TU Braunschweig

Master's Thesis

Stability of Product Sampling under Product-Line Evolution

Author:

Tobias Pett

08.11.2018

Advisors:

Prof. Ina Schaefer, Dr. Thomas Thüm, Tobias Runge

Department of Software Engineering and Automotive Informatics

https://doi.org/10.24355/dbbs.084-201812111412-0

https://doi.org/10.24355/dbbs.084-201812111412-0

Pett, Tobias:
Stability of Product Sampling under Product-Line Evolution
Master's Thesis, TU Braunschweig, 2018.

https://doi.org/10.24355/dbbs.084-201812111412-0

https://doi.org/10.24355/dbbs.084-201812111412-0

Abstract

Product-line sampling is a common method to cope with the exponential growth of products in product-line testing. Over the years, different sampling algorithms have been developed and validated against each other. Researchers strive to create efficient sampling algorithms to cope with large product lines. Typical criteria to evaluate sampling algorithms are the computation needed to calculate a sample, and the number of configurations a generated sample contains. Until now, no evaluation criteria considers the product-line evolution, as a factor for evaluating sampling algorithms. With this master thesis we present the stability of samples under product-line evolution as new evaluation criteria for sampling algorithms. Therefore, we define the meaning of stability in context of product-line evolution. Furthermore, we develop and implement metrics to measure the stability of sampling algorithms. Moreover, we classify whether established sampling algorithms produce stable samples or not, based on the results of our metrics.

https://doi.org/10.24355/dbbs.084-201812111412-0

https://doi.org/10.24355/dbbs.084-201812111412-0

Zusammenfassung

Die Erstellung von Beispielprodukten ist eine der am meisten genutzten Techniken, um mit der exponentiell wachsenden Anzahl an Produkten, beim Testen von Produktlinien, umzugehen. In den vergangenen Jahren wurden immer mehr und immer effizientere Algorithmen entwickelt, um Beispielprodukte für Produktlinien zu generieren. Typische Kriterien um diese Algorithmen zu bewerten sind, die Anzahl der erstelleten Konfigurationen und die benötigte Rechenzeit um diese Beispielprodukte zu erstellen. Bisher genutzte Kriterien beachten jedoch nicht die Stabilität der Algorithmen, wenn Beispielprodukte über die Evolution der Produktlinie erstellt werden. Im Rahmen dieser Master Thesis, definieren wir den Begriff der Stabilität im Bereich der Produktlinienevolution, als neues Bewertungskriterium. Zusätzlich, entwerfen und implementieren wir einige Metriken zum messen der Stabilität von Algorithmen zur Produkterstellung. Mithilfe unserer Metriken, messen wir die Stabilität verschiedener Algorithmen und klassifizieren sie als ein stabiles oder nicht stabiles Verfahren zur Erstellung von Beispielprodukten.

https://doi.org/10.24355/dbbs.084-201812111412-0

https://doi.org/10.24355/dbbs.084-201812111412-0

Contents

https://doi.org/10.24355/dbbs.084-201812111412-0

https://doi.org/10.24355/dbbs.084-201812111412-0

https://doi.org/10.24355/dbbs.084-201812111412-0

https://doi.org/10.24355/dbbs.084-201812111412-0

List of Figures

https://doi.org/10.24355/dbbs.084-201812111412-0

https://doi.org/10.24355/dbbs.084-201812111412-0

List of Tables

https://doi.org/10.24355/dbbs.084-201812111412-0

https://doi.org/10.24355/dbbs.084-201812111412-0

List of Code Listings

https://doi.org/10.24355/dbbs.084-201812111412-0

https://doi.org/10.24355/dbbs.084-201812111412-0

1. Introduction

A software product line is a collection of products, in a specific domain, which share a set core features and differ in a set of optional features [ABKS13]. A feature is a user visible behaviour. Relations between features are defined by a feature model. Products are generated by selecting a configuration of features. Core features are automatically selected and optional features can be manually selected to customize the product. Software product line engineering provides the means of developing highly-flexible and configurable systems for a specific domain. This helps developers to cope with the challenges of modern software development. Compared to single system development, software product line engineering decreases implementation costs, reduces time to market, and improves the quality of derived products [McG01, NC07]. Due to this reasons many companies switch to the software product line engineering process [Wei08].

Due to the increasing interest in software product line development, the reliability of variable products becomes more important. However, the vast amount of products, results often in a combinatorial explosion of test cases. Thus, an exhaustive test of a software product line is often infeasible. To cope with the combinatorial explosion of test cases combinatorial interaction testing is a promising approach [OMR10a, POS+12, PSK+10]. The combinatorial interaction testing is based on the assumption that most faults of software product line systems can be found in the interaction between just a few features [KAuR+09]. Over the past years different algorithms for software product line sampling such as Casa [GCD11], Chvatal [Chv79], ICPL [JHF11, JHF12] and IncLing [AHKT+16] were developed. Each algorithm produces a subset of all valid configurations of a product line, which are then tested.

Even though software product lines provide the means to meet differing customer needs, the complete future evolution of the product line can not be predicted in the design phase of the development. So, like any other software system, product lines change over time. To make sure changes do not introduce faults, retesting the software product line is inevitable. Regarding software product line testing with

https://doi.org/10.24355/dbbs.084-201812111412-0

sampling the retest procedure is to generate a completely new sample for every evolution step. Research on this area was not conducted until now [VAHT+18].

The research on combinatorial interaction testing under software product line evolution promises insights on, how the regression testing of software product lines can be improved. Especially knowledge about the stability of calculated samples can be used to improve testing quality and reduce test effort. For example, if it is known that a sampling algorithm generates stable samples, only a few new configurations will be contained in newly calculated samples. Hence, a previously calculated sample can be used to reduce the number of samples an algorithm needs to generate anew. Furthermore, a stable sample guarantees that configurations, which were previously fault free, will be tested again. Hence knowledge about testing stability can be used to flexibly adjust the test process.

1.1 Problem Statement

The aim of this master thesis is to examine the behaviour of different sampling algorithms in context of software product line evolution. Established sampling algorithms like Casa [GCD11], Chvatal [Chv79], ICPL [JHF11, JHF12] and IncLing [AHKT+16] will be examined and compared against a new sampling approach, which considers the software product line evolution history while sampling. The new sampling will be developed in context of the thesis. Among other criteria, the sample stability of the algorithms will be examined. Prior to the examination of sample stability, the meaning of sample stability will be defined. Furthermore, metrics to measure sample stability are developed. The aim of this master thesis leads to the following research questions.

Research Question 1: How stable are samples created by different sampling Algorithms?
To answer Research Question 1, we have to define a definition of sample stability. Based on this definition, we need to develop metrics for measuring sample stability. The metrics need to be applied to samples of different product lines. To generate those samples, we need to choose established sampling algorithms and develop a new sampling algorithm which aims to maximize the sample stability over the product-line evolution.

Research Question 2: How relevant are our self developed metrics, when applied to real world product lines?
For Research Question 2, we develop different metrics to measure sample stability and apply them to real world product line samples. Based on the results of the measuring, we will evaluate which metric can be reasonably used to measure the sample stability for industrial product lines.

Research Question 3: How does a self developed sampling algorithm, to maximize sample stability, perform compared to established sampling algorithms?
In Research Question 3, we evaluate the performance of our self developed sampling algorithm against the performance of established sampling algorithms. As comparison criteria we use Sampling efficiency, Testing efficiency and the sample stability of the algorithms.

https://doi.org/10.24355/dbbs.084-201812111412-0

1.2 Thesis Structure

We structured this master thesis as follows: Chapter 2 explains the the necessary background for this thesis. We introduce the basic of feature modelling and product-line engineering. In this context different established sampling procedures such as Casa, Chavatal, ICPl, and IncLing are presented. Furthermore, we present the concept of product-line evolution, by developing a running example for this master thesis. In Chapter 3 we introduce our definition of sample stability. In this context we present the concept behind our metrics for sample stability. Furthermore, we develop the concept of our own sampling algorithm. The aim of this algorithm is to maximize stability throughout the product-line evolution. Implementations for our developed concepts are described in Chapter 4. The chapter is divided in four main sections. First we explain the frame work on which our tool implementation is based. The other sections explain implementation details for our metric calculation, the stability oriented algorithm, and a tool to convert Linux variability models into a processable format. For the three tool implementations we describe their program structure and how they can be used. In Chapter 5 we evaluate the stability of different sampling algorithms by using our metrics. The evaluation is performed for different feature models such as Automotive2, Financial Services, and Linux. Additionally to accessing the sample stability of sampling algorithms, we estimate how our defined metrics work for the used feature models. Furthermore, we compare our self developed sampling algorithm against established sampling procedures. In Chapter 6, we present related work for this thesis. Afterwards, we summarize the achievements of our thesis, in Chapter 7. Finally, we describe topics, which could not be tackled in this master thesis, as future work in Chapter 8.

https://doi.org/10.24355/dbbs.084-201812111412-0

https://doi.org/10.24355/dbbs.084-201812111412-0

2. Background

In this chapter, we present background information for this master thesis. We start by introducing the term feature modelling in context of software product line engineering (Section 2.1). In Section 2.2, we present product sampling as a mean of generating a sample of products for a product line. We will regard different sampling algorithms. Section 2.3, contains information about the process of product-line evolution. In this context, we present the product-line evolution used as running example for this master thesis.

2.1 Feature Modelling

A software product line is a collection of products, in a specific domain, which share a set of common features and differ in a set of optional features [ABKS13]. To manage and express the variability of a software product line, a feature model is used [AHKT+16, TKB+14]. Furthermore, feature models are used to define valid feature combinations. A collection of selected features is also called a configuration for the feature model. Valid configurations represent the products, which can be generated by the product line. To check if a configuration is valid for the product line, a set of feature dependencies and constraints is defined by the feature model. A configuration is valid if none of those dependencies and constraints is violated. Typically feature models are visualized as feature diagrams, which express the hierarchical structure of feature models. An example feature diagram for our running example Graph Library is visualised in Figure 2.1.

Features contained in a feature model can be concrete or abstract [ABKS13]. The difference between both types of features is, that concrete features are mapped to implementation artefacts, while abstract features are not. In our example the features: *Graph Library*, *Wgt*, *Algorithms*, and *Edges* are abstract. All the other features contained in the example are concrete. Regardless of the feature type, each feature (parent) can have a of group subfeatures (children). The parent-child relationship between features claims that the parent must be selected if the child is selected. Moreover, feature models define that a child can be mandatory or optional.

https://doi.org/10.24355/dbbs.084-201812111412-0

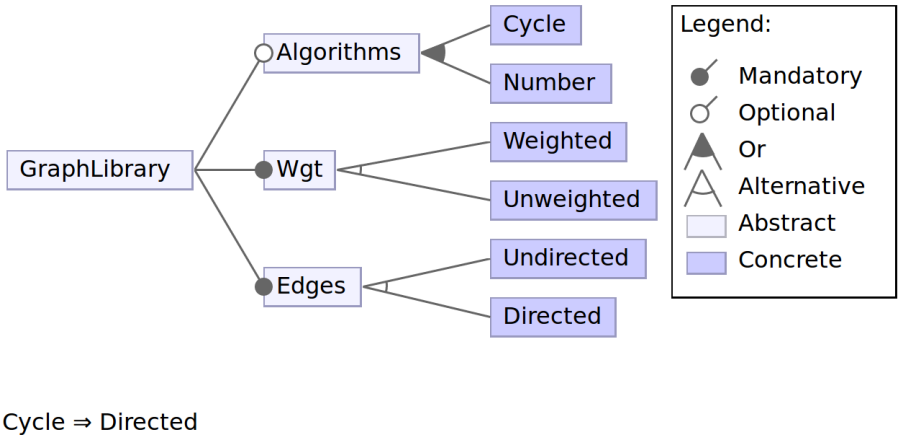

Figure 2.1: Example Feature Model from Graph Library Version 1

Mandatory features are required by the parent. Hence, they need to be selected, if the parent is selected. The feature *Edges*, shown in Figure 2.1, is an example for a mandatory feature, which means that each Graph Library needs edges. The features *Wgt* and *Algorithms* are optional features. As for the groups of subfeatures, and-, or-, and alternative-groups are defined. An and-group can contain an arbitrary number of mandatory and optional features. Or-groups define that at least one feature of the group must be selected. In Figure 2.1, *Cycle* and *Number* are contained in an or-group. Alternative-groups define that both features cannot be selected at the same time. In regard to our example feature model, this means that either the feature *Directed* or *Undirected* can be selected, but not both at the same time. Beside structural dependencies a feature model can include constraints as propositional formulas. Those are typically visualized under the feature diagram. In our example the constraint, that the feature *Directed* must be selected if the feature *Cycle* was selected, is defined for the Graph Library product line.

Based on structural dependencies and cross tree constraints, so called core and dead features can arise [BSRC10]. A core feature must always be selected to produce a valid configuration, while dead features must be deselected to produce valid configurations. If a feature is neither core nor dead, it is called a variant feature. A variant feature can become dead or core in a configuration, based on other feature choices. Such features are called conditionally core or conditionally dead. An example for a conditional core feature is given by the cross tree constraint of our example product line. If *Cycle* is selected, feature *Directed* becomes a core feature, which must be selected.

Since feature models can be used to check the validity of configurations, they are used as input to generate products [VAHT+18]. Those products are than used to test the correctness of the product line. The number of configurations, which can be produced for a product line increase exponentially with the number of features. Therefore, a complete test for product lines is often not feasible. Hence, product-line testing is, most of the time, executed on a sub set of all possible products. This sub set is called a product sample.

https://doi.org/10.24355/dbbs.084-201812111412-0

2.2 Product Sampling

As mentioned before, to test all products of a product line is not feasible for most product lines [AHKT+16]. Instead a subset of products, the so called product sample is used to test the product line. To generate those product samples, combinatorial interaction testing techniques can be used to create a subset of valid configurations for a product line.

The idea behind combinatorial interaction testing [LHFRE15] is to find erroneous behaviour of software products based on the interaction between features. In regard to feature interactions, different coverage criteria, such as feature-wise (1-wise), pair-wise (2-wise) or t-wise coverage can be defined [VAHT+18]. When using this t-wise combinatorial interaction testing, the defined feature interaction coverage must be achieved. For example, if pair-wise combinatorial interaction coverage is used, all valid pairs of features should be covered by configurations included in the sample.

Configurations found by t-wise combinatorial interaction testing, can be represented as so called covering arrays [AHKT+16]. To be precise, the whole process of generating configurations, which fulfil t-wise coverage, can be expressed as covering array problem [JHF11]. To generate covering arrays, different algorithms can be used. The following subsections present some of the established algorithms, to build product samples with t-wise coverage.

2.2.1 CASA

The Casa [GCD11] sampling algorithm is based on a simulating annealing algorithm. By using this evolutionary search strategy, covering arrays are created, which conform to t-wise coverage. The nature of simulated annealing, causes Casa to work non deterministic. That means, different results may be produced if the algorithm is executed multiple times for the same feature model.

2.2.2 Chvatal

The original Chvatal [Chv79, JHF11] algorithm is a greedy heuristic to generate a minimal covering array for a defined t-wise coverage. Originally the algorithm did not support feature dependencies. Therefore, it was not usable to generate samples. Johansen et. al [JHF11] improved the algorithm to incorporate feature dependencies. This way, the basic idea of the Chvatal algorithm can be used to produce samples, which fulfil t-wise coverage.

The algorithm creates configurations by incrementally adding feature combinations, from a previously generated list of all uncovered t-wise feature combination, to an empty configuration [JHF11, AHKT+16]. After a feature combination is added to the configuration, its validity is checked by a SAT solver. If the configuration is no longer valid the feature combination is removed. Thereafter the next feature combination is added to the configuration. This process repeats until the initial list of feature combinations is iterated through. If the created configuration contains at least one uncovered feature combination, it is added to the sample. Thereafter, the whole process starts again by creating a list of uncovered feature combinations.

https://doi.org/10.24355/dbbs.084-201812111412-0

2.2.3 ICPL

ICPL [JHF12] works essentially with the same greedy heuristic used in the Chvatal algorithm. However, Johansen et al. [JHF12] improve the performance of the original algorithm by parallelization and early elimination of invalid feature combinations. Their improvements make ICPL applicable for large software product line systems.

The parallelization Johansen et al. [JHF12] have implemented includes parting the core algorithm in different sub algorithms, which than can be run in parallel. Additionally to the parallelization, they use the idea that a covering array at time n-1 is a subset of (or the same set as) a covering array at time n, to quickly identify dead and core features for the feature model. Based on this information they eliminate invalid feature combinations early on in the sampling process.

2.2.4 IncLing

IncLing [AHKT+16] is an algorithm developed by Al-Hajjaji et al. [AHKT+16] to provide a efficient procedure for sample-based product-line testing. The IncLing algorithm works similar to the procedure of ICPL. Both algorithms generate new configurations for the sample, by sequentially selecting pairs of features not previously covered by configurations in the sample. However to increase efficiency IncLing provides the following modifications:

- **Incremental Approach:** To utilize testing time effectively IncLing generates configurations incrementally [AHKT+16]. This way, configurations can be tested and generated in parallel until the testing time is over. Furthermore, this approach provides the possibility to start the sampling procedure based on a previously calculated sample. If such a sample is provided, the algorithm avoids building already existing configurations by checking them against the provided set of previous configurations. To keep the set of previous generated configurations up to date, newly generated configurations are added to this set.

- **Detect Invalid Combinations:** Before starting the sampling process IncLing removes all invalid feature combinations, from the list of feature combination to search through [AHKT+16]. Removing invalid feature combination, reduces the search space for the algorithm. Hence, the calculation time needed for the sampling procedure is reduced.

- **Feature Ranking Heuristic:** To further improve the sampling efficiency IncLing uses a greedy strategy to cover the maximum number of still uncovered feature interactions, whenever a new configuration is generated [AHKT+16]. The heuristic tries to rank on the signum and frequency of a feature. The signum denotes to a value which increases by the number of how often a feature is selected and decreases by the number of how often a feature is deselected. Feature frequency represents the number of times a feature occurs in the remaining feature combinations. The greedy heuristic used in IncLing might produce more products than other sampling algorithms. However, it holds the potential advantage to cover many feature interactions as fast as possible.

https://doi.org/10.24355/dbbs.084-201812111412-0

- **Detecting Conditionally Dead or Core Features:** By using a satisfiability solver, IncLing individually tests each feature of the product line at hand, whether it can be selected or deselected in the current product [AHKT+16]. Doing so, conditionally dead or core features can be detected[BSRC10]. Feature pairs which contain conditionally dead or core features do not need to be considered, since they will be covered automatically. This way the search space for the algorithm is reduced, which increases the performance of the algorithm.

For our work the most interesting part about the IncLing algorithm is the incremental approach for generating sample sets. To be precise the possibility to provide an already existing sample as base for the calculation. This mechanism holds the possibility to provide an already existing sample from a previous product-line version, as base for the new calculation. This way, a possibly more stable sample can be produced.

For the sake of brevity we omit the implementation details of the IncLing algorithm. A detailed description of how IncLing works given by Al-Hajjaji et al. [AHKT+16].

2.3 Product-Line Evolution

The term product-line evolution describes, how the product line changes over time [BKL+15]. It is necessary to evolve the product line, to cope with changing requirements. As Alves et al. [AGM+06] describe, product-line evolutions should start by changing the feature model. Therefore, we focus on the feature model, as main artefact for product line evolution. To consider other aspects of evolution such as feature granularity [KAK08], feature to code dependencies, and code-level variability [LSB+10] is out of scope for this master thesis. Hence, whenever we speak of product-line evolution, we refer to changes in the feature models between two product-line versions.

Passos et al. [PGT+13] conducted an extensive analysis of change operations for the Linux kernel. As result they present, different change operations that occur in the Linux kernel and order them according to their rate of occurrence. Even though the analysis of Passos et al. [PGT+13] considers all three evolution spaces of Linux (Variability Model, Mapping, and Implementation), we can generalise their results to operations possible on feature models only. As general conclusion from the results of Passos et al. [PGT+13], we can extract, that features are more likely to be added (first) during the product-line evolution than removed (second). The third most frequent change operation is to rename features. Feature splits and merges appear the least frequent according to Passos et al. [PGT+13]

As a running example for this master thesis, we develop a small product-line evolution, based on the Graph Library example provided by FeatureIDE[1] [MTS+17a]. In FeatureIDE three different versions (tiny, small, medium) are included. We extend the tiny version of the example by adding two features, to create the first version for our example product-line evolution. From this version on, we use common change

https://doi.org/10.24355/dbbs.084-201812111412-0

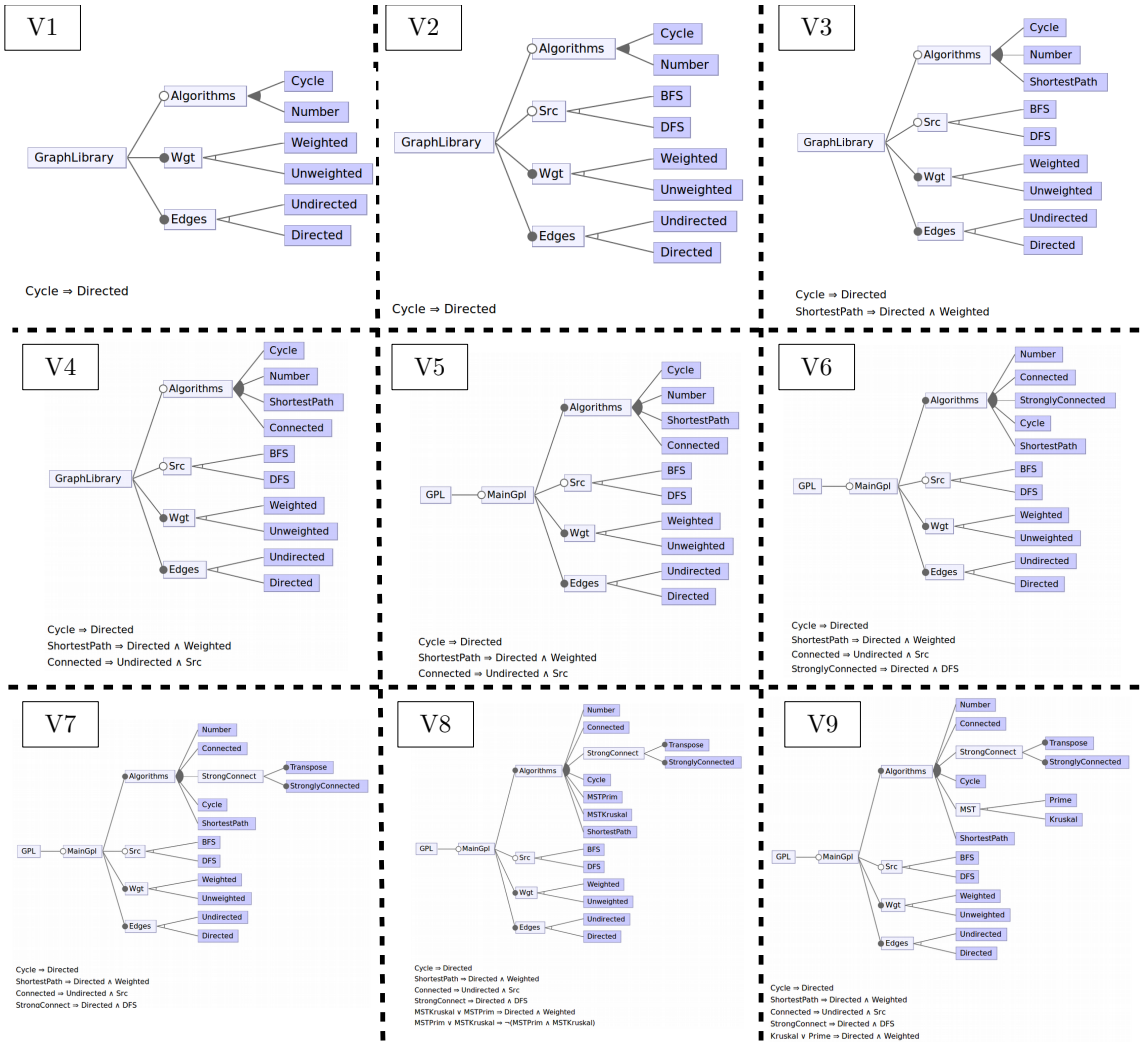

Figure 2.2: Evolution of the Feature Model of Graph Library

operations, in accordance to Passos et al. [PGT+13] to rebuild the evolution between the tiny and the small version of Graph Library.

