2 FUTURE WORK

We have demonstrated that our approach is promising, however, our tool still needs improvements and further evaluation, which we discuss in the following.

Our tool requires several improvements, which we have mentioned before and briefly summarize here. The creation of the commit sequences in the first phase must be extended

such that it considers more paths than just the first-parents. We discussed several problems of our heuristics in Section 5.8 and the suggested solutions should be implemented. Another very prominent problem is the inability of H_3 to work on some projects. This must be improved in the future to enable a better analysis and open the path to better heuristics. While a fully automated process is preferable, a user-supplied script to enable the analysis might also be an acceptable intermediate solution.

Furthermore, we should eventually support cross-tree-constraints and consider them when updating the models. Especially, the handling of constraints containing a feature that is removed from the model is interesting. Ideally, common constraints are handled autonomously, and if needed the user is asked for support.

We currently have a limited understanding of the importance of the chosen feature location. An in-depth study is needed that explores possible advantages or disadvantages of chosen location and number of chosen locations. In extension to this, a categorization of code locations could help to improve our tool's performance. Some code locations, for example enabling the feature by default, are not essential to a feature's existence, and their deletion does not suggest that the feature was removed, however, it is evidence that the feature still exists, if other locations are removed. This categorization allows a better interpretation of observed changes.

Of course, the development of new heuristics is also important. The current heuristics are not flexible enough and must allow some changes to allow a correct tracking of source code locations. A possible new heuristic could employ the approach of Baxter et al. [2] to detect code duplication on an AST level to detect possible new code locations.

A APPENDIX

Table A.1 to Table A.10 list the selected feature and chosen location for the experiments.

Table A.1: Selected features for synthetic project at commit `e96e20d0`

Name	Source Code Location
horizontal	`<Main.cpp(30:7-30:24)>`
vertical	`<Main.cpp(33:7-33:23)>`
dualGradient	`<Main.cpp(46:9-46:39)>`
gradient	`<Main.cpp(44:9-44:35)>`
logging	`<Main.cpp(27:7-27:22)>`

Table A.2: Selected features for synthetic project at commit `17dffc1f`

Name	Source Code Location
horizontal	`<main.cpp(39:13-39:30)>`
vertical	`<main.cpp(42:13-42:29)>`
gradient	`<main.cpp(53:17-53:43)>`
sobel	`<main.cpp(55:17-55:40)>`
logging	`<main.cpp(31:13-31:41)>`

Table A.3: Selected features for *lrzip* at commit `v0.600`

Name	Source Code Location
compressionBzip2	`<main.c(545:4-545:41)>`
compressionDisabled	`<main.c(594:4-594:38)>`
compressionGzip	`<main.c(566:4-566:40)>`
compressionLzo	`<main.c(584:4-584:39)>`
compressionZpaq	`<main.c(663:4-663:40)>`
level	`<main.c(587:4-587:45)>`
maxWindowSize	`<main.c(658:4-658:34)>`
processorCount	`<main.c(618:4-618:35)>`
thresholdTestingDisabled	`<main.c(638:4-638:40)>`

Table A.4: Selected features for *lrzip* at commit `v0.630`

Name	**Source Code Location**
compressionBzip2	`<main.c(339:4-339:42)>`
compressionDisabled	`<main.c(402:4-402:39)>`
compressionGzip	`<main.c(366:4-366:41)>`
compressionLzo	`<main.c(389:4-389:40)>`
compressionZpaq	`<main.c(486:4-486:41)>`
level	`<main.c(392:4-392:46)>`
maxWindowSize	`<main.c(481:4-481:35)>`
processorCount	`<main.c(430:4-430:36)>`
thresholdTestingDisabled	`<main.c(460:4-460:38)>`

Table A.5: Selected features for *vpxenc* at commit `v1.2.0`

Name	**Source Code Location**
allowResize	`<vpxenc.c(1856:13-1856:66)>`
arnrMaxFrames	`<vpxenc.c(1136:1-1137:62)>`
arnrStrength	`<vpxenc.c(1138:1-1139:59)>`
autoAltRef	`<vpxenc.c(1134:1-1135:79)>`
passes	`<vpxenc.c(1609:13-1609:51)>`
bestQuality	`<vpxenc.c(1627:13-1627:52)>`
goodQuality	`<vpxenc.c(1629:13-1629:52)>`
rtQuality	`<vpxenc.c(1631:13-1631:48)>`
threads	`<vpxenc.c(1835:13-1835:58)>`
tokenParts	`<vpxenc.c(1132:1-1133:81)>`

Table A.6: Selected features for *vpxenc* at commit `v1.3.0`

Name	Source Code Location
allowResize	`<vpxenc.c(1382:7-1382:60)>`
arnrMaxFrames	`<vpxenc.c(543:1-544:70)>`
arnrStrength	`<vpxenc.c(545:1-546:67)>`
autoAltRef	`<vpxenc.c(541:1-542:87)>`
frameparalleldecoding	`<vpxenc.c(562:1-563:79)>`
lossless	`<vpxenc.c(560:1-560:81)>`
passes	`<vpxenc.c(1144:7-1144:45)>`
bestQuality	`<vpxenc.c(1159:7-1159:46)>`
goodQuality	`<vpxenc.c(1161:7-1161:46)>`
rtQuality	`<vpxenc.c(1163:7-1163:42)>`
threads	`<vpxenc.c(1363:7-1363:52)>`
tokenParts	`<vpxenc.c(535:1-536:81)>`