In Figure 2.2, the evolution for Graph Library, we build as our running example and validation base, is shown. We start the evolution process with an extended version of the original Graph Library Tiny example from FeatureIDE. We extended the example by adding the mandatory abstract feature *Wgt* to the feature model. At the same time we introduce the features *Weighted* and *Unweighted* as alternative group under *Wgt*. In addition, we set the feature *GraphLibrary*, *Algorithms*, and *Edges* to abstract. In the first evolution step, we introduce the features *BFS* and *DFS* as alternative group under parent *Src* to the feature model. *Src* is an optional feature. Furthermore, it serves as a structural element, therefore it is from type abstract. The third evolution step introduces the new feature *ShortestPath* to the or-group under *Algorithms*. Moreover, we add a new constraint, which requires to select the feature *Directed* and *Weighted* if *ShortestPath* is selected. Version four, again adds a new feature (*Connected*) to the or-group under *Algorithms*. Further-

[1]https://github.com/FeatureIDE/FeatureIDE

https://doi.org/10.24355/dbbs.084-201812111412-0

more, a constraint which requires that *Undirected* and *Src* need to be selected if *Connected* is selected. From version four to version five, a re-structuring of the product line happens. First, the feature *GraphLibrary* is renamed to *GPL*. Second, a new abstract feature called *MainGpl* is added directly under *GPL*. The features previously under *GraphLibrary* are moved under *MainGpl*. The fifth evolution step, introduces a new algorithm to the product line, by adding the feature *StronglyConnected* and a corresponding constraint to the feature model. The added constraint requires that the features *Directed* and *DFS* need to be selected if *StronglyConnected* is selected. In the sixth step of the evolution history, a new abstract feature *StrongConnect* is added to the feature model as parent for the optional features *Transpose* and *StronglyConnected*. *Transpose* is newly added to the feature model, while *StronglyConnected* is moved from its previous position. In version eight, two new features (*MSTPrim* and *MSTKruskal*) as well as two corresponding constraints are introduced. The first new constraint states the following: If *MSTKruskal* or *MSTPrim* are selected than *Directed* and *Weighted* need to be selected as well. The second constraint ensures that *MSTKruskal* and *MSTPrim* are mutually exclusive ($MSTPrim \vee MSTKruskal \Rightarrow \neg(MSTPrim \wedge MSTKruskal)$). In the last evolution step, the constraint that *MSTKruskal* and *MSTPrim* are mutually exclusive is moved from textual description to a structural expression. To do so, a new abstract feature *MST* is added to the feature model. Furthermore the previous features *MSTKruskal* and *MSTPrim* are moved under *MST* as an alternative group. Additionally they are renamed to *Kruskal* and *Prim*.

During the course of this master thesis, we will refer to the first versions of the Graph Library evolution as examples to clarify our concepts. Moreover, the evolution history of Graph Library is used to validate our concepts and implementations. For this reason, we kept the product line evolution simple, but at the same time introduced typical change operations for product-line evolutions.

https://doi.org/10.24355/dbbs.084-201812111412-0

https://doi.org/10.24355/dbbs.084-201812111412-0

3. Stable Product Sampling for Product Lines

This chapter presents the conceptual ideas for analysing stability of product samples under product-line evolution. We define our understanding of sample stability in context of product-line evolution. In this regard, we describe requirements and assumptions we take for measuring stability. Additionally, we define metrics and describe how those metrics overcome certain challenges to conform to the requirements mentioned before. Furthermore we define a procedure to generate samples based the evolution history of product lines.

Sample stability can be described by how similar samples are to each other. Research on similarity between configurations is not new in the field of software product lines [HPP+14, Ham50, AHTM+14]. However, the analysis of sample similarity provides challenges, not occurring in the analysis of configuration similarity. To measure sample stability, we define the stability metrics *Ratio of Identical Configurations* and *Mean Similarity of Configurations*. Further we present a way to combine those metrics to create an even more refined metric for stability estimation. Beside the definition of stability metrics, we also develop conceptual ideas for a sampling algorithm which considers the evolution history of product lines. This algorithms is based on the idea that identical samples correspond to highest similarity between them. Therefore, we try to maximize stability by reusing as many configurations as possible from the previous samples.

This chapter is structured into four main sections. Section 3.1 presents a running example based on the Graph Library product line described in Section 2.3. This running example is used in other section of this chapter to describe how stability metrics work. Section 3.2 offers the initial definition of stability and defines requirements and assumptions for the metrics defined in Section 3.3 and Section 3.4. A possibility to build an even more refined metric, by combining the basic metrics is presented in Section 3.5. After we define how stability can be measured, a self developed concept of a sampling procedure is presented in Section 3.6.

https://doi.org/10.24355/dbbs.084-201812111412-0

3.1 Running Example

This section provides an example product-line evolution, from which example samples will be derived in the following sections. Those example samples will be used to clarify how metrics for sample stability work.

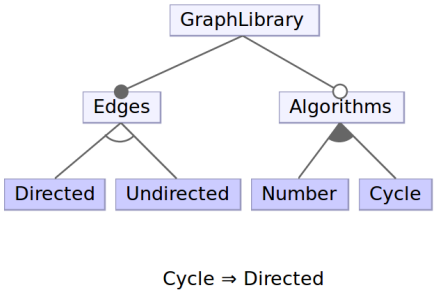

Figure 3.1: Example Feature Model for Graph Library Tiny

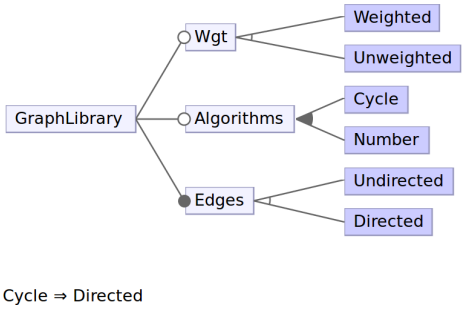

Figure 3.2: Example Feature Model for Graph Library Version 1

As examples we use product-line version based on the Graph Library evolution presented in Section 2.3. To keep our examples understandable we use the first version of our Graph Library evolution and a the Graph Library Tiny example from FeatureIDE. The feature models for those two are displayed in Figure 3.1 and Figure 3.2. From the presented feature models, feature sets listed in Equation 3.1 and Equation 3.2 were derived.

$$\begin{aligned} featuresetGPL_Tiny = \{&Directed(D), Undirected(U), \\ &Number(N), Cycle(C)\} \end{aligned} \tag{3.1}$$

$$\begin{aligned} featuresetGPLV1 = \{&Directed(D), Undirected(U), \\ &Number(N), Cycle(C) \\ &Weighted(W), Unweighted(UW)\} \end{aligned} \tag{3.2}$$

Equation 3.1 shows the feature set of Graph Library tiny. The features of Graph Library version 1 are visualised as feature set in Equation 3.2. Both feature sets contain only the concrete features of the feature models above.

Henceforth, we shorten the feature names of both Graph Library versions to the first letter, in cases where it is possible. In some cases the feature names are presented by the first two letters to prevent ambiguities.

During the reasoning about sample stability and the definition of stability metrics, example samples will be derived from the product lines visualized in Figure 3.1 and Figure 3.2. Those samples will be artificially constructed to full fill their purpose as visualisation example. Hence, they will not be derived by an sample algorithm. The derived samples will look as the examples is visualized by Table 3.1.

https://doi.org/10.24355/dbbs.084-201812111412-0

Table 3.1: Running Example: Example Samples Graph Library Version 1

Configuration	S	S'
C1	{D,N}	{D,N}
C2	{D}	{D,N}
C3	{D,C}	{D,N,W}

Table 3.1 shows two samples S and S'. Sample S is derived from Graph Library tiny to present the product line before evolution, sample S' is derived from Graph Library version 1 to represent the product line after the evolution step. For the sake of brevity we named the samples S and S' instead of Graph Library tiny and Graph Library version 1. During the description of our stability metrics, this naming will constantly be used, so that sample S always refers to a sample of Graph Library tiny and sample S' always refers to a sample of Graph Library version 1.

In the example from Table 3.1, each sample contains three configurations (C1, C2, C3). During the descriptions and visualisations we add to configurations from S' the $'$ modifier, to distinguish between configurations from S and S'. For the example this means, the three configurations from S' are called C1', C2' and C3'. This notation is used constantly during the following sections.

A configuration in our example is composed of selected features. Deselected features are not visualized explicitly but can be derived from the feature set of the product-line version. To derive deselected features the formula shown in Equation 3.3 can be used.

$$deselectedFeatures = featureSet \setminus configuration \tag{3.3}$$

Regarding the example displayed in Table 3.1 configuration C1 contains the features D and N. This means, for the product represented by C1 feature D and N are selected, all other configurations are deselected.

3.2 Sample-Stability Definition

Before we can reason about the stability of sampling algorithms, we need to define what sample stability in context of product-line evolution means. Therefore we analysed the meaning of stability in different contexts. The analysis results into definitions such as permanence, continuity and persistence. By examining stability definitions from other scientific areas particularly the definition of permanence stands out. For the stability definition in context of product-line evolution we take the meaning of permanence. This means a stable sample should change as little as possible during an evolution step. We can derive the following definition:

Definition 3.1. *Sampling stability in context of product-line evolution, describes how much a calculated sample changes over the product-line evolution.*

To examine how much a sample changes during an evolution step, we measure the similarity between samples before and after evolution. According to our definition

https://doi.org/10.24355/dbbs.084-201812111412-0

of stability different requirements and assumptions come up, which should be considered by measuring the similarity. For the rest of this section, we focus on the definition of those requirements and assumptions.

Requirement 1: Result is single value
We require for each metric to result into a single value. Later evaluation attempts will compare many results of those metrics. For this purpose, it is more advantageous to have each similarity result as a single value, instead of a list of values. If the metrics would result into a list of values, the following evaluation process would become confusing. So, if a metric yields a list of values as representation for sample similarity, this list need to be aggregated into one single value.

Requirement 2: Normalised values between zero and one
We require, that our metrics for measuring the similarity between two samples result in a normalised value between zero and one. Thereby, a value of one stands for full similarity and a value of zero stands for no similarity. Example 1 shows example cases where we expect our metrics to result into a value of one and zero.

Example 1.

Table 3.2: Identical Samples

S	S'
{D}	{D}

Table 3.3: Inverse Samples

S	S'
{D, C}	{U,N}

For this example, two sample pairs will be used. These pairs are displayed by Table 3.2 and Table 3.3. The samples contained in the sample pairs are artificially constructed from our running example presented in Section 3.1. Different from what is stated in Section 3.1, we derived the displayed samples S and S' from one and the same product line. To be precise from Graph Library version 1. Otherwise the purpose of this example can not be full filled.

Table 3.2 shows a pair of samples which contain one configuration each. The configurations select and deselect the same features of the product lines. Hence they are identical to each other. In such a case we expect our metrics to result into a value of one, which means full similarity.

The samples of the sample pair shown in Table 3.3 also contain one configuration each. This time the configurations are exactly inverse to each other. Which means the configuration of S selects exactly those features which are deselected in the configuration of S' and vice versa. In such a case, we expect our metrics to indicate no similarity. Hence, the metrics should result into a value of 0. □

Requirement 3: Consider selected and deselected features
We require from our metrics, that they consider the selected and deselected features during the process of calculating the similarity between samples. The need for considering both selected and deselected features is visualized in Example 2.

https://doi.org/10.24355/dbbs.084-201812111412-0

Example 2.
Imagine two product line versions with exact the same feature sets. Let the feature sets be relatively large (e.g. over 1000 features). Now we take one configuration for each product line version, where exactly one feature is selected, and calculate the similarity between them. If the selected features in both configurations are the same, a complete similarity will be calculated, no matter if only selected features are considered or if deselected features are considered as well.

However if the selected features in both configurations are different, considering only selected features, will result into complete dissimilarity. The majority of features contained in the feature sets will be ignored. Contrary, by taking selected as well as deselected features into account a high similarity score will be calculated. This reflects the fact, that both configurations contain allot of features they both deselect, which makes them similar. □

Even though the example on hand is an extreme case, it shows the problem of losing information through ignoring deselected features. Since we do not want to lose those similarity information we require, that our metrics consider selected as well as deselected features, by calculating the similarity on configuration level.

Requirement 4: Aggregation considers all values in the list
In Requirement 1 we stated, that the result of our metrics should be single values. However, some metrics result intermediately into a list of values. Hence, we need to aggregate such a list of values into a single value. Regarding the aggregation method we require, that it considers all values contained in the input list. The need for this requirement is described in Example 3.

Example 3.
For this example, we consider a stability metric which intermediately results into a list of similarity values. Such a list can look as those presented by Equation 3.4 and Equation 3.5.

$$simList1 = (0, 0, 1, 1, 1) \tag{3.4}$$

$$simList2 = (0, 0, 0, 1, 1) \tag{3.5}$$

We need to aggregate the intermediate list of values into a single value to conform to Requirement 1. This aggregation must reflect the similarity information contained in the set of values, as best as possible. If we base our aggregation on just one value and use it on the example list given in Equation 3.4 and Equation 3.5, the result will be biased. No matter which value of the provided lists we take, the result will be full similarity or no similarity. However, the entirety of values contained in the lists indicate a similarity in between the extreme points. □

Even though Example 3 displays a special case of intermediate results produced by a stability metric, it shows how information are lost by considering only one value of the similarity list and how the lost information bias the sample similarity in the end.

https://doi.org/10.24355/dbbs.084-201812111412-0

During the definition of the requirements above, we implicitly assumed some conditions for calculating the similarity between samples. Those assumptions will be explicitly described below. One thing assumed in all previous examples and visualisations of sample pairs, is that we see a sample as a set of configurations. The assumption is defined in Assumption 1.

Assumption 1: Sample is set of configurations
We assume, that a sample is a set of configurations. Configurations contained by a sample can be viewed as simple values, so that established set operations can be performed on samples. Due to the set character of samples, later defined metrics can not rely on the ordering of configurations contained in a sample. Furthermore, seeing a sample as set of configurations leads to the assumption, that no duplicate configurations are contained in the sample.

Another assumption implicitly assumed about measuring the stability, is that only two samples will be considered to calculate a similarity value. This assumption is explicitly defined in Assumption 2.

Assumption 2: Similarity is calculated between two samples
We assume, that the similarity metrics take only two samples as input to calculate the similarity between them. This way, the definition of how metrics work is easier to formulate and to understand. Contrary, it would be possible to define a metric which considers more than two samples to calculate a stability value. However, by calculating pairwise similarity for multiple sample pairs and aggregate the resulting sample stabilities, we can get equal results. Hence, we take the more simple approach of calculating pairwise similarity.

The third assumption we take considers the real structure of configurations. In Assumption 1 we defined, that configurations can be seen as simple value for the purpose of using set operations on sample pairs. Seeing configurations in this way limits the possibilities for more fine granular metrics. To enable a fine granular analysis, the real structure of configurations need to be considered. Hence, Assumption 3 defines how configurations can be seen in a more fine granular fashion.

Assumption 3: Configuration is a set of concrete and selected feature names
In Assumption 1 we defined, that samples can be seen as a set of configurations. In this context we defined configurations as simple values which are contained in the sample. In reality a configuration is a set of feature names. Due to the set character we can assume, that feature names contained in a configuration are not ordered. Furthermore, no duplicate feature names appear in a configuration. We need to consider this structure of configurations for later implementation of stability metrics. Furthermore, we can use the real structure of configurations as basis for more fine granular metrics. This assumption extends Assumption 1 through the statement, that a configuration can also be seen as a set of feature names, for the sake defining fine granular metrics.

https://doi.org/10.24355/dbbs.084-201812111412-0

3.3 Ratio of Identical Configurations (RoIC)

As the name suggests, *Ratio of Identical Configurations* is based on finding the identical configurations for a given pair of samples. However, how can identical configurations be identified? As stated in Assumption 1, a sample can be seen as a set of configurations. For the purpose of identifying identical configurations, a configuration will be seen as single value and not as a set of values as Assumption 3 suggests. Hence, we can use simple set operations to compare two samples. To identify identical configurations between a pair of samples, we use the intersection operation, as shown in Equation 3.6.

$$identicalConf(S, S') = |S \cap S'| \tag{3.6}$$

By using the intersection operation, a single number can be retrieved as similarity value. However, this similarity value is not normalized and conflicts with Requirement 2. The normalisation can be done by dividing the number of identical configurations by the total number of configurations contained in the samples. However, some restrictions need to be considered by building the divisor for normalisation.

For example, the divisor can not be zero, otherwise the normalization would not be defined for working with real numbers. Furthermore, the number of configurations in both samples need to be considered, because the samples could be of different sizes. Moreover, identical configurations in both samples should not be counted double. As methods for building the divisor, summing up all elements of both samples or building the union between the samples are possible. Example 4 explains how both methods work.

Example 4.

Table 3.4: Ratio of Identical Configurations: Influence of Default Values

Configuration	S	S'
C1	{D}	{D}
C2	{U}	{D,C,UW}
C3	{D,C}	{U}
C4		{D,C,W}
C5		{U,N,W}

For this example we use a sample before product-line evolution (S) and one after product-line evolution (S') provided in Table 3.4. Both samples are based on our example product line from Section 3.1. To fit the purpose of this example, we made sure to build one sample larger than the other and put identical configurations into the sample pair.

By looking at the sum method to build the divisor (divisor $= |S| + |S'| = 8$*) , we see that the sum calculates per definition a value larger than zero, as long as the size of one input sample is larger than zero. Furthermore, we can see that the*

https://doi.org/10.24355/dbbs.084-201812111412-0

size difference between both samples do not matter for the sum method. However duplicates of identical configurations, are introduced into the divisor, which makes it artificially higher than it should be. Because of this, the sum method is not applicable to build the divisor for normalising the number of identical configurations.

The second possibility to build the divisor is to use the set operation union on both samples. By doing so, (divisor $= |S \cup S'| = 6$*) a value larger than zero is calculated by definition, as long as one input sample contains more than zero elements. Moreover, size differences between samples do not matter for this operation. It always returns all elements contained in at least one sample. Beside this, duplicate elements will only count once. Hence, the union operation full fills the previously defined requirements for building the divisor for normalising the number of identical configurations.* □

As Example 4 shows, the union operation will be used as divisor for normalization. Therefore the calculation for the ratio of identical configurations follows an intersection over union. In literature this method is known as the jacard index [Jac12, TSK06]. This metric calculates the similarity between two sets of simple values, based on identical elements in the sets. The formula to calculate the jacard index as well as the result for the sample pair from Table 3.4 is given in Equation 3.7.

$$\begin{aligned} J(S, S') &= \frac{|S \cap S'|}{|S \cup S'|} \\ J(S, S') &= \frac{2}{6} \\ J(S, S') &= \frac{1}{3} \end{aligned} \tag{3.7}$$

Because the ratio of identical configurations is based on identifying identical configurations in a sample pair, it suffers from dead and core features, added or removed during the evolution step. Example 5 describes the challenge of added core features.

Example 5.
Imagine our running example is adjusted so that a new core feature Z is added during the last product-line evolution step. The following samples could be build for this imaginary product line.

Table 3.5: Ratio of Identical Configurations: Influence of Core Features

Configuration	S	S'
C1	{D}	{D,Z}
C2	{D,C}	{D,C,Z}
C3	{D,N}	{D,N,Z}
C4	{D,C,N}	{D,C,N,Z}
C5	{U}	{U,Z}

As shown in Table 3.5 feature Z, is contained in all configurations of S'. Considering the configurations of S, feature Z is not contained in any of the configurations.

https://doi.org/10.24355/dbbs.084-201812111412-0

Otherwise the samples are completely identical. However, by using Ratio of Identical Configurations *metric we get a similarity value of 0, which indicates no similarity.* □

The challenge described by Example 5 is the same for added dead features and removed core and dead features. Analogous examples could be described, but for the sake of brevity we refrain from doing so.

By removing core and dead features from both samples before comparing them, the described challenge can be solved. To do so, a short preprocessing is needed. During this process, the product-line versions of the samples need to be analysed for core and dead features. After identifying those features, they are removed from the configurations. This will generally lead to a higher similarity value, between samples.

However, does this artificial increase in similarity influence real application areas? As already described in Section 2.1 core feature need to be selected in every configuration, while dead features need to be deselected in every configuration of the product line. So, they are automatically covered if a valid configuration for the product line is build [BSRC10]. This means, a mismatch between compared samples and samples used in reality results from removing core and dead features during the similarity analysis. This mismatch, can be seen as error between real and estimated similarity of samples. However the error stays the same for all configurations, so that it does not influence the evaluation results.

Example 6 describes how the removal of core features can be used in case of the example samples in Table 3.5.

Example 6.
Imagine we take the sample pair S and S' from Table 3.5 and want to determine the ratio of sample reuse-ability by calculating their similarity. By removing all core and dead features from the sample pair, the ratio of identical configurations result to a full similarity. In context of reuse-ability this means, sample S can be used to analyse the new product-line version. However the new version requires the core feature Z to work. Because we know that this feature is a core feature for sample S', we can make sample S valid for the use in the new product-line version, by adding Z to all configurations of sample S.

Analogous to the described case, any sample of similar configurations found by Ratio of Identical Configurations *can be made valid for the new product-line version, by adding the necessary core features.*

For dead features the procedure differs a bit from the already described one. Instead of adding features we need to check if our sample for reuse contains feature which would be dead in the other product-line version. If so, we need to remove them, to make the sample valid again. □

Even though Example 6 shows, that the challenge of added and removed core and dead features can be solved, it also shows, that the *Ratio of Identical Configurations* metric is vulnerable against small differences in configurations. If configurations of a sample pair differ in one feature this metric does not consider them similar. The

https://doi.org/10.24355/dbbs.084-201812111412-0

reason behind this behaviour is the idea of only considering identical configurations. Even though, *Ratio of Identical Configurations* provides good results, if we are interested in reusing identical configurations of samples, sometimes we need a less restrictive metric. For example in the case of regression and performance testing it could be enough to know that similar but not identical configurations are contained in the sample pair. In such cases we still know, that the majority of previously tested features will be tested again, by using the new sample.

3.4 Mean Similarity of Configurations (MSoC)

As described in the previous section, considering identical configurations as similarity metric, can be to restrictive for some application areas. Hence, we developed another metric based on configuration similarity. This way we are able to consider the similarity information by all configurations and not only by the identical ones.

To calculate the sample stability, this metric determines the similarity between configuration pairs contained in both samples. The resulting configuration similarities are aggregated into a single value, which represents the sample stability between the samples on hand. Thereby Requirement 1 is full filled by the metric.

3.4.1 Similarity Measure

Based on Assumption 3, a configuration can be seen as a set of feature names. This way established metrics for calculating the similarity between sets can be used to calculate the similarity between configurations, further referred to as configuration similarity. Possible metrics are the Jacard index and the Hamming similarity.

Previously we used the Jacard index to calculate the ratio of identical configurations in Section 3.3. There we described how the Jacard index works and how it calculates the similarity between a pair of sets. Per definition the Jacard [Jac12] index only considers values contained in the sets. With regard to Assumption 3, a configuration contains feature names of concrete, selected features. The application of Jacard index to those sets results into a similarity value, based only on selected features. This behaviour conflicts with Requirement 3, which demands that selected as well as deselected configurations will be considered when measuring the similarity between configuration pairs.

The Hamming similarity was introduced in previous work of Al-Hajjaji et al. [AHTM+14] to calculate the similarity between configurations. They used configuration similarity to build an effective test case order. In their work Al-Hajjaji et al. [AHTM+14] state that the Hamming similarity considers the selected as well as deselected features, when calculating the configuration similarity. This behaviour conforms with Requirement 3. Hence, the Hamming similarity is applicable as similarity calculation for the *Mean Similarity of Configurations* metric.

$$d(C, C', F) = 1 - \frac{|C \cap C'| + |(F \setminus C) \cap (F \setminus C')|}{|F|} \tag{3.8}$$

https://doi.org/10.24355/dbbs.084-201812111412-0

By using the formula from Equation 3.8 Al-Hajjaji et al. [AHTM+14] calculate the difference between two configurations C and C' from the feature set F. By summing up the size of intersection between the features contained in the configurations ($|C \cap C'|$) and the size of the intersection between the features not contained in the configurations ($|(F \setminus C) \cap (F \setminus C')|$), Al-Hajjaji et al. [AHTM+14] consider selected as well as deselected features in their distance measure. The sum of intersection sizes is divided by the size of the feature set. This calculates a normalized similarity value between zero and one, where zero equals to no similarity and one equals to a full similarity. Hence, Al-Hajjaji et al. [AHTM+14] are interested in a value representing the difference and not similarity they subtract the similarity from one, to get the wanted result.

We can pick up the formula of Al-Hajjaji et al. [AHTM+14] but need to do a few adjustments to fit our purpose. As stated before Al-Hajjaji et al. [AHTM+14] are interested in the difference between configurations. Therefore, they use the $1-$ term in their formula. We are interested in the similarity between configuration, so that we can leave the $1-$ term out of the formula. The resulting formula looks like following:

$$sim(C, C', F) = \frac{|C \cap C'| + |(F \setminus C) \cap (F \setminus C')|}{|F|} \tag{3.9}$$

The following discussions will refer to the similarity value calculated by this formula as $sim()$.

3.4.2 Combining Feature Sets

The configurations used by Al-Hajjaji et al. [AHTM+14] originate from the same feature set. In our case the configurations originate from feature sets before and after a product line evolution step. As already mentioned in Section 3.1, we refer to the feature set before evolution as F and to the feature set after evolution as F'. Depending on the operations performed in the evolution step, F and F' could be equal or different from each other. To solve this challenge we need to build a combined feature set. Because a feature set is a set, we can use established set operations, such as intersection or union, to build the combined feature set.

By using the intersection features which are not contained in both configurations, will be removed from the combined feature set. So that, the remaining features were contained in both original configurations. This enables the similarity calculation to reach a similarity score of one, even if the feature sets F and F' differ significantly from each other. However applying the intersection operation to get the combined feature set, equals to a loss of information.

Example 7. Intersection leads to false statement

Imagine we take the product-line versions from our running example presented in Section 3.1 and construct one sample for the product line before evolution (S) and one sample for the product line after evolution (S'). Those samples are displayed

https://doi.org/10.24355/dbbs.084-201812111412-0

in Table 3.6. Both samples contain only one configuration. C1 and C1' differ only in the feature W. By looking at the feature sets of the running example we can see, feature W is only available in the product line after evolution.

Table 3.6: Mean Similarity of Configurations: Example Samples for Different Feature Sets

configuration	S	S'
C1	{D}	{D,W}

Now imagine we want to ascertain the degree of re-usability between those configurations, by calculating their hamming similarity. For the calculations we build the combined feature set by using the intersection operation. Hence, all features, which were contained in only one of the original feature sets are removed from the combined set. This way, the feature set F, displayed in Equation 3.10, is created. Because of the intersection operation, the features {W,UW} are removed completely from the similarity calculation.

$$
\begin{aligned}
F &= \{D, C, N, U\} \cap \{D, U, C, N, W, UW\} \\
&= \{D, C, N, U\} \\
\\
Sim(c1, c1', F) &= \frac{|\{D\} \cap \{D, W\}| + |(\{D, C, N, U\} \setminus \{D\}) \cap (\{D, C, N, U\} \setminus \{D, W\})|}{|\{D, C, N, U\}|} \\
&= \frac{1 + 3}{4} \\
&= 1
\end{aligned}
\tag{3.10}
$$

As shown by Equation 3.10, the hamming similarity between C1 and C1' results to one. According to our definition of full similarity from Requirement 2 this means, the configurations are identical. In terms of real world application areas identical configurations should produce the same results when analysed by performance or software tests. In case of our configurations C1 and C1', this does not have to be the case, because the high similarity score only results from lost information by building the combined feature set. To be precise C1 and C1' seem similar for the calculation, because feature W was removed from F due to the intersection operation. However C1' still contains the feature W, which can influence real world applications. Due to this influence, testing results for C1 and C1' can be different. In such a case, the statement of our similarity metric would be false, in context of real world applications. □

Example 7 shows that using the intersection operation for building the combined feature set, can lead to a false similarity estimation. Even though the example uses a constructed case, we can not allow that our metric estimates similarity values that do not reflect the reality. Hence, we can not use the intersection operation for combining feature sets.

https://doi.org/10.24355/dbbs.084-201812111412-0

If the union of original feature sets is build, the combined set contains all features contained in either one of the original sets. This way, we consider all features added or removed by the evolution step. Features not contained in the combined feature set, but not in either one of the original feature sets will be treated as deselected features for the corresponding product line.

By using a union between the original feature sets all features and information will be preserved. This way, even by comparing identical configurations, a similarity score of one is only possible if the configuration pair is identical. This is shown by Example 8.

Example 8.

This example presents how the union operation works, when used for building a combined feature set. We start by picking up the conditions used in Example 7. Imagine, we take the samples as presented in Table 3.6 and want to calculate the hamming similarity for the configurations C1 and C1'. However, instead of using the intersection to build the combined feature set we use the union operation. Equation 3.11 shows the resulting feature set F.

$$
\begin{aligned}
F &= \{D, C, N, U\} \cap \{D, U, C, N, W, UW\} \\
&= \{D, C, N, U, W, UW\}
\end{aligned}
$$

$$
\begin{aligned}
Sim(C1, C1', F) &= \frac{|\{D\} \cap \{D, W\}| + |(F \setminus \{D\}) \cap (F \setminus \{D, W\})|}{|F|} \\
&= \frac{1 + 4}{6} \\
&= \frac{5}{6} = 0.833
\end{aligned}
\tag{3.11}
$$

Now we simply build the hamming similarity as shown in Equation 3.11. As a result we get a similarity value of 0.833. This is still a high result, but it does not indicate that the configurations are identical. So, by using the union operation we preserve important information of selected features and can prevent wrong similarity statements in regard to real world applications.

Even though, the usage of the union operation for combining feature sets makes it more difficult to reach a similarity score of one, it is not impossible. To show this we constructed the samples S and S', shown in Equation 3.12, from the product lines of our running example.

Table 3.7: Mean Similarity of Configurations: Example Union for Identical Feature Sets

configuration	S	S'
C2	{D}	{D}

https://doi.org/10.24355/dbbs.084-201812111412-0

Both samples hold exactly one configuration C2 and C2'. This configurations are identical in regard to their selected features. The only difference between the samples S and S' is the feature set they result from. We calculate the hamming similarity between the configurations C2 and C2' as shown in Equation 3.12.

$$
\begin{aligned}
F &= \{D, C, N, U\} \cup \{D, U, C, N, W, UW\} \\
&= \{D, C, N, U, W, UW\}
\end{aligned}
$$

$$
\begin{aligned}
Sim(c2, c2', F) =& \frac{|\{D\} \cap \{D\}| + |(F \setminus \{D\}) \cap (F \setminus \{D\})|}{|F|} \\
=& \frac{1+5}{6} \\
=& 1
\end{aligned}
\tag{3.12}
$$

The calculation results into a similarity of one. Which indicates that both input configurations are identical. By looking at the configuration C2 and C2', we see this is true. Hence, in this case the union operation to combine feature sets reflects the relation between configurations realistically. If we take real world applications such as software or performance testing into account, the similarity value of one indicates that, C2 and C2' should produce the same testing results. By looking at the features selected by these two configurations that should be the case. Hence, the similarity estimation reflects the real world application areas under the condition that the feature implementation stayed the same during the product-line evolution. □

Example 8 shows, that by using the union operation for building a combined feature set, do not lead to wrong similarity estimations in regard to real world applications. This makes the union operation more applicable for the use in similarity measures. Hence, we will only use the union operation to build combined feature sets. Hence, if further examples and descriptions refer to a combined feature set, they refer to a feature set build by the union operation.

Table 3.8: Mean Similarity of Configurations: Example Samples

configuration	S	S'
c1	{D}	{D}
c2	{D,C}	{D,C}
c3	{D,N}	{D,N}
c4	{U,N}	{U,N}

Table 3.9: Mean Similarity of Configurations: Example Similarities

	c1'	c2'	c3'	c4'
c1	1	0.83	0.83	0.5
c2	0.83	1	0.67	0.33
c3	0.83	0.67	1	0.67
c4	0.5	0.33	0.67	1

The example samples used until now were, small artificial example constructed to visualize one specific case. One larger example sample pair for the running example product line, described in Section 3.1, is shown in Table 3.8. Table 3.9 shows similarity values calculated with hamming similarity and union feature set combination for all possible configuration combinations of the sample pair displayed in Table 3.8.

https://doi.org/10.24355/dbbs.084-201812111412-0

3.4.3 Aggregating Multiple Similarity Values

By using the previously described methods we can calculate a similarity value between any two configurations of the samples at hand. This way we get a list of similarity values, which represent the similarity between the samples. Henceforth, we will refer to this list of values as similarity list and use the shortened form *simList* for formulas and examples. What we are missing is a method to aggregate the individual values into one combined value, representing the similarity between two samples. Possible aggregations are: building the sum of all values, determining the minimum or maximum value of the similarity list, calculating the median between the values or calculating the arithmetic mean value. The following paragraphs discuss applicability of possible aggregation methods.

To visualize how the aggregation methods work the following similarity lists will be used as example basis.

$$\begin{aligned} simList1 =&(0.33, 0.33, 0.5, 0.5, 0.67, 0.67, 0.67, 0.67, \\ &0.83, 0.83, 0.83, 0.83, 1, 1, 1, 1) \end{aligned} \tag{3.13}$$

$$simList2 = (0, 0, 1, 1, 1) \tag{3.14}$$

$$simList3 = (0, 0, 0, 1, 1) \tag{3.15}$$

The similarity list from Equation 3.13 is another representation of the matrix shown in Table 3.9. It contains similarity values calculated with hamming similarity. The similarity lists from Equation 3.14 and Equation 3.15 are artificially created lists. They represent special cases of calculated similarity values.

The most simple aggregation method is to sum up the similarity values of the configuration pairs. According to our definition of similarity values in Section 3.4.1, no negative results for configuration similarity is possible.

$$\begin{aligned} sum(simList) &= \sum_{n=1}^{n=|simLst|} x_n \\ sum(simList1) &= 11.66 \\ sum(simList2) &= 3 \\ sum(simList3) &= 2 \end{aligned} \tag{3.16}$$

Hence, by summing up the single configuration similarities the sample similarity grows monotonically. This behaviour is shown in Equation 3.16. It is visible that, non of the calculated sample similarities lies between zero and one. Hence, for all example cases, the sum aggregation conflicts with the Requirement 2. Hence, we can not use the sum aggregation to aggregate the similarity lists.

Beside calculating the sum of values, determining the minimum or maximum value is another possible aggregation method. The minimum determines the lowest value

https://doi.org/10.24355/dbbs.084-201812111412-0

in the similarity list, while the maximum determines the highest value in the list. Except that both procedures deliver the exact inverse result of each other, they work the same. Hence, we will only reason about the minimum aggregation. Reasoning about maximum aggregation is analogous.

$$\begin{aligned} min(simlist1) &= 0.33 \\ min(simList2) &= 0 \\ min(simList3) &= 0 \end{aligned} \tag{3.17}$$

The example calculation in Equation 3.17 shows that final sample similarity score calculated by the minimum aggregation is based on one single value. This conflicts with Requirement 4. Hence, the minimum aggregation is not applicable for aggregating similarity sets. The same reasoning holds for the maximum aggregation.

Another aggregation method would be, to calculate the median of all similarity values. To calculate the median, the list of similarity values must be ordered according to the similarity score. From this ordered list the middle is taken as the median. Should the list contain an even number of elements, two middle values can be derived. Both values are added and then divided by two, to calculate the median.

$$\begin{aligned} median(simlist1) &= 0.75 \\ median(simList2) &= 1 \\ median(simList3) &= 0 \end{aligned} \tag{3.18}$$

As Equation 3.18 visualizes the mean aggregation produces normalised configuration similarities which conform to the Requirement 2. However, the aggregated sample similarity is based on only one or two values of a larger list of similarity values. So, it conflicts with Requirement 4.

The next aggregation method to be analysed is the calculation of the arithmetic mean value. The single similarity scores are summed up as shown in the sum aggregation method. But the resulting value is qualified by the number of values involved in the summation, as shown in Equation 3.19

$$Mean(SimList) = \frac{1}{|SimList|} \times \sum_{i=1}^{|SimList|} sim_i \tag{3.19}$$

$$\begin{aligned} mean(simlist1) &= 0.73 \\ mean(simList2) &= 0.6 \\ mean(simList3) &= 0.4 \end{aligned} \tag{3.20}$$

The example shown in Equation 3.20 visualizes the results of the arithmetic mean for the previously presented similarity list examples. The results conform to Requirement 2. Furthermore it conforms to the Requirement 4 in respect, that all elements of the similarity list are considered during the aggregation process. This makes the arithmetic mean value the most applicable aggregation method for the *Mean Similarity of Configurations* metric.

https://doi.org/10.24355/dbbs.084-201812111412-0

3.4.4 Configuration Matching

As Assumption 1 indicates, a sample can be seen as set of Configurations. This means, the comparison between two samples can be seen as a comparison of two disjoint sets of values.

As previously discussed in Equation 3.9, we can build the similarity between two configurations by calculating the hamming similarity. By doing so, a mapping between those configurations is established. The similarity can be seen as a connecting edge between configurations. Due to this, building the similarity of configurations can be seen as a matching problem in a bipartite graph. An example is shown in Figure 3.3.

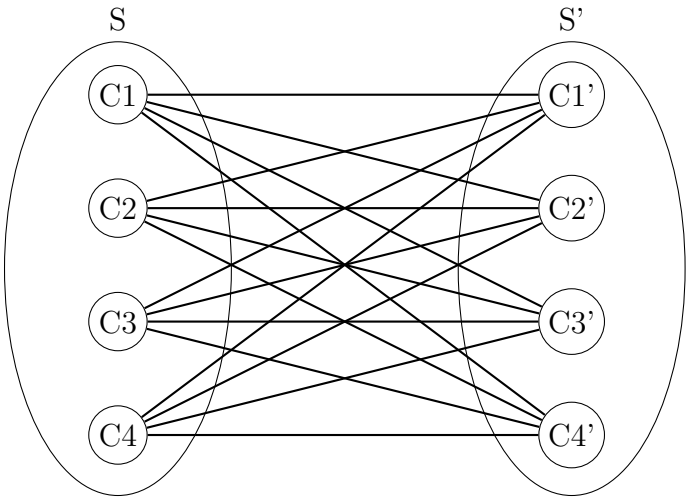

Figure 3.3: Mean Similarity of Configurations Complete Matching

Figure 3.3 visualises a matching between the example samples from Table 3.8. In this example, Sample S is used as mapping source and sample S' is used as mapping pool. That means each configuration from S is matched to a configuration of S'. The visualised matching is called a complete match between those sets, because it realises all the possible matches between all configurations. An edge between the matched configurations represents the similarity between them. So, after the matching edges can be grouped together as a list of similarity values. As stated in Section 3.4.3, such a list needs to be aggregated before it can be used as representation for the sample similarity.

The sample similarity, depends on how the matching between configurations is done. Hence, we face the challenge of choosing a suitable matching. Aggregating over a complete match between samples, does not hold much information gain. So we need to define a similarity criteria to optimize our aggregated value in the direction of maximum similarity or maximum dissimilarity. By taking only the most similar configurations into the similarity set, the maximal similarity between two samples will be calculated (maximum matching). Contrary, taking only the most dissimilar configurations into the similarity set, the minimal similarity between the samples is calculated (minimum matching).

In regard of simply measuring stability between samples as an abstract concept, it does not matter if maximum or minimum matching is used. Both variants produce

https://doi.org/10.24355/dbbs.084-201812111412-0

results which can be interpreted in the right way, as long as it is known which technique was used. However, by considering practical applications, for example the reuse of configurations (products) in a performance test, the maximum matching should be used.

Example 9.
Imagine we want to compare the performance of two product-line versions by using generated samples. To gain a significant result, it is important that the configurations (products) of the sample before evolution (S) and the sample after evolution (S') are as similar as possible. To reflect this with our similarity measure, we need to use maximum matching. If we would use minimum matching, the most dissimilar configuration will be matched together. This behaviour does not reflect the need of the application area performance test. □

Example 9 describes that the minimum matching is not applicable for the application area of performance testing. This application area is one of the main area the stability metrics will be used for. Hence, instead of minimum matching the maximum alternative will be used for *Mean Similarity of Configurations*. The following examples and discussions about the matching for *Mean Similarity of Configurations* will assume the maximum matching as chosen similarity criteria. The Examples and discussions for minimum matching are analogous, except that the criteria for best match is switched to minimal similarity. Because both procedures are so similar and the preferable matching process is maximum matching, we refrain from presenting and describing matching for minimal similarity.

Algorithm 1. Basic Greedy Matching

```
1  function: greedyMatch(S,S',F)
2  S ← sample before evolution
3  S' ← sample after evolution
4  F ← combined feature model S and S'
5  SimilarityList ← List of best matched edges
6  for i ← 1 to |S| step 1 do
7    edgeList ← List of edge objects
8    for j ← 1 to |S'| step 1 do
9      sim ← HamSim(S[i],S'[j],F)
10     edge ← Edge(S[i],S'[j],sim)
11     edgeList add(edge)
12   end for
13   bestMatch ← get edge with maximal similarity from edgeList
14   similarityList add(bestMatch)
15   drawEdge(bestMatch)
16 end for
17 end function
```

Listing 3.1: Mean Similarity of Configurations: Basic Greedy Matching

Figure 3.4 shows a matching example where similarity criteria is set to find configuration pairs which are most similar to each other. The algorithm to produce such a matching result is given in Algorithm 1

https://doi.org/10.24355/dbbs.084-201812111412-0

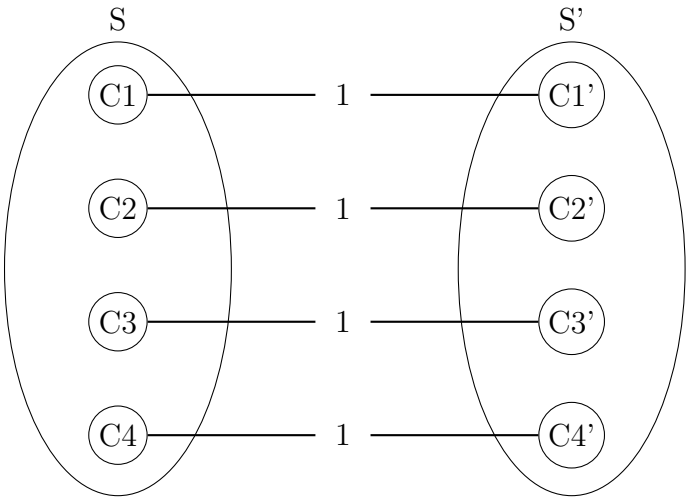

Figure 3.4: Mean Similarity of Configurations: Controlled Matching

The algorithm above takes two samples S and S' as origin. S represents a sample from a product-line version before evolution, S' represents a sample from a product-line version after evolution. Furthermore a combined feature set F for both versions is given. The configurations of S are iterated through (Line 6). For each configuration all possible similarity values to the configurations of S' are calculated and stored as edges into a list (Line 8-12). After all possible combination for the current configuration of S are build, the edge with the maximum similarity is filtered out (Line 13) and taken as the best match possible for the current configuration of S. Regarding the filter process one special case can occur, if two edges hold the same similarity value. In such a case the first edge found by the filter function is returned. The resulting edge is stored in a similarity list (Line 14). For the graph representation the process of storing the edge into the list of similarities means the edge is drawn between the configurations contained in the edge object (Line 15). The algorithm runs until all configurations in S have a matching partner in S'.