Table A.7: Selected features for *grep* at commit `v2.20`

Name	Source Code Location
count	`<src/grep.c(2063:9-2063:27)>`
ignore-case	`<src/grep.c(2112:9-2112:25)>`
invert-match	`<src/grep.c(2163:9-2163:27)>`
line-buffered	`<src/grep.c(2242:9-2242:27)>`
no-message	`<src/grep.c(2159:9-2159:29)>`
silent	`<src/grep.c(2146:9-2146:27)>`
basic	`<src/grep.c(2022:9-2022:29)>`
extended	`<src/grep.c(2010:9-2010:30)>`
pearl	`<src/grep.c(2018:9-2018:29)>`
string	`<src/grep.c(2014:9-2014:30)>`

Table A.8: Selected features for *grep* at commit `v3.6`

Name	**Source Code Location**
count	`<src/grep.c(2599:9-2599:30)>`
ignore-case	`<src/grep.c(2664:9-2664:28)>`
invert-match	`<src/grep.c(2719:9-2719:27)>`
line-buffered	`<src/grep.c(2811:9-2811:30)>`
no-message	`<src/grep.c(2715:9-2715:32)>`
silent	`<src/grep.c(2702:9-2702:30)>`
basic	`<src/grep.c(2557:9-2557:48)>`
extended	`<src/grep.c(2545:9-2545:49)>`
pearl	`<src/grep.c(2553:9-2553:48)>`
string	`<src/grep.c(2549:9-2549:49)>`

Table A.9: Selected features for *Curl* at commit `curl-7_30_0`

Name	**Source Code Location**
compressed	`<src/tool_getparam.c(472:9-472:35)>`
dump-header	`<src/tool_getparam.c(1123:7-1123:44)>`
http1.0	`<src/tool_getparam.c(874:7-874:51)>`
include	`<src/tool_getparam.c(1273:7-1273:40)>`
ipv4	`<src/tool_getparam.c(890:7-890:30)>`
ipv6	`<src/tool_getparam.c(894:7-894:30)>`
no-keepalive	`<src/tool_getparam.c(769:9-769:52)>`
ssl-allow-beast	`<src/tool_getparam.c(1202:11-1202:44)>`
tlsv1	`<src/tool_getparam.c(878:7-878:51)>`
trace	`<src/tool_getparam.c(423:9-423:39)>`
verbose	`<src/tool_getparam.c(1562:9-1562:41)>`

Table A.10: Selected features for *Curl* at commit `curl-7_40_0`

Name	**Source Code Location**
compressed	<src/tool_getparam.c(581:9-581:35)>
dump-header	<src/tool_getparam.c(1275:7-1275:44)>
http1.0	<src/tool_getparam.c(1000:9-1000:53)>
http1.1	<src/tool_getparam.c(1004:9-1004:53)>
http2	<src/tool_getparam.c(1008:9-1008:53)>
include	<src/tool_getparam.c(1421:7-1421:40)>
ipv4	<src/tool_getparam.c(1042:7-1042:30)>
ipv6	<src/tool_getparam.c(1046:7-1046:30)>
no-alpn	<src/tool_getparam.c(539:9-539:47)>
no-keepalive	<src/tool_getparam.c(878:9-878:52)>
no-npn	<src/tool_getparam.c(529:9-529:46)>
pinnedpubkey	<src/tool_getparam.c(1363:9-1363:48)>
sasl-ir	<src/tool_getparam.c(971:9-971:34)>
ssl-allow-beast	<src/tool_getparam.c(1354:11-1354:44)>
tlsv1	<src/tool_getparam.c(1016:9-1016:53)>
tlsv1.0	<src/tool_getparam.c(1020:9-1020:55)>
tlsv1.1	<src/tool_getparam.c(1024:9-1024:55)>
tlsv1.2	<src/tool_getparam.c(1028:9-1028:55)>
trace	<src/tool_getparam.c(526:9-526:39)>
verbose	<src/tool_getparam.c(1703:9-1703:41)>

BIBLIOGRAPHY

[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.

[2] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377, 1998.

[3] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change Detection in Hierarchically Structured Information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, page 493–504. ACM, 1996.

[4] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *NUMERISCHE MATHEMATIK*, 1(1):269–271, 1959.

[5] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and Accurate Source Code Differencing. In *Proceedings of the International Conference on Automated Software Engineering*, pages 313–324, Västeras, Sweden, 2014.

[6] Kevin Feichtinger, Daniel Hinterreiter, Lukas Linsbauer, Herbert Prähofer, and Paul Grünbacher. Guiding feature model evolution by lifting code-level dependencies. *Journal of Computer Languages*, 63, 2021.

[7] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering*, 33:725–743, 12 2007.

[8] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wąsowski. Variability-aware Performance Prediction: A Statistical Learning Approach. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 301–311, 2013.

[9] Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. Combining Multi-Objective Search and Constraint Solving for Configuring Large Software Product Lines. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 517–528, 2015.

[10] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, and Sven Apel. The Interplay of Sampling and Machine Learning for Software Performance Prediction. *IEEE Software*, 37(4):58–66, 2020.

[11] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. Distance-based sampling of software configuration spaces. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1084–1094, 2019.

[12] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990.

[13] Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

[14] Michael Nieke, Adrian Hoff, and Christoph Seidl. Automated Metamodel Augmentation for Seamless Model Evolution Tracking and Planning. In *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2019, Athens, Greece, October 21-22, 2019*, pages 68–80. ACM, 2019.

[15] Yusuf Sulistyo Nugroho, H. Hata, and K. Matsumoto. How Different Are Different Diff Algorithms in Git? *Empirical Software Engineering*, 25:790–23, 2019.

[16] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kastner. White-Box Analysis over Machine Learning: Modeling Performance of Configurable Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1072–1084, 05 2021.

[17] Max Weber, Sven Apel, and Norbert Siegmund. White-Box Performance-Influence Models: A Profiling and Learning Approach. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1059–1071, 05 2021.