With the algorithm presented in Algorithm 1, a basic procedure to match configurations of two samples is presented. By using this algorithm three different cases concerning the size difference between samples need to be regarded. Those cases appear under the conditions $S = S'$, $S > S'$ and $S < S'$.

In the first case $S = S'$, Algorithm 1 matches all configurations in S to a configuration in S', such as described before. Thereby Algorithm 1 does not restrict the number of matching partners a configuration in S' could have. Due to the missing restrictions, the algorithm could match two or more configurations of S to one configuration of S'. Furthermore, Algorithm 1 does not require that every configuration of S' gets a matching partner. This way, unmatched configurations in S' can appear.

For the second ($S > S'$) and third ($S < S'$) condition, the behaviour of Algorithm 1 is fairly the same. The algorithm terminates if all configurations of S are matched to a configuration of S', there by a configuration of S'can have multiple partner in S. Furthermore, not all configurations in S' need to be matched to a partner in S.

Example samples for the conditions $S > S'$ and $S < S'$ are given in Example 10.

https://doi.org/10.24355/dbbs.084-201812111412-0

Example 10.
Imagine, we extract two new sample pairs from the product-line evolution of our running example. One where the condition $|S| > |S'|$ holds and the other where condition $|S| < |S'|$ holds. As example sample pairs Table 3.10 represents the condition $|S| > |S'|$ while Table 3.11 presents condition $|S| < |S'|$. The sample pairs displayed below are artificially constructed to serve the purpose of visualizing the challenges on hand. Beside each sample, a matrix of similarity values is displayed. The matrices contain similarity values for all possible matches between the given configurations calculated with the hamming similarity defined in Section 3.4.1.

Table 3.10: Mean Similarity of Configurations: Example Samples Sample > Sample'

configuration	S	S'
C1	{D,C }	{D }
C2	{U,N }	{U.N }
C3	{ U,N,W}	{ D,C}
C4	{ D,N,UW}	

	C1'	C2'	C3'
C1	0.83	0.33	1
c2	0.5	1	0.33
C3	0.33	0.83	0.17
C4	0.67	0.5	0.5

Table 3.11: Mean Similarity of Configurations: Example Samples Sample < Sample'

configuration	S	S'
C1	{D }	{D,C }
C2	{U,N }	{U.N }
C3	{ D,C}	{ U,N,W}
C4		{D,N,UW }

	C1'	C2'	C3'	C4'
C1	0.83	0.5	0.33	0.67
C2	0.33	1	0.83	0.5
C3	1	0.33	0.17	0.5

In both visualized conditions we face the challenge, that one sample is larger than the other. So that we can not match every configuration of the larger sample to a configuration of the smaller sample, without using a configuration of the smaller sample twice. □

As described before, by using Algorithm 1 unmatched configurations as well as duplicate matchings can appear. Until now we did not describe how the metric handles those occurrences. We present two different procedures to handle the challenge of duplicate matchings and unmatched configurations in Section 3.4.5 and Section 3.4.6.

3.4.5 1:1 Matching

We start by analysing the 1:1 matching. This procedure restricts the number of partners a configuration can match with to one. Matching two configurations of a sample pair follows the algorithm shown in Algorithm 1, with a simple adjustment. After a matching pair of configurations is found, the corresponding configurations are removed from the sample sets. The matching is repeated until every configuration from S has a partner or all configurations from S' are removed, depending on which condition occurs first. This procedure equals to a typical greedy approach, to find best matches. For the example sample pairs of Table 3.10 and Table 3.11 this matching was conducted. The results are visualized in Figure 3.5 and Figure 3.6.

https://doi.org/10.24355/dbbs.084-201812111412-0

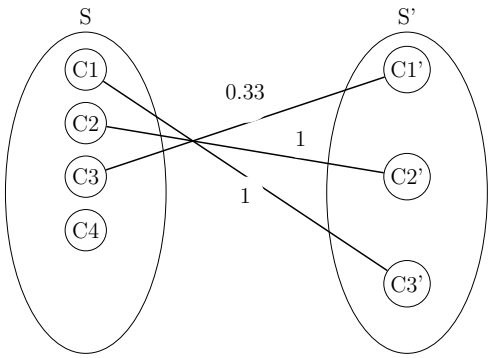

Figure 3.5: Mean Similarity of Configurations: 1:1 Matching for Sample > Sample'

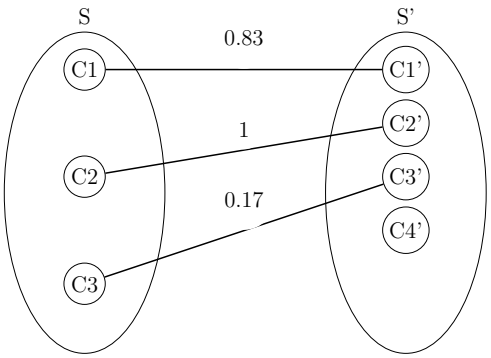

Figure 3.6: Mean Similarity of Configurations: 1:1 Matching for Sample < Sample'

In both cases we can clearly see that some configurations of the larger sample remain unmatched. To handle this challenge of unmatched configurations two possible solutions exists. First, any configuration without a matching partner could be ignored completely, so that it will not contribute to the aggregated similarity. This approach equals to a loss of information. Due to the exclusion of unmatched configurations it is theoretically possible to reach a very high similarity value, even though the samples differ significantly in size. Contrary to ignoring configurations, a default similarity value (e.g. 0) can be assigned to unmatched configurations. This way, unmatched configurations are not completely ignored.

By using a default value for unmatched configurations, the aggregated similarity can be influenced. Two cases to mark the border of this influence are a default value of zero and a default value of one. Table 3.12 shows the aggregated similarity values of the matching visualised in Figure 3.6. Sample similarities for the cases of a default value 0 and 1 are shown, as well as the sample similarity in case unmatched configurations are ignored completely. For the sake of brevity aggregated similarities for the matching shown in Figure 3.5 are omitted.

Table 3.12: Mean Similarity of Configurations: Influence of Default Values

Default	SimList	Mean(SimList)
0	(0.83, 1, 0.17,0)	0.5
ignore	(0.83, 1, 0.17)	0.67
1	(0.83, 1, 0.17,1)	0.75

Previously we described that ignoring unmatched configurations equals to artificially constructing samples of the same size. Hence, for the following comparison the case of ignored unmatched similarities can be used as comparison base. The example from Table 3.12 shows that the aggregated similarity in case of a default value of zero is lower than the similarity for ignored configurations. With a default value of one the aggregated similarity is higher than in the case of ignoring unmatched configurations completely.

Even though the example shown in Table 3.12 is based on one configuration matching, it shows that a default value of one artificially increases the similarity of a sample

https://doi.org/10.24355/dbbs.084-201812111412-0

pair. This contradicts with the intuitive expectation that samples of different sizes should be less similar to each other. Hence, the default value of one, should not be used. An similar argument holds for ignoring unmatched configurations. This procedure in handling the size difference between samples, ignores it completely. Doing so can lead to identical configuration, even though the configurations differ in size. This behaviour again conflicts with the intuitive view of samples with different sizes. The result of a decreasing similarity value, if unmatched configurations exist conforms to the intuitive sense of a lower similarity value if samples differ in size. Hence, for further analysis a default value of zero is taken for unmatched configurations.

Another challenge the 1:1 matching must face, is based on the greedy matching strategy initially described. As described by Cormen in [CLRS09] a known problem of greedy strategies is, that due to the removal of already matched partners possible better matchings are prevented. Hence, greedy algorithms are vulnerable to find a local optima, instead the global one.

However, the 1:1 matching prevents double mapping between configurations by default. This way found pairs conform more to the reality, as each configuration only matches to one other. Another advantage of the 1:1 matching concerns the possibility of unmatched configurations. By weighting these configurations with a default value of zero, the size difference is mirrored more prominently, than by calculating the similarity to another configuration.

3.4.6 N:M Matching

As already seen the 1:1 matching procedure presented previously handles the challenge of different sample sizes by introducing a default value for unmatched configurations. With the N:M matching procedure we present another way to handle size differences between sample pairs. Similar to the 1:1 matching, N:M matching uses the basic greedy Algorithm 1. However, instead of restricting the number of possible matches for a configuration, we allow an arbitrary number of matches for a configuration of either sample. Furthermore we require for each configuration from both samples to find at least one matching partner in the other sample. This requirement solves the challenge of size differences between sample pairs. We will analyse how the N:M matching handles different sample sizes by regarding the artificially created similarity matrix in Table 3.13.

Table 3.13: Mean Similarity of Configurations: Example Similarities N:M Matching

	C1'	C2'	C3'
C1	0.83	0.33	1
C2	0.5	1	0.33
C3	0.33	0.83	0.17
C4	0.33	0.5	0.83

Example 11.
Because the N:M matching puts no restrictions on the number of matches a configuration can hold, the greedy algorithm presented in Algorithm 1 runs until all

https://doi.org/10.24355/dbbs.084-201812111412-0

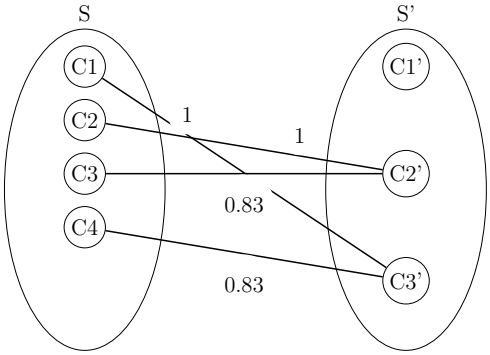

Figure 3.7: Mean Similarity of Configurations: N:M Matching for Sample > Sample'; Sample' unmatched

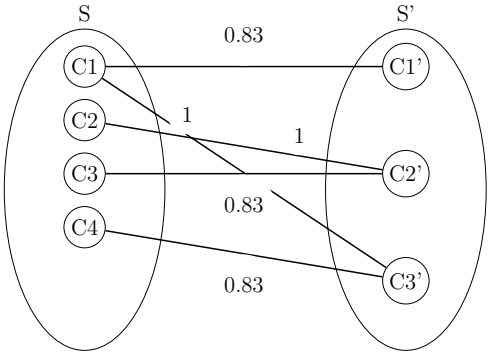

Figure 3.8: Mean Similarity of Configurations: N:M Matching for Sample > Sample'; all matched

configurations from sample S have a matching partner. This way the challenge of size difference under the condition $S > S'$ is solved by default. However, because the number of matching partners is not restricted, it could happen that two or more configurations of S are mapped to one configuration of S'. This behaviour is shown in Figure 3.7. As shown by this example, this behaviour leads to unmatched configurations in sample S'. By applying algorithm Table 3.12, this challenge can be handled, so that every configuration has at least one matching partner. The final matching for this example is shown in Figure 3.8. □

Algorithm 2. N:M Post Processing

```
1  function: matchUnmatched(S, S', SimilarityList)
2  S ← sample before evolution
3  S' ← sample after evolution
4  SimilarityList ← List of best matched edges, calculated by greedyMatching
5  unmatchedConfs ← getUnmatched(S', SimilarityList)
6  intermediateSimList ← greedyMatching(unmathcedConfs,S)
7  SimilarityList ← SimilarityList merge intermediateSimList
```

Listing 3.2: Mean Similarity of Configurations: N:M Post Processing

Algorithm Algorithm 2 shows how the post processing, to find matches for unmatched edges, works. This algorithm requires two input samples S and S'. Further a list of already matched configurations is required. As already described unmatched configurations in sample S are not possible for the N:M matching because of the greedy algorithm displayed in Algorithm 1. So, only unmatched configurations in sample S' need to be found. After determining these unmatched configurations, they are stored in a configuration set. To match the unmatched configurations greedyMatching() from Algorithm 1 is used. This algorithm takes the unmatched configurations from S' and the sample set S as input. This way, an intermediate similarity list, containing matches for the previously unmatched configurations is generated. This list of similarities is merged with the similarity list previously found by the greedyMatching(S,S'), to create a final list of similarities. This way the challenge of unmatched configurations in sample S' is solved. Hence, using the post processing

https://doi.org/10.24355/dbbs.084-201812111412-0

of Algorithm 2 the challenge of unmatched configurations under all conditions is solved.

The advantage of N:M matching over 1:1 matching is, that no unmatched configuration occur if the samples of a sample pair are of different sizes. Due to this, no artificial default values need to be introduced into the final set of similarities.

However, allowing an arbitrary number of matching partners for each configuration can lead to a degenerated mapping between sample pairs. Such a mapping occurs only in special cases of sample pairings. A similarity matrix to visualise such degenerated mapping is presented in Table 3.14.

Table 3.14: Mean Similarity of Configurations: Example Samples Degenerated Matching

	c1'	c2'	c3'
c1	0.83	0.83	0.83
c2	0.83	0.17	0.5
c3	0.83	0.5	0.67
c4	0.83	0.5	0.17

Example 12.
The example of Table 3.14 shows an artificially constructed similarity matrix based on the running example from Section 3.1. Prominently to see is, that the first row and column of the matrix contain the highest similarity values of the whole matrix. This leads to the matching shown in Figure 3.9. We see all configurations from S map only to the configuration C1' in sample S'. Similar, all configurations of sample S' map to configuration C1 of sample S. This degenerated mapping is not shown in the final similarity value, due to the aggregation. Hence the final similarity value can be very high, just because one configuration of S or S' is similar to all other configuration of the other set.

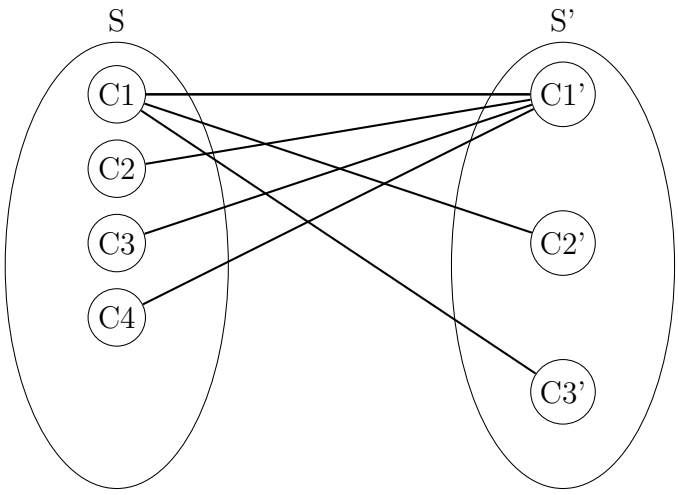

Figure 3.9: Mean Similarity of Configurations: N:M Degenerated Matching

If we consider the application area of reusing samples for performance testing, such a degenerated mapping influences the application negatively. For this area it is im-

https://doi.org/10.24355/dbbs.084-201812111412-0

portant that the samples stay mostly the same, which should be displayed by the similarity. However a high similarity score based on the mapping described above, does only mirror that one configuration of each sample is highly similar to all configurations of the other sample. □

Example 12 shows that the N:M mapping suffers under the challenge of degenerated mappings. If such a case occurs, the similarity estimation is biased in regard to real world application scenarios. Furthermore, by using N:M mapping as in a similarity metric, can hide possible size differences between samples. This behaviour results from mapping all configurations from one sample to at least one configuration from the other sample. Even though the N:M mapping can have possible applications in measuring similarity. However the more preferable mapping approach is the 1:1 mapping with a value of zero as default for unmatched configurations.

3.5 Filter Identical and Match Different Configurations

Section 3.3 and Section 3.4 describe two different metrics and there variations. Both metrics basically differ in their strategy to measure similarity. The *Ratio of Identical Configurations* metric searches for identical configurations in two different samples, while the *Mean Similarity of Configurations* metric works on a more fine granular level by measuring configuration similarities. Even though both metrics use different strategies to generate a similarity estimation, they take the same basic structure as input. The idea now is to combine the basic ideas of those metrics into one metric. By doing so, drawbacks of each idea will be mitigated by the benefits of the respectively other one.

The idea behind a combined use of both metrics is to filter out identical configurations before measuring the configuration similarities. This way, the estimation of *Mean Similarity of Configurations* is qualified in regard that only similar and not identical configurations are analysed.

In regard to this idea, the *Ratio of Identical Configurations* metric is not needed in its full extent. Only the functionality to filter out identical configurations is needed. As already seen in Example 5, we face the challenge, that added and removed core and dead features lead to almost identical configuration (difference is one core or dead feature), which are not recognize as such. To prevent such cases, we filter core and dead features from both samples out, as described in Example 6.

https://doi.org/10.24355/dbbs.084-201812111412-0

As described in Section 3.4 the *Mean Similarity of Configurations* has different variations that could be used. However, we already explained which variations are the favourable in context of similarity measure for practical application areas. To sum up our previous argumentation, following list presents variation decisions, which will be used for further analysis.

- Similarity Measure: Hamming Similarity
- Combining Feature Sets: Union Set Operation
- Aggregation Method: Arithmetic Mean
- Similarity Criteria: Maximum Similarity
- Matching Procedure: 1:1 Matching
- Default for Unmatched: 0

Combining the filtering of identical configuration with a similarity analysis on configuration level, can be a promising approach to calculate a meaningful similarity value for a pair of samples. How we combined those approaches is shown in Algorithm 3.

Algorithm 3.

```
1  function FIMDC(F,S,F',S')
2    identConf ← S ∩ S'
3    I ← S \ identConf
4    I' ← S' \ identConf
5    SimList ← 1:1Matching(F,F',I,I')
6    for(|identConf|)
7      SimList add 1
8    end for
9    StabValue ← aggregate(SimList)
10 end function
```

Listing 3.3: Filter Identical Match Different Configurations: Procedure

The function *Filter Identical Match Different Configurations* combines basic ideas of *Ratio of Identical Configurations* and *Mean Similarity of Configurations*. As input the feature model before (F) and after (F') the evolution step is taken into account as well as a sample for each product-line version (S, S'). At first, the algorithm identifies identical configurations, by using the intersection operation on S and S' (Line 2). The calculated set is stored. Secondly intermediate samples are build, by removing the identical configurations from the respective samples (Line 3 and 4). To do so, the difference set operation is used. After the creation of two disjunct sets of configurations the configuration similarity is analysed. This is done by using the 1:1 matching with a default value of zero, described in Section 3.4.5. As input the feature models F and F' as well as the respective intermediate samples I and I' are used. The procedure results into a list of similarity values called simList (Line 5).

https://doi.org/10.24355/dbbs.084-201812111412-0

After the similarity list is created, the number of identical configurations need to be integrated into this list. Therefore, the meaning of a identical configuration in context of similarity need to be considered. As defined in Requirement 2, an identical configuration equals to a similarity value of one. So, for each identical configuration found by the intersection a similarity of one is added to the similarity list (Line 6 to 8). Finally, the values contained in the similarity list are aggregated (Line 9). The aggregation is done by building the arithmetic mean over all values contained in the similarity list. Thereby a representation of the similarity between the input sample pair is build. To visualize how the algorithms works on an example, we describe the procedure in Example 13.

Example 13.
This example is based on the product-line evolution presented in our example from Section 3.1. The feature models of the running example before evolution (F) and after the evolution step (F') are taken as input for Algorithm 3. Furthermore, the input samples S and S', shown in Table 3.15, were derived from the respective feature model.

Table 3.15: Filter Identical Match Different Configurations: Input Sample for Combined Metric

Configuration	S	S'
C1	{D}	{U,W}
C2	{D,C}	{D,C}
C3	{D,N}	{U,N,UW}
C4	{D,C,N}	{D,N}
C5	{U}	{D}
C6	{U,N}	{D,C,N,W}

By looking at Table 3.15 we can see, that both samples contain six configuration. Between them, three pairs of identical configurations can be found. As described in Algorithm 3 the set of identical configurations is found, by building the intersection between S and S'. For our example the set visualized in Equation 3.21 is generated.

$$\begin{aligned} identicalConf &= S \cap S' \\ &= \{\{D\}, \{D, C\}, \{D, N\}\} \end{aligned} \tag{3.21}$$

We remove the identical configurations by building the set difference between original samples and the set of identical configurations. Thereby, two new intermediate samples I and I' are generated. Both samples are shown in Table 3.16.

https://doi.org/10.24355/dbbs.084-201812111412-0

Table 3.16: Filter Identical Match Different: Sample after Identical Configurations Removed

configuration	S	S'
C1	{U,N}	{U,W}
C2	{D,C,N}	{U,N,UW}
C3	{U}	{D,C,N,W}

Table 3.17: Filter Identical Match Different: Similarities after Identical Configurations Removed

	C1'	C2'	C3'
C1	0.67	0.83	0.33
C2	0.17	0.33	0.83
C3	0.83	0.67	0.17

After building the intermediate samples I and I' we apply the 1:1 matching to them. Table 3.17 visualizes the calculated configurations similarities for all possible configuration pairs. Those configuration similarities, are calculated with the hamming similarity as described in Section 3.4.1. Through the matching process, the similarity list displayed in Equation 3.22 is generated.

$$simList = (0.83, 0.83, 0.83) \tag{3.22}$$

$$simList = (0.83, 0.83, 0.83, 1, 1, 1) \tag{3.23}$$

The similarity list displayed in Equation 3.22 does not contain any information about the identical configurations, previously filtered. To integrate those information into the similarity list, we add a value of one, for each identical configuration. This way we build the similarity list displayed in Equation 3.23. After doing so we aggregate the values contained in the similarity list to a single value representing the similarity between the sample pair. Formula and result for this calculation is given in Equation 3.24.

$$\begin{aligned} arithMean(simList) &= \frac{\sum_{i=0}^{|simList|} sim_i}{|simList|} \\ &= \frac{5.49}{6} \\ &= 0.915 \end{aligned} \tag{3.24}$$

The calculation for this example results in a value of 0.915, which indicates a similarity between the input samples. By analysing the example sample pair manually, we identify three identical configuration. Furthermore, we identify configuration pairs, which only differ in one selected feature. So, by analysing the sample pair manually, we would come to the result, that those samples are very similar to each other, the same as the metric indicates. Hence, we can say the metric reflects the intuitive under standing of similar samples. □

By applying the procedure described in Algorithm 3, the disadvantages of *Ratio of Identical Configurations* and *Mean Similarity of Configurations* are mitigated by the respectively other metric.

https://doi.org/10.24355/dbbs.084-201812111412-0

The *Ratio of Identical Configurations* metric for example, suffers under the challenge that small changes between configuration (e.g. one feature changed) leads to the estimation, that those configurations are completely dissimilar (similarity = 0). In regard to many application areas, this estimation is not favourable. By using *Filter Identical Match Different Configurations* we mitigate this challenges, by using the 1:1 matching on all configuration, which are not completely similar (similarity = 1). This way, similarity value for non identical but similar configuration will be represented in the final similarity value as well as identical configurations

The 1:1 matching used by *Mean Similarity of Configurations*, suffers under the challenges of using a heuristic matching procedure. Due to the heuristic, it is not guaranteed that always the best matching pair is found. It could happen that identical configurations are not mapped together, because one partner of the identical pair, is already mapped to another configuration with less similarity. This challenge is mitigated by *Filter Identical Match Different Configurations*, because identical configurations are already filtered out and, before the matching starts. This way we can guarantee, that all identical configurations will be represented in the similarity estimation.

3.6 Algorithm for Stable Product Sampling

In this section we provide conceptual ideas for a sampling procedure, which considers the evolution history of product lines. We define this procedure as alternative to the sampling algorithms presented in Section 2.2. Even though the procedures presented in Section 2.2 work quite well to generate samples on one product-line version, non of them considers the product-line evolution as sampling factor. By considering the product-line evolution history, we provide the possibility to generate stable samples between product-line versions.

To have a sampling procedure which enables us to generate stable samples, holds advantages for application areas such as performance or regression testing of product lines. A use case where stable sampling helps to qualify results of product-line performance-tests is described in Example 14.

Example 14.
Imagine we have two versions of a product-line. The current version at time t0 is called P0 and the previous version at time t-1 is called P-1. We are interested in how the performance of our product line developed in the evolution step from t-1 to t0. One sample for each product-line version is calculated by one of the established procedures presented in Section 2.2. We make a performance estimation on each of those samples. While doing so, we make sure the test conditions for both samples are the same. So, differences in performance can only be introduced by the products contained in the tested samples. Regardless of the performance result, the question of how significant the results are, comes up. If the samples tested are similar the results can be seen as significant, otherwise not. To be sure the performance test results are significant, the samples must be analysed by using the previously defined metric (Section 3.2).

https://doi.org/10.24355/dbbs.084-201812111412-0

However, by using a sampling algorithm which considers the product-line evolution history, we can guarantee similarity between samples. This way performance test will always be significant, in regard to the samples used. □

Aim 1: Provide most stable sample from evolution step to evolution step
As Example 14 has shown, for some applications it is important that a sample algorithm produces stable samples. However, established sample algorithms do not consider the stability between samples as an criteria. To change that, we aim to develop a sampling algorithm which produces most stable samples from one evolution step to the other. As described in Section 3.2 stability between samples, is defined as their similarity to each other. To put it another way, the sample algorithm needs to produce most similar samples from one evolution step to another. The optimal similarity between samples is a value of one, as defined in Requirement 2. Because of changes during the evolution step, a full similarity (value of one) is not possible in most cases. Furthermore, how similar two samples can be to each other, depends on how much the product line changed during an evolution step. Because of this, it is difficult to define a precise aim, other than the similarity should be as close as possible to one.

Requirement 1: Work without product-line evolution history
As previously described, aim of the new algorithm is to sample in regard to a product-line evolution history. However, the algorithm needs to be usable even without a provided product-line evolution history. This means, it needs to provide the possibility to sample without previous calculated samples. We require for our algorithm that it can be used based on a provided evolution history, which is composed of feature models representing the product line in different time steps. Furthermore we require that the algorithm can run on a single feature model, to provide a sample.

Requirement 2: Start point is either, sample or product-line version
This requirement is about the initial input for the algorithm. It should provide the possibility to start an evolution based sampling from scratch. That means the initial input for the algorithm is only an evolution history provided as feature models. The algorithm needs to calculate the first sample by it self. Another possibility is, that a previously calculated sample should be used as starting point for the algorithm. In this case, the initial input for the algorithm is the evolution history as well as the previously calculated sample. We require, that the algorithm works in both cases.

Requirement 3: Final sample is valid for the respective product-line versions
Sample provided by the algorithm are used to analyse the respective evolution steps of the product-line. So, the algorithm needs to guarantee, that the sample is valid for the respective feature model provided in the evolution history. Therefore, we require that the new algorithm checks the validity of produced samples on the respective product-line versions.

Requirement 4: Sample conforms to user defined test coverage
Section 2.2 describes, that many established sample algorithms can conform to com-

https://doi.org/10.24355/dbbs.084-201812111412-0

binatorial interaction testing [OMR10a, POS+12, PSK+10]. This means, the calculated samples try to cover a predetermined coverage of feature interactions, the so called T-Wise coverage. Typical coverages used for sampling are pair wise coverage (T = 2) or three wise coverage (T = 3). Established sampling algorithms provide the possibility to choose the degree of coverage and guarantee that the samples conform to this coverage criteria in the end. We require for the new algorithm that it guarantees up to pairwise coverage for calculated samples.

Assumption 1: Product-line history is list of feature models.
According to Requirement 2, the algorithm takes the product-line history as input to calculate stable samples. We assume that the whole product-line history is provided as ordered list of feature models. In regard to the ordering, we expect an ascending order from the oldest to the newest version. If such a list is provided the algorithm needs to calculate an initial sample for the first feature model in the list. Alternatively the initial sample can be provided by the user. If such a starting point is given, we assume it belongs to the predecessor of the first feature model provided in the list. In such a case, the algorithm does not need to calculate the initial sample separately and can start by calculating samples based on evolution history.

Assumption 2: Identical samples are most stable
For the previously defined stability metrics the highest stability value is one. As Requirement 2 states, this value is only reachable, if both samples are completely identical. Hence we assume, identical samples as aim for the calculation.

3.6.1 Preservative Approach

This approach focuses on maximising the stability between samples of two consecutive product-line versions. As Assumption 2 indicates, the most stable sample between two product-line versions is the identical sample. Hence this approach tries to maximize stability by reusing a sample of the previous product-line version. By doing so, a list of consecutive samples is calculated, over the product-line evolution history. Each sample in this list is most similar to its predecessor. The basic procedure to do so, is visualized in Algorithm 4. How this algorithm works is described in Example 15.

Algorithm 4. Naive Procedure of Preservative Sampling

```
1   begin function basicIncrementalEvolutionSampling(FeatureModelList,
        intisalSample, globalSampleTime)
2     sampleList // List of samples, to represent the evolution history
3     sampleTime ← globalSampleTime / |FeatureModelList|
4
5     if(initialSampe != null)
6       S ← initialSample
7     else
8       F ← FeatureModelList[0]
9       S ← sampling(F)
10      sampleList <− S
```

https://doi.org/10.24355/dbbs.084-201812111412-0

```
11        FeatureModelList ← FeatureModelList \ F
12      end if
13
14      while(FeatureModelList has next)
15      do
16        F' ← FeatureModelList next element
17        for (S has conf)
18          if (isSatisfiable(F', conf))
19            continue
20          else
21            S ← S \ conf
22          end if
23        end for
24        while ((Coveredcombinations(F',S) < coverage) ∧ (passedTime < time
              )) do
25          conf_new ← generateConf(F')
26          if (conf_new ∉ S)
27            S ← conf_new
28          end if
29        end while
30        sampleList ← S
31      end while
32      return sampleList
33    end function
```

Listing 3.4: Preservative Sampling: Concept for a Naive Procedure of Preservative Sampling

Example 15.
Imagine we have a product-line evolution history, composed of three evolution steps at the time t_{-2}, t_{-1}, t_0. So that, we can derive three different feature models (F_{-2},F_{-1},F_0) from them. Now we want to generate stable samples between consecutive feature models. Pairs of consecutive feature models in our example would be (F_{-2}, F_{-1}) and (F_{-1}, F_0). To do so, we use the function described in Algorithm 4, to calculate a sample.

The input values for the algorithm are a list of feature models and optionally a initial sample, as described by Assumption 1. In our case the list of feature models contains feature-model version (F_{-2},F_{-1},F_0) and no initial sample is provided. Furthermore we enable the user to set a global sample time. This time value defines how long the algorithm can run in total. By setting the global sample time to a higher or lower value, the user can decide if a fast result or am extensive coverage should be focused by the algorithm. For example, if we disable the global sample time (equals infinite sampling time), the algorithm runs until the defined T-Wise coverage is reached.

Initially, an empty list of samples is created (Line 2). It represents the product-line evolution history and is filled over the run time of the algorithm.

Algorithm 4 recognizes, that no initial sample is provided (Line 5 - 12) and starts with calculating a sample S for the first feature model in the provided list (Line 9). In

https://doi.org/10.24355/dbbs.084-201812111412-0

our example this is F_{-2} Because it is the first feature model of the evolution history, the sampling uses one of the established procedures to calculate the sample. After the sample S is calculated the feature model F_{-2} is removed from the list of feature models and S is taken as the new initial sample, for further calculations.

After the initialization of sample S, the algorithm starts calculating samples by incrementally considering the evolution history (Line 14). The process runs until no unconsidered elements remain in the list of feature models. The first unconsidered feature model of the list, is taken as current feature model F' (Line 16).

The next step of the algorithm is to check if, the configuration in sample S are still valid in regard to the current feature model F'. This is done, by using a satisfiability solver with S and F' as input. For each configuration of S the algorithm checks, if the configuration is still satisfiable on the feature model F'(Line 17 - 23). If this is true, the configuration remains in S (Line 16), if not it is removed (Line 18).

Sample S, which now contains only valid configurations is used as base sample for the feature model F'. It is checked, if the sample already conforms to the user defined interaction coverage (Line 24). If so, the sample S is added to a list of samples and the algorithm starts with the next feature-model version in the list of feature models.

However, if the coverage is not yet reached by sample S and enough testing time remains, new configurations are calculated by the algorithm. If the algorithm calculates a configuration already contained in S, this configuration is ignored. So, only configurations not contained in S are added to it. As mentioned before this process runs until no testing time remains or the required interaction coverage is reached. If one of the exit criteria is met, sample S is added to the list of samples and the next feature model is analysed with sample S as initial sample.

After every feature model is analysed, the algorithm finishes and a list of samples, which represent the product-line evolution history is returned. □

3.6.2 Reuse of Established Algorithms

As initially defined the final samples of Algorithm 4 need to conform to different requirements. For example, it needs to calculate a sample which is valid for the respective product line. That means in line 25 the algorithm needs to guarantee, valid configurations. Furthermore, Requirement 4 defines that a certain coverage need to be full filled by the final sample. Proofing that the self implemented solution, full fills both requirements will need high test and validation efforts. Contrary, established sampling algorithms can be used to guarantee those requirements. Established procedures to calculate samples, are highly tested and validated, to full fill exactly the requirements mentioned before. Because of this, it is aspired to integrate one of the established sampling algorithms as core functionality of Algorithm 4.

On first sight any sample algorithm, which takes a feature model as input, can be used to produce the initial sample and the additional configurations for Algorithm 4. However with a closer look at this class of sample algorithms [VAHT+18], the subclass of incremental sampling algorithms, stands out more prominently than the others. Representatives are for example, MoSo-PoLiTe (Model-based Software Product Line Testing) [OMR10b] and IncLing [AHKT+16]. Both algorithms provide

https://doi.org/10.24355/dbbs.084-201812111412-0

the possibility to take an already existing sample as input for the sampling process. Because of this, newly generated samples can be based on the previously calculated sample.

In regard to Algorithm 4, this incremental sample calculation is exactly what is needed. As described in Example 15, samples for newer product-line versions will be calculated by reusing the sample of the previous version. This is exactly what incremental sampling algorithms are capable of. Hence parts of Algorithm 4 can be substituted by already existing incremental sampling approaches. The result of this substitution is shown in Algorithm 5.

Algorithm 5. Advanced Procedure of Preservative Sampling

```
1   begin function basicIncrementalEvolutionSampling(FeatureModelList,
      intisalSample, globalSampleTime)
2     sampleList // List of samples, to represent the evolution history
3     sampleTime ← globalSampleTime / |FeatureModelList|
4
5     if(initialSampe != null)
6       S ← initialSample
7     else
8       F ← FeatureModelList[0]
9       S ← incrementalSampling(F,sampleTime)
10      FeatureModelList ← FeatureModelList \ F
11    end if
12
13    while(FeatureModelList has next)
14    do
15      F' ← FeatureModelList next element
16      for (S has conf)
17        if (isSatisfiable(F', conf))
18          continue
19        else
20          S ← S \ conf
21        end if
22      end for
23      S ← incrementalSampling(F',S,sampleTime)
24      sampleList ← S
25    end while
26    return sampleList
27  end function
```

Listing 3.5: Preservative Sampling: Advanced Procedure

By comparing the first algorithm (Algorithm 4) and the second algorithm (Algorithm 5), we can see, that two essential parts of the first algorithm can be substituted by incremental sampling procedures. The first is the initial sampling in line 9 of Algorithm 4. The second part is the while loop from line 24 to 29. By substituting

https://doi.org/10.24355/dbbs.084-201812111412-0

those parts with an established incremental sampling approach, we can guarantee that the results conform to Requirement 3 and Requirement 4.

3.6.3 Challenges

As described above, the preservative approach to generate stable samples is based on the assumption, that identical samples are most stable. Because of this assumption, the approach tries to maximize stability by reusing as many configurations of the previous sample as possible to build the new sample. Configurations of previous samples should full fill most of the coverage criteria. Unfulfilled coverage is covered by newly calculated configurations. This behaviour, results in possible larger samples, in comparison to standard sampling approaches.

Beside the amount of identical configurations, the size difference between samples is another factor of stability as Requirement 2 shows. So, generating samples which grow uncontrolled in size, can have negative influence on stability. To counter the unchecked growth, a size limit for samples can be defined. Doing so, can keep samples nearly the same size, so that the stability is not influenced negatively. However, limiting the sample size also means limiting covering capabilities of the sample. By regarding those possibilities it is clear that controlling the growth of sample sizes is a challenge, which can not be solved easily.

To discuss how the basic preservative approach can be adjusted to keep a balance between all three sample criteria is a challenge, which is not discussed in this work.

3.7 Summary

In this chapter we presented the basics about measuring stability between samples over the product-line evolution history. First of all, we defined stability between samples, as the degree of similarity between them. The definition can be found in 3.1. Based on this definition, we derived the following requirements and assumptions for measuring stability, in Section 3.2.

- Requirement 1: Metric result is a single value.
- Requirement 2: Metric results are normalized values between 0 and 1.
- Requirement 3: Metrics consider selected as well as deselcted features.
- Requirement 4: Aggregation consider all intermediate values.

- Assumption 1: A sample is a set of configurations.
- Assumption 2: The similarity is calculated between two samples.
- Assumption 3: A configuration is a set of concrete, selected features.

https://doi.org/10.24355/dbbs.084-201812111412-0

After defining those requirements and assumptions, we defined three metrics (*Ratio of Identical Configurations*, *Mean Similarity of Configurations* and *Filter Identical Match Different Configurations*). Each metric conforms to the defined requirements and works according to the assumptions given in Section 3.2.

As the name suggests the *Ratio of Identical Configurations* metric is based on the ration of identical configurations between two samples. In Section 3.3 is defined how the calculations behind the metric work and which design decisions were take to conform to the requirements. Because only identical configurations are consider in the similarity estimation, this metric is limited to application areas where such estimations are sufficient.

A more fine granular metric is defined in Section 3.4. The *Mean Similarity of Configurations* metrics works by comparing the similarity between configuration to produce intermediate results. Those intermediate results are than aggregated to represent the sample similarity. During the process of conceptually developing *Mean Similarity of Configurations* different aspects of estimating sample stability via measuring configuration similarity came up. Those aspects are the following:

- Which metrics can be used to calculate configuration similarity.(Jacard or Hamming similarity)
- How to handle different feature sets. (Union or Intersection operation)
- Which aggregation method is used? (Arithmetic mean, Median, Sum, Max / Min)
- Which configuration pairs are considered in the aggregation. (N:M or 1:1 mapping)

In Section 3.4 all of those aspects are discussed in detail. Each aspect can be handled by different techniques. Those techniques are presented and discussed. The aspect of choosing configuration pairs, takes a special role. For this aspect two techniques can be derived the N:N and 1:1 Mapping. Both of those techniques contain different design decisions, which influence the final stability value. For example it makes a difference, if the configurations are matched based on minimum or maximum similarity. Furthermore a design decision must be taken, considering unmatched configurations. Eventually the best technique to handle the aspects, in regard to the defined requirements is chosen.

https://doi.org/10.24355/dbbs.084-201812111412-0

In regard to the application area of reusing samples for performance testing the following techniques will be used to measure sample stability with *Mean Similarity of Configurations* metric.

- Hamming similarity for measuring similarity between configurations
- Union set operation to build a common feature set, between both samples
- Arithmetic mean value as aggregation method.
- 1:1 Mapping to decide which configurations are mapped together
  - Mapping based on maximal similarity
  - unmatched configuration get a similarity of zero

The last metric defined in this chapter is the *Filter Identical Match Different Configurations*. As Section 3.5 describes, this metric is a combination of basic ideas from *Ratio of Identical Configurations* and *Mean Similarity of Configurations*. It filters the identical configurations before applying the *Mean Similarity of Configurations* to the sample pair. This way, it is possible to guarantee that identical configurations will be considered in the final similarity estimation.

Additional to the definition of stability metrics, we define the concept behind an algorithm to produce stable samples over the course of the product-line evolution history. This algorithm is based on the idea that identical configurations are most similar to each other. Therefore, the algorithm is based on the simple principle of reusing as many configurations from the previous sample as possible. Doing so, the final sample in the product-line evolution history is incrementally build. Due to this incremental character, parts of the algorithm can be substitutes by already existing incremental sampling techniques.

This chapter focuses on conceptual descriptions and definitions on what sample stability is and how it can be measured. No implementation details are provided, for the different techniques. Chapter 4 provides more details on the implementation process of the metrics and preservative sampling algorithm.

https://doi.org/10.24355/dbbs.084-201812111412-0

https://doi.org/10.24355/dbbs.084-201812111412-0

4. Tool Support for Stable Product Sampling and its Evaluation

In the previous chapter we presented conceptual ideas on how the stability of product samples can be measured. Furthermore, we presented the concept for a sampling procedure to generate stable samples. Now, we present the required tool support to implement the conceptual ideas. In the scope of this master thesis, we implement the concepts as stand alone tools. However, we envision to integrate our conceptual ideas into a larger product-line analysis frame work, such as FeatureIDE [ABKS13, TKB+14, KTS+09]. Furthermore we want to support research on product-line stability. Therefore, we published our code[1] artefacts as well as our data[2] artefacts produced during the implementation as open-source repositories.

As already mentioned, we envision to integrate our implementations into an product-line evaluation framework, such as FeatureIDE. To prepare for the integration into FeatureIDE our standalone tools are based on the FeatureIDE library, provided by the developers. For a better understanding of FeatureIDE and how its functionality can be used by third party developers, we start this chapter by introducing FeatureIDE and its library. In this context we show the structure of FeatureIDE and how our own implementations will be connected to them.

After introducing FeatureIDE, we describe how the stability metrics described in Chapter 3 are implemented. As a first implementation of the metric concepts, a stand alone calculation system is created. This system, combines all three metrics into one tool, to calculate stability for one product-line history with a single execution. Even though the first implementation combines all three metrics into one program, we keep the metric implementations independent of each other. Details about the implementation are described in Section 4.2.

[1] https://github.com/PettTo/Master_Thesis_Tools.git
[2] https://github.com/PettTo/Master_Thesis_Data.git

https://doi.org/10.24355/dbbs.084-201812111412-0

Beside calculating the stability of established sampling algorithms, we want to compare them with an algorithm which aims to produce most stable samples between two product-line versions. Therefore, we developed a concept for such an algorithm in Section 3.6. The implementation of this algorithm is described in the third section of this chapter (Section 4.3).

As described in Chapter 2, the Linux kernel is a large scale product line with a long history to analyse. Hence, it is qualified as an addition to our stability analysis. However, the Linux model is encoded in a special description language called KConfig. The KConfig language can not be processed directly by the basic framework (FeatureIDE) used for our analysis. Hence, a conversion into an processable format needs to be implemented. In Section 4.4 we present concepts of the KConfig language for a better understanding. Furthermore, this section discusses the implementation of two possible methods of converting the Linux variability model into a processable format. The conversion methods are implemented as fork of FeatureIDE.[3]

4.1 FeatureIDE

FeatureIDE is an extensible framework for feature-oriented software development [TKB+14, KTS+09, MTS+17b]. The development of FeatureIDE, aims to improve the quality and comfort of feature-oriented software development by providing an integrated development environment. Therefore, users can handle all phases of feature-oriented development in FeatureIDE [MTS+17b]. Out of the extensive functionality FeatureIDE provides, the sample generation based on different sample algorithms is the most important for this master thesis.

In context of generating product samples, FeatureIDE supports many different sample procedures. Among others, the functionality to generate all possible products of a product line, a random sample, or a sample which covers a certain degree of T-wise coverage [AHMK+16]. In regard of generating T-wise samples FeatureIDE provides the sample algorithms Chvatal, ICPL, and IncLing. For the purpose of the evaluations of this master thesis, all of the three algorithms will be used to produce samples, as described in Chapter 2.

Internally, FeatureIDE works with a specific XML data structure to represent feature models [MTS+17b]. All internal functionality is based on this feature model format. However, to support product lines not natively developed in FeatureIDE, export and import functionality for different file formats is provided. Among others, feature models given as conjunctive normal form (CNF) formulas can be imported from *.dimacs or simple *.txt files.

FeatureIDE is an open-source framework based on the Eclipse platform. Because of this nature, users can freely extend the basic framework with new functionalities. This way, FeatureIDE does not only provide a vast amount of reusable functionality, but also provides the means to be extended by users. Furthermore, the developers of FeatureIDE encourage third party developers to reuse FeatureIDE functionality by providing an external FeatureIDE library [KPK+17]. This library contains all the core functionality of FeatureIDE, but is independent from Eclipse. So, third

[3] https://github.com/PettTo/FeatureIDE

https://doi.org/10.24355/dbbs.084-201812111412-0

party developers can implement light-weight Java applications based on FeatureIDE functionality independent of the heavy-weight Eclipse framework.

The vast amount of reusable functionality in regard to product sampling, the flexible extensibility make FeatureIDE a valuable basis for the work at hand. Furthermore, the possibility to reuse implemented functionality as library independent from eclipse makes it possible to run later implemented stability analysis as simple command line tools.

Usage of FeatureIDE Library

As described above, FeatureIDE provides many useful functionality in context of analysing product lines and generating samples. Those functionalities can be used via user interface of FeatureIDE or reused in own implementation via the FeatureIDE library. Thereby FeatureIDE provides powerful, and flexible means to be used as basic tool for our evaluation process. Furthermore we envision the code artefacts produced in the course of this master thesis to be integrated into a product-line analysis framework such as FeatureIDE. To make the transition between stand alone tools used in this master thesis and integration into a huge frame work easier it is best to align the implementations to the destination framework. Hence, functionality of the FeatureIDE library will be used as basis for our implementations. To provide a short overview of FeatureIDE library, Figure 4.1 shows the most important library components for our implementations and how they work together.

In Chapter 3 we present three different metrics to calculate sample stability. We want to use all three metrics to evaluate the stability of established sample algorithms such as ICPL,Chvatal, and IncLing. Hence, we need to implement those metrics as a calculation tool, which takes generated samples as input. To do so, we use the FeatureIDE library as stated before. Beside evaluating established sample algorithms, aim of this master thesis is to compare the sample stability of established sample algorithms with results of an own algorithm. To do so, we need to implement this algorithm as well. The concept behind our preservative sampling algorithm is described in Section 3.6. Similar to the implementation of our metrics, the implementation of preservative sampling is also based on FeatureIDE library function, for reasons stated above.

Figure 4.1 shows the implementation of preservative sampling (PreservativeSampling) as well as the stability calculation (StabilityCalculator) as separate units, which access different components of the FeatureIDE library from the outside. Both external units access the FeatureModelManager and the Sat Solver component. Additionally PreservativeSampling accesses IMonitor and PairwiseConfigurationGenerator of the FeatrueIDE library.

The FeatureModelManager component, is used to handle input files or file paths to create a FileHandler for the provided feature model formats. In case of the external implementations, the FeatureModelManager is used to read existing feature models from the file system. Therefore, the external units provide a file path to the FeatureModelManager component. Based on the content contained in the provided file, the respective FileHandler is created.

https://doi.org/10.24355/dbbs.084-201812111412-0

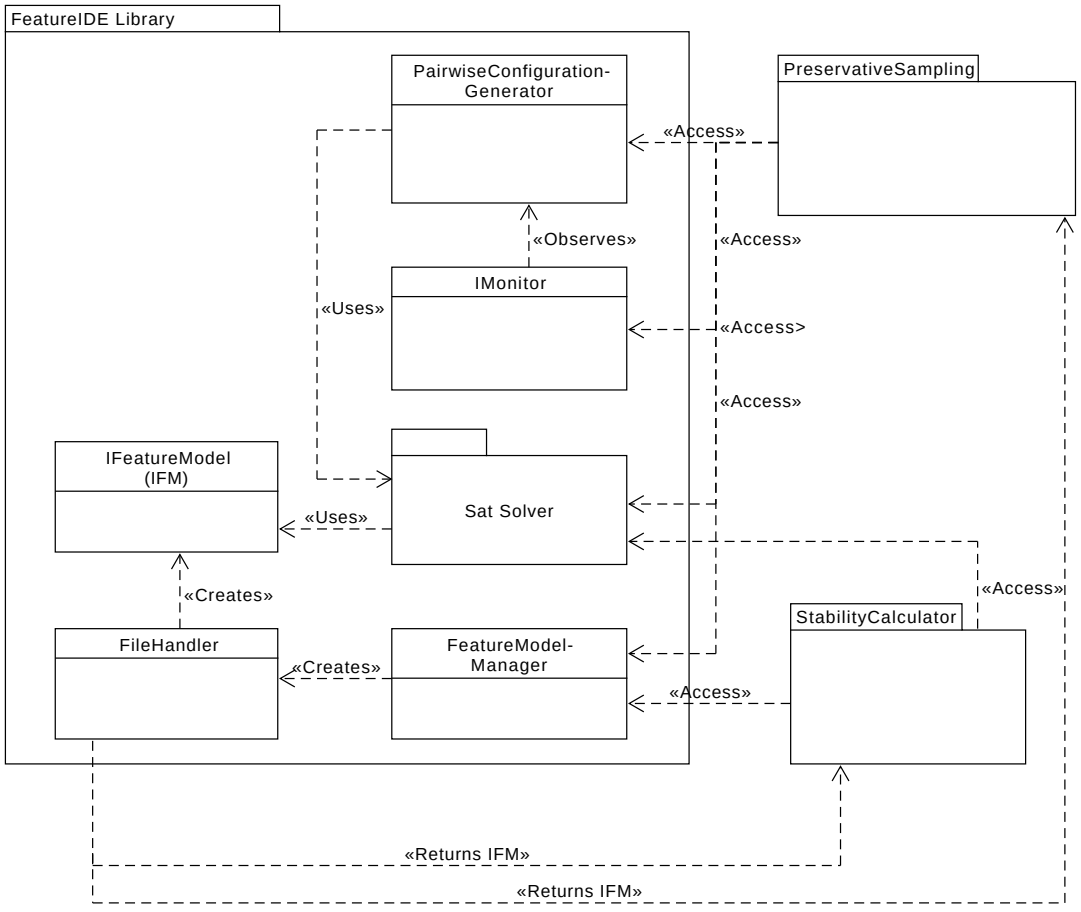

Figure 4.1: Overview Schema: FeatureIDE Library

As the name suggests, a FileHandler provides input and output operations for a specific feature model format. In context of the implementation of this master thesis, the component is used to create FeatureModel objects, based on the information loaded by the FeatureModelManager. The created object is returned to the external unit. A FeatureModel object, contains all information needed for further processing, such as sampling or validity analysis of configurations.

For the purpose of performing different validity analysis, both external units use the SatSolver component. Different problems such as finding dead and core features or checking whether a sample contains invalid configurations can be performed by using a satisfiability solver. The Sat Solver component implements those analysis functionality and provides it to third party developers.

In case of the PreservativeSampling unit, FeatureIDE's Sat Solver component is used to check the validity of generated configurations. Those configurations are generated by accessing the PaiwiseConfiguration generator. This library component finds new configurations for a sample based on a provided feature model and an optionally provided start sample. The PairwiseConfigurationGenerator component of FeatureIDE represents the implementation of the IncLing sampling algorithm [AHKT+16].

https://doi.org/10.24355/dbbs.084-201812111412-0

To monitor the generation of samples FeatureIDE library provides different kinds of monitor components. All of them implement the IMonitor interface shown in Figure 4.1. The interface defines all the functionality to monitor the generation of configurations for a sample. Hence, any component which implements the interface, can be used to observe and control the sample generation. For example it is possible to output status messages produced during the generation process. Furthermore, a monitor component can be used to cancel the sample generation, after a time out occurs.

As already mentioned, the schema shown in Figure 4.1 has the purpose to visualize the connections between FeatureIDE library components, which are most relevant for our own implementations. For the sake of brevity, not all components of the library are visualized. Furthermore, we visualized the components as black boxes, without showing their internal implementations.

4.2 Metric Calculation

In this section, we discuss the implementation of our stability metrics *Ratio of Identical Configurations* (Section 3.3), *Mean Similarity of Configurations* (Section 3.4), and *Filter Identical Match Different Configurations*(Section 3.5). Even though the conceptual principles of those metrics are described in Chapter 3, we now focus on how to apply the conceptual ideas to a program structure. As described in the respective sections of Chapter 3 all metrics aim to calculate a stability value between two samples. Therefore, each metric must conform to different requirements and assumptions. During the metric discussions, different concepts for each metric were discovered. The advantages and disadvantages are discussed in the respective sections of Chapter 3 for each metric. Considering the scope of this master thesis it is not possible to implement every concept for each metric. Hence, only the most promising approach for each metric is used. As an introduction we describe each metric shortly in the following paragraph.

Ratio of Identical Configurations is the most simple of the three metrics described in Chapter 3. It calculates the stability between samples based on how many identical configurations can be found between two samples. To do so, the Jacard metric [Jac12, TSK06] is used. Because of this, *Ratio of Identical Configurations* provides only low stability values if the samples are not identical to each other.

To consider also similar configurations, when assessing sample stability, the second metric *Mean Similarity of Configurations* is discussed in Section 3.4. Basic principle of the *Mean Similarity of Configurations* metric is to match similar configurations based on their similarity. In Section 3.4 different specialisations for *Mean Similarity of Configurations* are discussed. In regard to the implementation, only the approach listed in Section 3.7 will be used.

The third metric developed in Chapter 3 is the *Filter Identical Match Different Configurations*, described in Section 3.5. Aim of using the *Filter Identical Match Different Configurations* is to combine the advantages of *Ratio of Identical Configurations* and *Mean Similarity of Configurations* to produce a more realistic stability value.

https://doi.org/10.24355/dbbs.084-201812111412-0

Calculation of Stability

The first prototypes of our metric calculations are implemented as part of a small calculation system (StabilityCalculator), which provides controlling mechanisms as well as methods to read and write files to the file system. By implementing the metrics as part of this system, we keep testing and maintaining the metric components simple for this master thesis. However, we envision to integrate the metric components into a larger software product line analysis framework, such as FeatureIDE. Furthermore, we want to provide the metric calculations as library functions for third party developers. For both cases the metric calculations need to be independent of each other. Even though those endeavours are not in the scope of this master thesis, we design the program structure according to our aim. In Figure 4.2, a schema of the prototype's program structure is visualized, as UML class Diagramm. For the sake of brevity we display only the most relevant methods in the class diagram. Furthermore, the implemented metric calculations, use methods of the FeatureIDE library, which are not explicitly visualized in the schema.

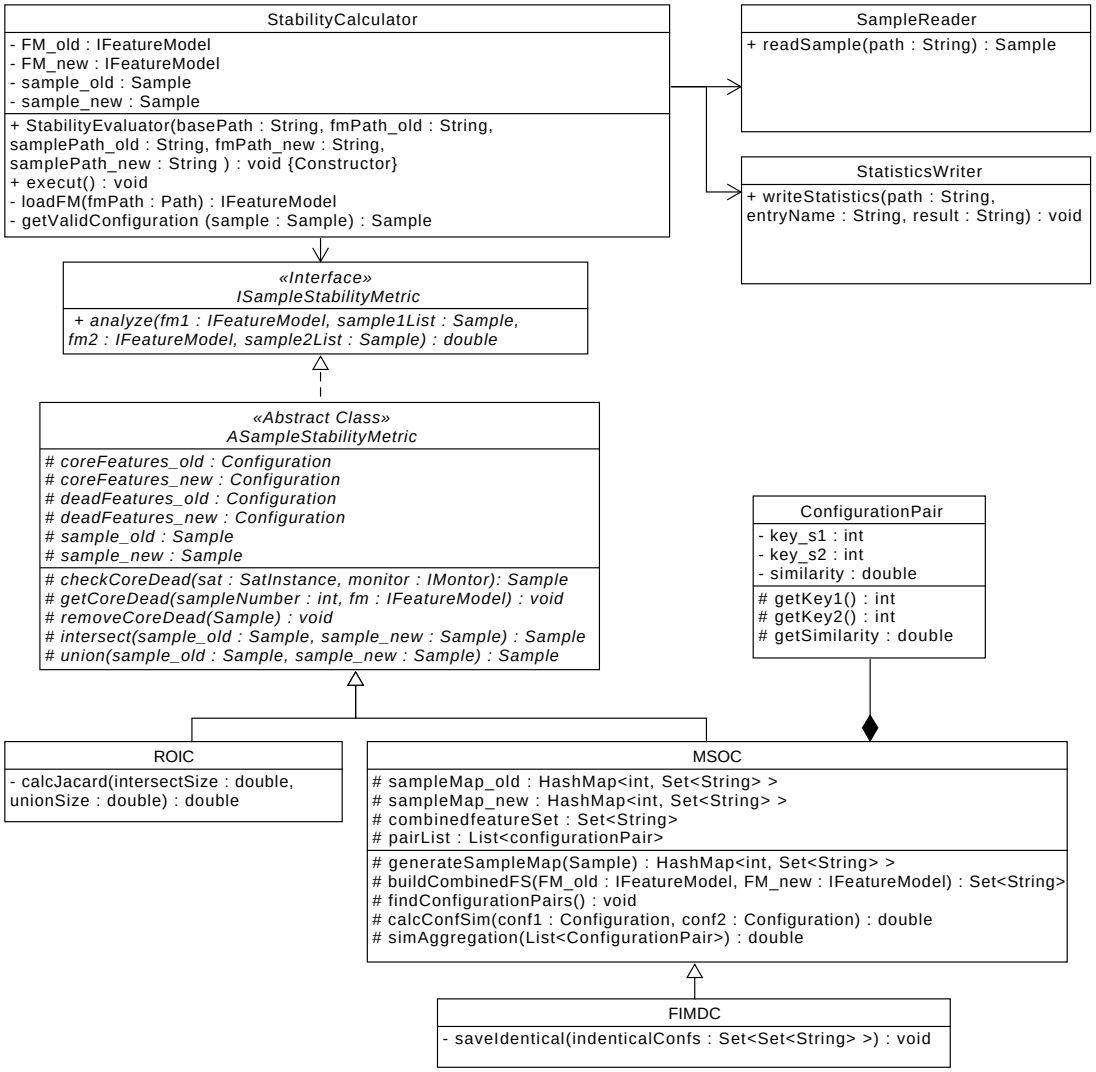

Figure 4.2: Class Diagram for StabilityCalculator

https://doi.org/10.24355/dbbs.084-201812111412-0

To simplify reasoning about the data flow, we will use the terms Sample and Configuration as representation for real samples and configurations, when discussing the program structure. Regarding the implementation those structures are represented by Java set data types. A Configuration is represented as a set of feature names (Java objects) and a Sample is represented by a set of Configurations.

The programs core unite is the class StabilityCalculator. This class manages the programs execution. Therefore, it contains an execute method, which starts the metric calculation process. Furthermore, it serves as the data storage for the input samples, and their respective feature models. To load the feature models, FeatureIDE library functions are used. The workflow of loading a feature model is combined in a method called loadFeatureModel.

The functionality to load samples from file system is grouped together in the SampleReader class. It serves as the main input controller of our design. To load a sample from file system, the readSample method is called. It expects the file location of a sample on the file system. The provided file path is checked and if it is valid, a sample is created from the input data and returned. Otherwise the program returns an exception.

While the SampleReader class serves as the main input controller of our program structure, the SampleWriter acts as the main output controller. All needed functionality to write out files to the file system is encapsulated in this class. Access to those functions are provided by the writeSample method. It expects a file location to store the data, an entry name and results of the metric calculation as input parameters. After writeSample is called with the right input parameters, the validity of the provided file path is checked. If it is not valid, an exception is thrown. Otherwise, the provided name and the metric results are written as *.csv file to the file system.

Before the metric calculation can start, validity of samples in regard to the provided feature models need to be checked. Because samples as well as feature models are loaded separately, it can happen that they do not match. In this case, an invalid sample for a feature model would be processed in the metric calculation. This would result in an internal error, of the metric calculation. To avoid this scenario as soon as possible, the method getValidConf checks the conformance of sample and respective feature model. This method takes a sample and a feature model as input parameters and checks the validity of each configuration contained by the sample with respect to the provided feature model. Invalid configurations are removed from the sample, so that after the method execution only valid configurations remain in the sample.

To calculate similarity between the provided samples, the class StabilityCalculator needs to call any of the implemented metrics. Each metric is represented by an own class. The implementation of *Ratio of Identical Configurations*is represented by the class ROIC. Accordingly the classes MSOC and FIMDC are implementations of *Mean Similarity of Configurations* and *Filter Identical Match Different Configurations*. In case of the StabiltyCalculator program, each metric calculation class is called once, with the sample pair as input parameter. Hence, after the program is finished three stability values can be provided.

Even though this program uses all metric calculations one after another, our software design considers future use cases, where each metric can be used separately. For

https://doi.org/10.24355/dbbs.084-201812111412-0

future applications of the metric calculators (ROIC, MSOC, FIMDC), we designed them as implementations of the ISampleStabilityMetric interface. This way, each metric calculator can easily be swapped against another. The interface provides access to the metric calculation via the analyze method. To start calculating the stability value between two samples the analyze method of a metric calculator needs to be called. As input parameters the samples and their respective feature models are expected.

While analysing the concepts of our metric definitions, we realised that a lot of functionality is used by two or more of the metrics. So we decided, to design an abstract class called AMetric to group those functionality together. Due to the grouping of functionality, a better maintenance and extensibility of the implementations is reached. Most of the grouped functionality considers the processing of core and dead features. Even though, Section 3.4 states, that a removal of core and dead features is not necessary for *Mean Similarity of Configurations* we still decided to group those functions together, because the ROIC class and the FIMDC class make use of them. Furthermore, by grouping this functionality together, we support future extension and alternative implementations of stability metrics.

Besides the functionality to process core and dead features, we implemented our own intersection and union method for the Sample data structure. Originally we wanted to use the Google Guava library [Goo18], which provides different set operations for Java data structures. However, during the tests of our implementations we realized that the google guava library shows error prone behaviour in regard to our used data structure.

Algorithm 1. Intersection Method

```
1   function: intersection(sample1, sample2)
2     intersection : Set<Set<String>>
3     ∀ conf1, conf1 : Set<String> ∈ sample1 : {
4       ∀ conf2, conf2 : Set<String> ∈ sample2 : {
5         if(|conf1| = |conf2|)
6           checkSet : Set<String>
7           checkSet ← addAll(conf1)
8           checkSet ← removeAll(conf2)
9           if(|checkSet| = 0)
10            intersection ← add(conf1)
11            break;
12          end if
13        end if
14      }
15    }
16    return intersection
17  end function
```

Listing 4.1: Metric Implementation: Intersection Method

Algorithm 1 shows the conceptual algorithm behind the implementation of our intersection method. The function starts creating an intersection object (Line 1) to

https://doi.org/10.24355/dbbs.084-201812111412-0

save all configurations (sets of string values), which are contained in both of the provided samples. To find identical configurations of sample1 and sample2 it is necessary to iterate over both structures. For each configuration contained in sample1, each configuration contained in sample2 is analysed (Lines 3 and 4). Configurations can be similar, only if they contain the same amount of feature names (Line 5). If both configurations contain the same amount of elements, it is to be determined if the feature names contained in conf1 and conf2 are equivalent. The check is implemented in three steps. Step one adds all feature names of conf1 to an intermediate set (Line 7). In step two the feature names contained in conf2 are removed from this intermediate set (Line 8). If the configurations are identical, the intermediate set contains no elements, after removing conf2 (Line 9). In case the configurations are identical conf1 can be added to the set representing the intersection (Line 10). Thereafter, the next configuration of sample1 can be analysed. After all configurations of sample1 are analysed, the aim of creating a set containing all identical configurations of sample1 and sample2 is reached.

For the implementation of the union method, Java functions are used. Java provides the set data structure, which avoids duplicates by nature. Hence, to implement the union between two sets, we just need to add the elements of one set to the other. If duplicate elements exist between both samples, the insertion of the duplicate element is avoided. Hence, after adding all configurations (Set<String>) of our sample2 into sample1, a union set is build.

As shown in Figure 4.2, the ROIC and the MSOC class, directly implement the abstact class ASampleStabilityMetric, which implements a lot of the functionality needed to calculate the *Ratio of Identical Configurations* metric. Hence, the implementation of the ROIC class itself is fairly easy. The only functionality implemented into this class is the calculation of the Jaccard distance. This implementation follows the conceptual descriptions of Section 3.3.

In comparison to the ROIC class, the MSOC class does not profit a lot from the functionality implemented in ASampleStabilityMetric. Hence, more functionality needs to be directly implemented into this class. In accordance to the definition of *Mean Similarity of Configurations*, the MSOC class provides functionality to build a combined feature set (buildCombinedFeatureSet), based on the union between input feature sets. Furthermore, the class implements functionality to find configuration pairs, based on a matching between configuration similarities. The calculation of configuration similarity is done by using the Hamming similarity as described in Section 3.4.1. Regarding the matching process, the n:m matching heuristic described in Section 3.4.6 is used. To keep track of found configuration matches a new data structure is implemented. The so called ConfigurationPair data structure, consists of a similarity value and unique ID's of two configurations. As configuration id the object ID's of configuration objects are used. To match the object ID to the respective configuration two Java hashmaps are used as storage structures. The MSOC class provides a method to generate those maps.

Section 3.5 describes the *Filter Identical Match Different Configurations* metric as a combination of *Ratio of Identical Configurations* and *Mean Similarity of Configurations*. This concept is transferred to the implementation, by using the FIMDC class

https://doi.org/10.24355/dbbs.084-201812111412-0

as an extension of the MSOC class. By extending the MSOC class FIMDC implements automatically all the functionality implemented in ASampleStabilityMetric and MSOC. As described before those functionalities conform to the metric definitions of Section 3.3 and Section 3.4. Hence, they can be used in accordance to the metric definition in Section 3.5. A functionality that is missing, is the assignment of stability values to identical configurations. Hence, this functionality is implemented directly by the FIMDC class. To assign a stability value, to a pair of identical configurations, a ConfigurationPair object is created with the respective configuration ids as keys. In accordance to Section 3.5, the generated ConfigurationPair gets a stability value of one assigned.

4.3 Preservative Sampling based on IncLing

This section discusses the implementation of the preservative sampling algorithm. As described in Section 3.6 the preservative sampling algorithm aims to generate a most stable sample between two product-line versions. The basic concept of the preservative algorithm is already shown in Algorithm 4. The implementation is bound to different requirements and assumptions also described in Section 3.6. During the discussions of Section 3.6 it was discovered that the use of already existing incremental sampling procedures can simplify the implementation of our preservative sampling algorithm. An established sampling algorithm can be used as core functionality of the preservative sampling.

As result of the discussions in Section 3.6, two already established incremental sampling algorithms are named. Theoretically it is possible to use either MoSo-PoLiTe [OMR10b] or IncLing [AHKT+16] as basis for the implementation. However, we envision to integrate the preservative algorithm into FeatureIDE. Considering this purpose, using the IncLing algorithm, which is already integrated into FeatureIDE, as basis promises less difficulties for future work. Moreover, the incremental algorithm of MoSo-PoLiTe is integrated into an own tool chain [OMR10b], which complicates the access to it. Even though, the MoSo-PoLiTe tool chain can easily be integrated into any eclipse based feature editor [OZML11], the integration would still be as external plugin without access to the source code. Contrary IncLing is an open source tool.[4] Hence, using IncLing as basis for the preservative algorithm provides more flexibility. Based on the arguments stated above, we made the decision to use IncLing rather than MoSo-PoLiTe.

For the first approach of implementing the preservative sampling algorithm, we decided to implement the preservative sampling as a Java program, which uses FeatureIDE library functions [KPK+17]. This way, we support simple maintenance and flexible extensibility for the first prototype of the algorithm. Furthermore, we keep the module testing and debugging effort for our first prototype more simple.

We access the IncLing algorithm as described in Section 4.1. Because we use Incling as sampling algorithm, we do not need to implement any sampling specific functionality our self. However, we still need to handle the input / output data as well as

[4] https://github.com/FeatureIDE/FeatureIDE/blob/develop/plugins /de.ovgu.featureide.fm.core/src/org/prop4j/analyses/PairWiseConfigurationGenerator.java

https://doi.org/10.24355/dbbs.084-201812111412-0

possible data transformations. To do so, we developed the program structure shown in Figure 4.3.

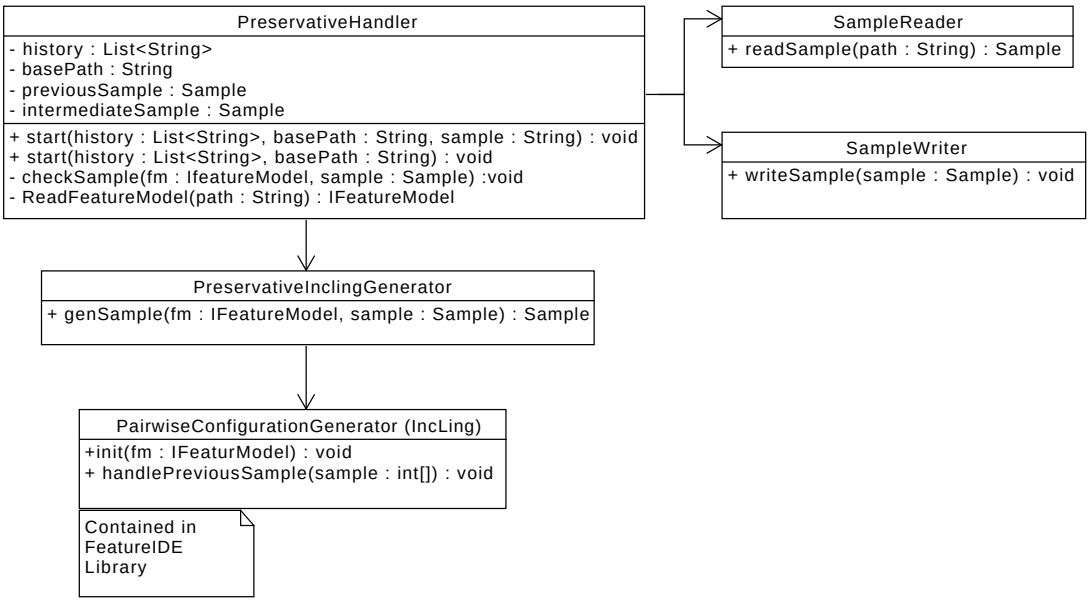

Figure 4.3: Class Diagram for Preservative Sampling

The schema shown in Figure 4.3 is a simplified class diagram, which represents the structure of the preservative algorithm tool. To keep the attention on important classes and methods, utility methods and classes are omitted in the schema. Another simplification we use is to omit utility classes of the FeatureIDE library which are used by the implemented components.

To make it easier to reason about the program structure and its data flow, we define the data structures configuration and sample. As the names suggest, those data structures represent samples and configuration of product lines. In accordance to real world product lines, a configuration consists of a list of feature names. The names are represented as Java objects. All feature names belonging to one configuration, are stored together as a Java list object. To build a sample, all configurations belonging to the sample are grouped together as Java list object.

The central unit of our preservative sampling implementation is the *Preservative-Handler*. As the name suggests, it handles the program workflow and connects all the other components with each other. The workflow starts when the *start* method is called. As input, users can provide a list of paths leading to the feature models of different product-line versions, the base path where generated samples will be stored as well as a path to an initial sample. The path leading to the initial sample is optional, just as Algorithm 5 suggests. The path list leading to the feature models is saved as list of string objects and the base path is stored as simple string variable. In case a path to the initial sample is provided, *PreservativeHandler* calls the *readSample* method of class *SampleReader*. This method checks the provided path and loads the sample, if the path is valid. In case of a valid path, the loaded sample is saved in the *PreservativeHandler* as previous sample. Otherwise, an exception is returned.

https://doi.org/10.24355/dbbs.084-201812111412-0

After processing the initial sample, the program starts to iterate over the list of provided Feature Models, as line 14 of Algorithm 5 indicates. For each entry in the list of feature model paths, the method *readFeatureModel* is called to load the respective feature model. To do so, the method checks if the given path exists. If this is not the case an exception is returned. Otherwise, a feature model in the FeatureIDE XML format is generated by using functionalities of the FeatureIDE library.

According to lines 16 to 22 of Algorithm 5, every configuration contained in the previous sample needs to be checked for validity on the newly loaded feature model. This is realised by implementing a function, which returns a list of valid configurations for the respective feature model. This list of configurations is saved as intermediateSample.

The next step of Algorithm 5 is to call the incremental sampling algorithm (IncLing). To do so, *PreservativeHandler* calls the *genSample* method of class *PreservativeGenerator* and provides the current feature model and the intermediate sample as input. *PreservativeGenerator* handles the incremental sampling process. This includes thread handling and a mechanism to stop the incremental sampling if a predefined time out is reached. To start the incremental sampling process, *PairwiseConfigurationGenerator* from the FeatureIDE library is initialized by calling *init* method. After initializing the *PairwiseConfigurationGenerator*, the sampling process is started by calling the *handlePreviousSample* method. This method expects a previously calculated sample as input. If this sample is empty, a complete new sample is calculated. Otherwise the algorithm will use the provided configurations as starting point to find additional configurations.

Any new configuration found by IncLing is added to an intermediate sample, which contains previous configurations, if any. After the execution of IncLing is stopped, the intermediate sample contains configurations which fulfil pairwise coverage for the current feature model. This resulting sample is returned to *PreservativeHandler* as intermediate sample variable. The old value of this variable is overwritten. The generated sample is saved to the file system, by calling the *writeSample* method of class *SampleWriter*.

After writing out the generated sample, the program control returns to *PreservativeHandler*. It overwrites the variable *previousSample* with the new generated sample. This ends one iteration of Algorithm 5 and a new feature model is loaded. This process continues until all feature models of the provided list are processed.

We use the described program as a simple jar file, which can be started from any command line interface. The input parameters are provided over the command line interface as well. By doing so, we have the possibility to automate the program execution. Even though we envision to integrate the algorithm into FeatureIDE's sampling tool chain, this is not in the scope of this master thesis.

4.4 Importing Linux Variability Models into FeatureIDE

As Chapter 2 states, we want to use the Linux kernel in our evaluation. The Linux kernel variability model is described in KConfig, a language especially developed

https://doi.org/10.24355/dbbs.084-201812111412-0

for the configuration of Linux kernel [ESKS15, Zip17]. KConfig is a powerful and complex language. This makes analysis of KConfig variability models rather difficult [ESKS15]. Even though, there are different tools for extracting the variability model from KConfig files, our tool of choice (FeatureIDE) does not provide such a functionality. Hence, we need to implement an extraction and transformation mechanism our self. To do so, we can use one of the already existing tools to transform a variability model in KConfig to a boolean representation. In Chapter 3 we defined that, our analysis of sample stability is based on the evolution of a product line. Therefore, we are not interested in a single version of Linux, but rather in the product-line history. Hence, we need to develop a work flow to automatically extract a boolean representation of the variability model from different Linux kernel versions.

The content of this section focuses on the variability model described in the KConfig language. We provide insides of the language concepts, describe the tool we use to transform a KConfig variability model into boolean logic. In this context we provide insides on how our automation work flow for the extraction works. Furthermore, we describe how FeatureIDE was extended to support the import and export of Linux variability models based on boolean constraints.

4.4.1 KConfig Language

The variability model of Linux kernel is represented in the KConfig language [Zip17]. KConfig is a special configuration language designed to describe the variability model of the Linux kernel [ESKS15]. Even though, the KConfig language was already described by many authors [Zip17, PGT+13, DvDP17, SLB+10], we give a short introduction into the language specifics to keep the work at hand self containing. Listing 4.2 shows a snippet of the Linux kernel version 4.15 of the x86 architecture. For the sake of brevity, we will not go into much detail of the language concepts, if readers have more interest on those details we refer to the official kconfig documentation [Zip17].

```
1  config GENERIC_BUG
2    def_bool y
3    depends on BUG
4    select GENERIC_BUG_RELATIVE_POINTERS if X86_64
5
6  config RAPIDIO
7    tristate "RapidIO support"
8    depends on PCI
9    default n
10
11 config GENERIC_BUG_RELATIVE_POINTERS
12   bool
13
14 config ARCH_DEFCONFIG
15   string
```

https://doi.org/10.24355/dbbs.084-201812111412-0

```
16      default "arch/um/configs/i386_defconfig" if X86_32
17      default "arch/um/configs/x86_64_defconfig" if X86_64
```

Listing 4.2: Configuration Snippet Linux Kernel 4.15

The basic configuration elements of KConfig are *config* elements shown in Listing 4.2. They represent the features of the kernel. Configs can be of one of the following types: boolean, tristate, string, hex, or int. In general, the type definition of a config is given in the next line after the confgi name. As an example, the Line pairs 11-12 and 14-15 can be taken into account. Type definitions accept an input prompt, as shown in Line 7. Prompts are shown to the user, during the configuration of Linux kernel. They represent text messages to help the user configure Linux kernel. Therefore, prompts do not influence the structure of the linux variability model.

The most simple config type is boolean. Those configs can be either selected (y) or deselected (n). Examples for this configuration type can be seen in Line 1 and Line 11 of example Listing 4.2.

Similar to simple boolean configs, tristate configs can also be selected or deselected. However, tristate configs can also be selected as modules (m). To select a tristate config as module means, the functionality is not active in the later product, but can be enabled at runtime. An example for a tristate config is *RAPIDIO* (Line 6).

The third kind of config elements are so called assignment configs. They can take a value of type string, hex, or int. Those values represent path names or other specific variables important for later execution. The configuration element *ARCH_DEFCONFIG* (Line 14) is an example for a typical string assignment config.

In case of *ARCH_DEFCONFIG*, another language specific feature can be seen. The used *default* keyword assigns a default value to the config, if no other value is manually set during the configuration process. For the example at hand, a default path is assigned to a configuration. Another way to define a default value is to use the *def_*, before the type definition. An example for this language construct is given in Line 2.

Another language feature is the possibility to define conditional behaviour by using the keyword *if*. By using it, dependencies to previously chosen configuration options can be defined. An example can be seen in Line 16, and Line 17. Depending on whether the config option *X86_32* or *X86_64* was chosen previously, Line 16, or Line 17 will be the chosen as default value.

Cross-tree constraints in the KConfig language are expressed by the attributes starting with *depends on* or *select*. An exemplary use of both language constructs can be found in Line 3, and Line 4 of Listing 4.2. The *depends on* keyword defines a dependency for the config element. Therefore, Line 3 of our example indicates, that *GENRIC_BUG* can only be selected if *BUG* is also selected. The *select* keyword defines config elements which must be selected together with the config element containing the *select* keyword. In regard to the example, Line 4 forces the config element *GENERIC_BUG_RELATIVE_POINTERS* to be selected.

Config elements can be nested in other config elements or under menu or choice entries (not shown in the example). Nested configuration options are important

https://doi.org/10.24355/dbbs.084-201812111412-0

for the kernel configurator to build a tree structured user interface. A menu group represents a simple grouping of config entries. The choice group represents a decision between config entries. Such a decision can be of boolean or tristate nature. In choices with the boolean nature only one of the grouped configurations can be chosen. The tristate nature indicates that only one feature can be chosen explicitly, but an arbitrary number of config elements can be chosen as modules.

4.4.2 Variability Extraction Tool

To use the Linux kernel as an example product Line in our evaluation, the variability model described in KConfig must be converted into a different format. Many tools were developed to do so, for example the Undertaker tool chain [VAM18], the Linux variability analysis tool (LVAT) [lva18] and KConfigReader [Kä18a]. Due to the broad language concepts of the KConfig language, complex expressions and special corner cases can appear. Conversion tools, need to handle those corner cases to allow reliable product-Line analysis. El-Sharkawy et al. [ESKS15] analysed the three named tools in regard of handling different corner cases of the KConfig language. Their results show that none of the tools can handle all corner cases. However KConfigReader handles most of the possible corner cases with the highest precision. Based on the result of El-Sharkawy et al. we choose the KConfigReader as conversion tool for the KConfig language.

KConfigreader is a tool developed by Kästner et al. [Kä18b] in context of the type chef tool chain [KGR+11], to convert KConfig-files to boolean formulas [Kä18a, Käs17]. The conversion of KConfig files with KConfigReader follows two steps. First, a KConfig file is read and transformed into an intermediate XML structure. The resulting XML file is saved as an .rsf file. To transform kconfig files into the intermediate XML structure, a modified version of undertakers [VAM18] dumpconfig component is used. As second step the intermediate XML structure is transformed into boolean formulas, stored in a model file (*.model). Alternative it is possible to generate an equivalent of the boolean formulas in Conjunctive Normal Form (CNF), as DIMACS file (*.dimacs). Regardless of the output file format, the results support boolean and non-boolean features, such as tristate features and assignment features of all kinds.

In the following we describe how KConfigReader handles Tristate options, Numeric and String options, option dependencies, invisible options, and Choices of Kconfig files. The information of the descriptions are obtained from Kästner et al. [Käs17].

Tristate options: KConfigReader translates the three-value logic of tristate options into propositional logic by introducing two mutually exclusive boolean variables. To mark a variable introduced for the purpose of handling tristate options the key word *MODULES* is used.

Numeric and String options: To handle Numeric and String options KConfigReader, searches through the Kconfig file to find all occurring values for such an variable. Thereafter, all known values are modelled as boolean variables. To make those variables mutually exclusive new constraints are introduced. By handling Numeric and String options in this way, KConfigReader prevents modelling each value of large or possible infinite domains.

https://doi.org/10.24355/dbbs.084-201812111412-0

Option dependencies: As the name suggest, option dependencies, describe dependencies between configuration options in KConfig. KConfigReader handles those constructs by directly translating them into propositional logic. The dependence from a configuration option to another is modelled by an implication operation.

Invisible options: The term invisible option, describes configuration options of Kconfig which are not shown to the user. This is possible, because only configuration options, which contain prompt are shown to the users. Therefore, any option without a prompt or where the prompt depends on other selected options is invisible. KConfigReader handles those configuration options similar to normal configuration option, by translating them into an constraint in propositional logic. Because of the visibility condition of invisible options, the resulting constraint contains at least implications.

Choices: As mentioned before Choices in Kconfig represent a grouping of configuration option, where at least one option must be selected and no two options can be selected at the same time. KConfgreader models simple choices as constraints, where at least one of the inner options must be selected. The constraints must also require that no two inner options are selected ate the same time.

4.4.3 Automated Extraction of Variability Model

As already introduced, we are not only interested in single product-line version, but rather in the history of the product Line. Therefore, we need a way to automatically transform the Linux kernel variability model from kconfig to boolean formulas. Using KConfigReader as conversion tool, we developed the automation work flow shown in Figure 4.4.

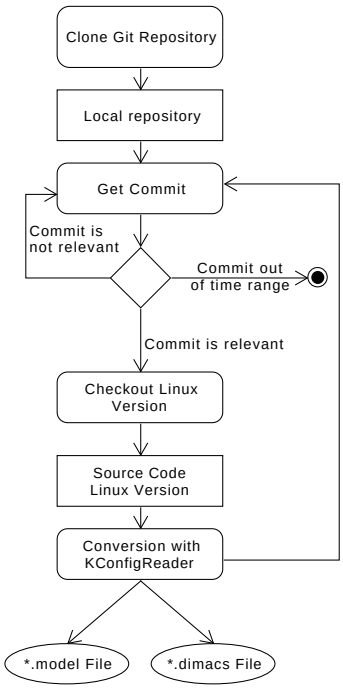

Figure 4.4: Flow Chart: Workflow KConfig Conversion

https://doi.org/10.24355/dbbs.084-201812111412-0

Before the workflow of Figure 4.4 can start, the KConfigReader needs to be initialized as described by Kästner et al. [Kä18a]. After everything is set up, the workflow starts by cloning the Git repository of the Linux kernel[5] to create a local copy of the repository. From the local repository a list of all commits in a specified time range. To specify a time range is necessary, because the syntax of KConfig files changed over the years and KConfigReader needs to be configured differently in accordance to the syntax used.

Further analysis relies on the feature model and the samples produced from it. Hence, the analysis results will not differ for different versions, if the respective feature models are the same. Therefore, we distinguish between relevant and irrelevant commits. All commits changing the feature model of Linux kernel, are seen as relevant, the others are irrelevant. Hence, the relevance of each commit is examined and only relevant commits are used for further processing. Furthermore, it is examined whether the commit date is out of the defined time range. If a commit date lies out of the defined time range, the end of processing is reached and the process terminates.

The relevant commits are checkout from the Linux Kernel git repository. By doing so, the source code of different Linux versions is available for further analysis. One after another, the Linux versions are used as input for KconfigReader. With KConfigreader the Linux variability model defined in the respective KConfig files is extracted and converted into a boolean formula. As output formats the model format (*.model) and the dimacs format (*.dimacs) are used. After processing a version of the Linux kernel, the next version is checked out and the processing repeats.

We implemented this automation workflow as a simple bash script. The script can run on any Linux machine, where git is installed and the KConfigreader is set up.

4.4.4 Conversion to FeatureIDE Model Format

As described before FeatureIDE uses an own XML file structure to represent variability models as feature models following the principles of Feature-Oriented Domain Analysis (FODA) [TKB+14]. All the internal functionalities of FeatureIDE work with this XML representation. Based on the results from Section 4.4.2 the internal FeatureIDE XML structure can be generated either by using the **.dimacs* or the **.model* file. In both cases a conversion from the input format to the XML representation needs to be done.

FeatureIDE already supports the import and export of dimacs files. This way no effort needs to be invested to get the FeaturIDE XML file format from a dimacs file. However, the way KConfigReader creates dimacs files, can cause challenges for the later evaluation. To create dimacs files, KConfigReader internally converts boolean formulas, into conjuctive normal form (CNF). The process of converting complex boolean expressions into CNF, can be time consuming. In order to save calculation time, KConfigReader uses the Tseitin transformation. This method divides complex boolean expressions into smaller sub expressions, which are more simple to transform into CNF. Each sub expressions is represented by a new variable. increasing

[5] https://github.com/torvalds/linux Last visited on 2018-10-07

https://doi.org/10.24355/dbbs.084-201812111412-0

the number of variables in the CNF formula, means increasing the number of features in the resulting feature model. This again, leads to challenges processing the feature models. For example, some sampling algorithms do not scale to well for large numbers of features.

Another challenge for follow up analysis, which comes with using the dimacs format as basis is, that tristate assignment features cannot be removed during the conversion into FeatureIDE's XML format. This takes the freedom of choosing if those features should be included in the follow up analysis. Even though FeatureIDE provides functionality to remove existing features from a an existing feature model (Slicing) [KST+16, KSTS16], this procedure is time consuming for large feature models [KST+16].

As an alternative to the already integrated dimacs import, we implemented an import and export functionality for *.model files into FeatureIDE. We need the import and export functionality to convert boolean constraints of the Linux variability model into the FeatureIDE XML representation of feature models.

Syntax of *.model files

The first step to successfully implement the import and export function for *.model files is to analyse the file formats syntax. To describe different syntax elements of the *.model files, Listing 4.3 shows selected syntax elements as examples.

```
1 #item 104_QUAD_8
2 ((def(PC104)&def(X86)&def(ISA_BUS_API)&def(IIO))|(def(PC104)&def(X86
      )&def(ISA_BUS_API)&(def(IIO)|def(IIO_MODULE)))|def(MODULES)|!
      def(104_QUAD_8))
3 #item IIO
4 #item ARCH_DEFCONFIG
5 (!def(X86_32)|def(ARCH_DEFCONFIG=arch/x86/configs/i386_defconfig))
6 #item BCH_CONST_M
7 (!def(BCH_CONST_M=1)|!def(BCH_CONST_M=5))
```

Listing 4.3: Syntax Elements of Model Files

Line 1 of Listing 4.3 shows how a feature is represented in a *.model file. Any feature contained in the variability model is listed as a single Line introduced by a #, followed by the feature name. That makes it easy to create a list of features. However, scanning the model file for features does not reveal the type and attribute of the feature. For example, feature *IIO* defined in Line 3, can be found by searching for the *# item* pattern. However, the tristate nature of this feature is only revealed by analysing the boolean constraint in Line 2. The *.model syntax refers to tristate features by adding the *_MODULES* keyword to them, as shown in Line 2. In regard to assignment features a similar notation can be found. For example, in Line 4 and Line 6, the features *ARCH_DEFCONFIG* and *BCH_CONST_M* can be found. However, their assignment feature nature is only be revealed after analysing the constraints in Line 5 and Line 7.

https://doi.org/10.24355/dbbs.084-201812111412-0

FeatureIDE supports simple boolean features as well as features extended by attributes. Hence, tristate and assignment features could be expressed in FeatureIDE, by using attributed features. However, currently the established sample algorithms cannot handle attributed features. Hence, tristate and assignment features need to be encoded into boolean-logic. To do so, an additional variable is introduced for every non boolean state a feature can have. For example a tristate feature such as *IIO* has the three states: selected, selected as module, and deselected. This can be represented by using a feature variable IIO and introducing one additional feature variable called IIO_MODULES. Both feature variables need to be mutually exclusive, which is guaranteed by the constraints of the *.model file. Due to the used encoding, the three tristate states can be represented, as Table 4.1 shows.

Table 4.1: Linux Conversion: Tristate Representation in FeatureIDE

Feature State	IIO	IIO_MODULES
deselected	deselected	deselected
selected as module	deselected	selected
selected	selected	deselected

For assignment features a similar encoding can be used. Different assignments to a feature, are represented by additional boolean variables. Those boolean variables need to be mutually exclusive, which is guaranteed by the constraints of the *.model file.

Boolean constraints of a *.model file, conform to a general abstract syntax schema. Each boolean constraint starts with an left parenthesis followed by an arbitrary number of subconstraint. Sub constraints are connected by a logical operator. The definitions end is indicated by a right parenthesis. Figure 4.5 shows the general syntax schema of subconstraint.

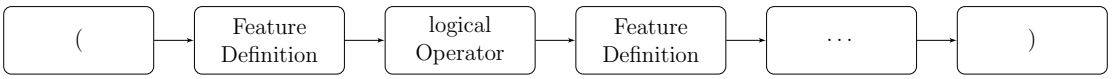

Figure 4.5: Abstract Schema for Sub Constraints in Model Files

A sub constraint always starts with an left parenthesis followed by a feature definition. An abstract schema for feature definitions in *.model files is given in Figure 4.6. Feature definitions are connected by a logical operator, such as *AND* or *OR*. A logical *AND* is represented by the symbol &, while a logical *OR* is represented by the symbol |. Expressions, such as implication and equivalence do not appear in the *.model files produced by KConfigReader. The definitions end of a sub constraint is indicated by a right parenthesis. In between of the left and right parenthesis an arbitrary number feature names followed by logical operators can be expressed. Furthermore, subconstraint can be nested in other subconstraint.

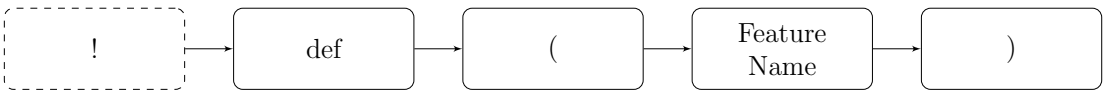

Figure 4.6: Abstract Schema of Feature Definitions in Model Files

https://doi.org/10.24355/dbbs.084-201812111412-0

Figure 4.6 shows an abstract schema of how features are represented in constraints of *.model files. The syntax of *.model files supports possibilities to express that a feature needs be be selected as well as to express that a feature needs to be deselected. To express that a feature is deselected the feature is negated. A negation of feature is represented by a bang (!) symbol at the start of the feature definition. Therefore, a feature definition can start with an optional bang symbol, if the feature is negated. After the optional bang symbol, follows the mandatory keyword *def* and an left parenthesis, to open the feature definition. In the centre of a feature definition stands the feature name. This can be the simple feature name, such as those listed by # *item* keywords or feature names extended by the *_MODULES* keyword. Furthermore assignments to the feature, such as shown in Line 5 of Listing 4.3, can appear in the feature definition as well. The end of a feature definition is indicated by a right parenthesis.

4.4.4.1 Integration into FeatureIDE Structure

After understanding the model file syntax, an appropriate import and export functionality can be integrated into FeatureIDE. As already described in Section 4.1, FeatureIDE supports extensions of third party developers. Newly implemented functionality can be easily integrated into the plugin based architecture of FeatureIDE. First step of doing so is, to find the appropriate place to integrate the extension into the existing architecture. To do so, we analysed the already existing import and export functionalities. Based on the analysis results we developed the integration schema for our import and export functionality shown in Figure 4.7.

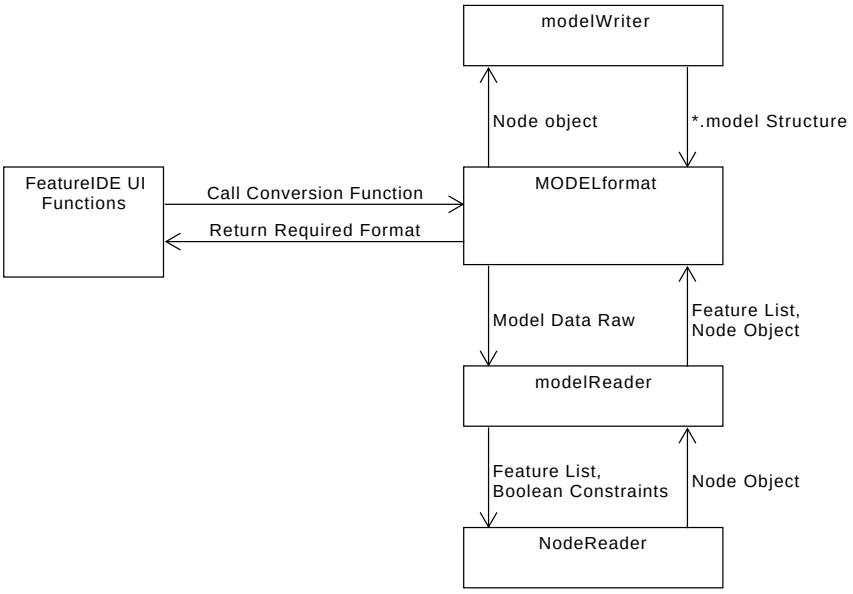

Figure 4.7: Overview Schema: Conversion of *.model Files

As shown in Figure 4.7 we split the new implemented functionality in three classes. The basis is build by a format class, which defines special notations and syntax features of the external file format. Beside the format class, a reader and a writer class is implemented. The reader class handles the conversion of the external file

https://doi.org/10.24355/dbbs.084-201812111412-0

format into the internal FeatureIDE XML structure (import). Conversions from FeatureIDE XML structure to an external format is handled by the writer class (export). The format class acts as an connector between reader, writer and other FeatureIDE modules. For the sake brevity Figure 4.7 abstracts FeatureIDE modules, which are not directly involved with the import and export, as a black box.

Import of Model Files

The modelReader module takes a *.model as input. This file is read as multi line string and is converted into a list of features and a list of boolean constraints. To do so, the input data is read line by line. Each line is analysed, whether it starts with the keyword *# item*. If this is not the case, the line is considered a constraint and saved into a global list of constraints. Otherwise, the keyword is cut off and the remaining string, without leading and trailing spaces, is saved into a global list of feature names. By using this method of collecting feature names, all features used in the *.model file can be derived. However, the feature nature (boolean, tristate, assignment) can only be derived by analysing constraints of the *.model file, as stated in Section 4.4.4. So, after generating the global constraint list, it needs to be analysed as well. For each constraint a regular expression is used to separate the feature names from other syntax elements. Thereafter, every feature name is added to the global list of feature names. To avoid duplicates, only feature names not already contained in the list are added.

After every item of the global constraint list is analysed, a complete list of feature names and a complete list of constraints are generated. Now the real conversion of the input data can be done, by using an integrated module of FeatureIDE, the so called NodeReader. NodeReader builds a FeatureIDE data structure called Node, from the list of feature names and the list of constraints. Before the FeatureIDE node structure can be build, specifics of the constraint syntax must be announced to the NodeReader. Therefore, we extended the NodeReader module so that the syntax of *.model files can be processed. Based on the provided syntax elements, NodeReader splits a boolean constraint into its subconstraints. Each sub constraint is recursively split further into subconstraints, until no further splitting is possible. This is the case, if a constraint only consists of two feature names connected by a boolean operator. To convert a complex and long constraint into a node can be time consuming. Therefore, we need to estimate the scalability of this method.

The generated Node is returned to the modelReader and further propagated to the MODELformat module, where it is used to generate the FeatureIDE XML structure for representing feature models.

https://doi.org/10.24355/dbbs.084-201812111412-0

Algorithm 2. Node to FeatureIDE Feature Model Structure

```
1   begin function addNodeToFM(IFeatureModel fm, Node n)
2     IFeature root ← getFeature()
3     fm ← setRootFeature(fm, root)
4
5     List<String> features ← getFeatures(n)
6     List<Node> constraints ← getConstraints(n)
7
8     for(features has feature)
9       fm ← addFeature(fm, feature)
10    end for
11
12    for(constraints has clause)
13      fm ← createConstraint(fm, clause)
14    end for
15  end function
```

Listing 4.4: Linux Conversion: Transform Node to Feature Model Structure

Algorithm 2 shows the algorithm used to create a FeatureIDE feature model structure from a node object. As input parameters an empty feature model (fm) and a node object (n) are expected. As first step of the algorithm, an artificial root feature is created and added to the empty feature model (Lines 2 and 3). Thereafter, a list of feature names (Line 5) and a list of constraints (Line 6) are extracted from the Node object. Next, each feature name contained in the list of feature names (features) is added to the provided feature model (Lines 8 to 10). Last step of Algorithm 2 is to add all constraints from the constraint list to the feature model. After function addNodeToFM is executed a feature model in FeatureIDE format is generated, which represents the variability model described in the initially provided *.model file. The converted model is returned to the requesting FeatureIDE module for further analysis or representation to the user.

Export Work Flow

For the conversion from a FeatureIDE XML structure to a *.model file the modelWriter class is used. This class takes a FeatureIDE Node object as input. To create a node object from feature models in FeatureIDE representation the MODELFormat class is used. A node object of FeatureIDE is powerful data structure, which contains all information from the original model. Therefore only this data structure us needed to generate a *.model file.

https://doi.org/10.24355/dbbs.084-201812111412-0

Algorithm 3. Feature Model to *.Model File

```
1   begin function generateModelStruct(Node n)
2
3   List<String> simpleFeatureNames
4   List<String> strConstraints
5   List<String> all_features ← getFeatures(n)
6
7   for(features has feature)
8     pattern ← "_MODULES | ← \w*"
9     if(not pattern matches features)
10      simpleFeatureNames ← add("# item" + feature)
11    end if
12  end for
13
14  List<Node> constraints ← getConstraints(n)
15
16  for(constraints has clause)
17    String strClause ← getStr(clause)
18    List literals ← getLiterals(clause)
19    for(literals has literal)
20      String replacement ← "def(" + asString(literal) + ")"
21      String replacementNeg ← "!def(" + asString(literal) + ")"
22      if(literal is negative)
23        String strClause ← replace(strClause, replacementNeg)
24      end if
25      if(literal is positive)
26        String strClause ← replace(strClause, replacement)
27      end if
28    end for
29  end for
30  List<String> modelSyntax ← simpleFeatureNames
31  modelSyntax ← strConstraints
32
33  return modelSyntax
34  end function
```

Listing 4.5: Linux Conversion: Convert Feature Model to *.Model File

As the first step, a global list of features is extracted from the Node object (Line 5). The list of feature names can contain boolean features as well as tristate and assignment features. Tristate and assignment features are modelled by adding additional variables to the feature model as described in Section 4.4.4. However, in *.model syntax the feature nature is only expressed in the constraints. Therefore, the list of features needs to be filtered (Lines 7 to 12). A pattern matching with regular expressions is used to sort out non simple feature names from the list of features. To match the original *.model syntax, the prefix *# item* is added to each remaining feature name.

https://doi.org/10.24355/dbbs.084-201812111412-0

After parsing the list of features, a list of constraints is extracted from the Node object (Line 14). For each constraint contained in the list, a string representation is created (Line 17). The logical operators to connect feature names already conform to those used in the *.modle syntax. Hence, no further conversion needs to be done for them. However the feature names contained in the constraints do not conform to the model file syntax. Therefore, each feature name needs to be extended by the syntax definition of features shown in Figure 4.6 (Lines 16 to 29).

First step to covert the feature names to the right syntax, is to extract all literal objects of a constraint (Line 18). A literal object A Literal object is an internal FeatureIDE data structure which represents a variable. For each literal its state (selected, deselected) is checked. Depending on the literals state, the respective feature will be replaced with the positive or negative replacement string shown in the lines 20 and 21 of Algorithm 3.

By doing the previous steps, two lists of strings are created. Both lists are merged into one combined list of strings, which is returned to the MODELformat module. This module propagates the string list to the requesting FeatureIDE module, to do further processing or to write the *.model file to the filesystem.

4.4.5 Partial use of Linux Model

As stated above, the methods for converting *.dimacs and *.model files into FeatureIDE feature models (XML format) suffer under possible scalability issues. If those scalability issues occur during the conversion, we are not able to evaluate the sample stability on Linux variability models at all. To prevent this scenario, we implement a tool, which builds a partial model of the Linux feature model created by FeatureIDE's dimacs conversion.

The basic concept behind our tool is to build a new feature model, which contains a processable number of features and constraints from the complete Linux feature model. The selections of features follows a random procedure. By doing so, we prevent influences to the new feature model, based on our own assumptions. However, by simply selecting random features from the original feature model there is no guarantee that any of the original constraints can be build. Therefore, we decided to start by collecting constraints randomly instead of features. The feature list for the feature model is than build by using all features contained in the selected constraints. Algorithm 4 shows the conceptual procedure of building a feature model, based on a portion of the original Linux model.

https://doi.org/10.24355/dbbs.084-201812111412-0

Algorithm 4.

```
1  begin function generatePartialFeatureModel(List<FeatureModel>
       featureModels, double percentToUse)
2    List<featureModel> newModels
3    int maxFeatureNumber
4    List<Feature> features
5    List<Constraint> constraints
6
7    FeatureModel randomFeatureModel ← selectRandomFeatureModel(
       featureModels)
8    maxFeatureNumber ← getMaxFeatureNumber(randomFeatureModel,
       percentToUse)
9    selectRandomFeatures(randomFeatureModel)
10
11   for(featureModels has featureModel)
12     constraints ← clear(constraints)
13     features ← getStillValidFeature(featureModel, features)
14     selectRandomFeatures(featureModel)
15     completeConstraints(featureModel)
16     FeatureModel fm ← createFeatureModel(features, constraints)
17     newModels ← add(fm)
18   end for
19   return newModels
20 end function
21
22 begin function selectRandomFeatures(FeatureModel featureModel)
23   List<Constraint> originalConstraints ← getConstrants(featureModel)
24   while(sizeOf(features) ≤ maxFeatureNumber)
25   do
26     Constraint const ← getRandomConstraint(originalConstraints)
27     constraints ← add(const)
28     originalConstraints ← remove(const)
29     features ← add(getFeatures(const))
30   end while
31 end function
```

Listing 4.6: Linux Conversion: Partial Feature Model Generation

Our procedure to calculate partial feature models for a product-line history of Linux kernel, starts by calling the function generatePartialFeatureModel. This function takes a list of feature models (product-line history) and the percentage of how many features should be used from the original models, as input value. From the provided list of feature models one feature model is selected randomly (Line 7). Based on the randomly selected feature model and the provided percentage value (Line 8), the maximum number of feature contained in the later generated feature models are defined. Furthermore, we use this randomly selected feature model to define a basic set of features used to create later feature models for the Linux product-line history. By using a randomly selected feature model as base for generating later

https://doi.org/10.24355/dbbs.084-201812111412-0

feature models, we keep this process free from our assumptions about the possible evolution of the product line.

The randomly selected feature model is used to select the first set of features and constraints (Line 9). This procedure is represented by function selectRandomFeatures (Line 22 to Line 31). The function takes a feature model as input. All constraints contained in this feature model are extracted and stored as list (Line 23). Based on the extracted constraints, a list of features is generated. To do so, a random constraint from the list of constraints is selected and added to the global list of constraints (Lines 26 and 27). Additionally the constraint is removed from the extracted list of constraints to prevent, that this constraint is selected a second time (Line 28). The contained features of the selected constraint are extracted and added to a global list of features (Line 29). Only feature not already contained in the global list of features are added, to prevent duplicates. The procedures of selecting a random constraint, extracting features, and adding them to a global list of features, continues until the global list of features contains more than the maximum number of features (Line 24).

After filling the global feature list with initial features, the actual generation of a partial Linux feature models starts (Lines 11 to 18). For each feature model contained in the initial provided list of feature models, a smaller feature model is created. First step in doing so, is to remove previously found constraints from the global list of constraints (Line 12). Thereafter, the previously found features are examined, whether all feature are still valid for the current feature model (Line 13). A feature is valid for the current feature model, if the feature model contains this feature. All invalid features are removed from the global list of features. To refill with new features, the selectRandomFeatures method is called (Line 14). As previously described, this method selects randomly features from a feature model, until the maximum number of features is exceeded. After filling the global collection of features, all constraints of the current feature model, which can be build with the features from the global feature list, are added to the global list of constraints (Line 15). Based on the collected features and constraints a new feature model is created (Line 16). It is added to a list of new generated feature models and the process restarts with the next feature model from the of original Linux feature models. The process continues until all feature models of the provided product-line history are processed. Thereafter, the list of new generated feature models is provided.

4.5 Summary

In this chapter, we present the implementation of different tools for this master thesis. As introduction into the chapter, we described the software product-line analysis framework (FeatureIDE), on which our implementations are based on(Section 4.1). During the description of FeatureIDE, we especially focus on the possibilities to extend the framework with new functionality. Furthermore, we describe how FeatureIDE functionality can be reused in third party developments, through the FeatureIDE library. By introducing FeatureIDE, we transmit a basic understanding on how reused functionality correlates with the complete frame work.

After introducing FeatureIDE, we discuss the main tool of this master thesis, the stability analyser. This tool implements our stability metrics described in Chapter 3.

https://doi.org/10.24355/dbbs.084-201812111412-0

The current version of the stability analyser is implemented as stand alone Java application. Even though the prototype is implemented as stand alone application, we considered future reuse of our implementation artefacts in the software design. For example, we defined a stability metric interface to promote maintainability and encourage third party developers to create new metric implementations.

By implementing the stability metric calculations we are able to analyse product-line samples calculated by established sampling algorithms. However, we are still missing a base line to compare those results against. To change this, we implemented the preservative sampling algorithm, developed in Section 3.6. The first prototype of our implementation is a stand alone Java command line application. Using the sampling algorithm as stand alone command line tool, provides the possibility to easily automate the algorithms execution by using a simple bash script.

After implementing the preservative sampling algorithm, the last thing missing, is a large scale product line with a long history to be analysed. The Linux kernel fulfils those criteria. However, as previously stated the variability model of Linux is described in a special description language, which is not supported by established sampling algorithms. Hence, we implemented a conversion system our selves. Section 4.4 describes the details of our implementation. To convert the Linux variability model into a processable format we implemented an automation script to convert variability models in KConfig to boolean formula in dimacs and model format. This conversion is based on KConfigReader. Furthermore, we extended FeatureIDE by a new import and export function for *.model files. By using the implemented import, a provided *.model file is converted to the XML format of FeatureIDE. This XML format is processable by the sample algorithms we use for this master thesis.

The implementations described in this chapter, build the basis of the evaluation process of this master thesis. During our evaluation we will analyse different product lines. Among others the Linux kernel is one of them. Processable feature models for the Linux kernel will be generated by using the KConfig conversion workflow presented in Section 4.4. To analyse the stability of product samples we use the preservative sampling algorithm as base line. To analyse the stability of produced samples, our metric implementations described in Section 4.2 will be used. The evaluation is described in Chapter 5.

https://doi.org/10.24355/dbbs.084-201812111412-0

https://doi.org/10.24355/dbbs.084-201812111412-0

5. Evaluation of Stable Product-Line Sampling

In this chapter, we evaluate the sample stability of different sampling algorithm by applying them to product-line evolution histories. To measure the stability between samples, we use our self developed metrics. Section 5.1 describes our experiment set up. The description includes a short introduction of our research question. Thereafter, the process to generate all data, required for the evaluation is explained. This process includes three steps from collecting product-evolution histories, over generating samples for the product-line evolution, to measuring the sample stability of calculated samples. In Section 5.1, we also define which concrete algorithms and sample stability measures are used for the evaluation. In Section 5.2, we describe the product line evolutions used for our experiment. In this context, we explain how the size of features and constraints grows for each product line over the course of its evolution. Thereafter, we present the results of our evaluation, in Section 5.3. We analyse measured sampling and testing efficiency for the sampling algorithms included in our experiment suit. Furthermore, an analysis of sample stability for those algorithms is conducted. We provide our analysed data in our Git repository used as data repository for this master thesis.[1] In Section 5.4, we discuss the evaluation results in context of our research questions. The last section, Section 5.5, discusses threats to the validity of our results.

5.1 Experiment Setup

In the following section we describe the set-up of our evaluation system for sample stability. Beside introducing our research questions, we focus on the evaluation procedure from selecting appropriate product-line histories to the calculation of sample stability measures. In this regard, we present our selected product lines, chosen sampling algorithms, and metrics used to generate the evaluation data. Furthermore, we describe the execution environment used for generating the necessary data.

[1] https://github.com/PettTo/Master_Thesis_Data

https://doi.org/10.24355/dbbs.084-201812111412-0

5.1.1 Research Questions

As introduced in Chapter 1, we aim with this master thesis to evaluate the stability of different sampling algorithms. Therefore, we implemented our sample stability metrics (Chapter 3) and a sampling algorithm, called Preservative Sampling (Section 3.6). Which also need to be evaluated. In context of performing those three evaluations, we want to answer the following research questions:

Research Question 1: How stable are samples created by different sampling Algorithms? We evaluate the stability of samples created by different sampling algorithms. To do so, the sample stability is measured for different product-line evolutions. Based on the measuring results, we want to raise a conclusion on which sampling algorithms produce stable samples and which do not.

Research Question 2: How relevant are our self developed metrics, when applied to real world product lines? We generate samples for the evolution of real world product lines. Thereafter, the stability of those samples are measured with the three metrics: *Ratio of Identical Configurations*, *Mean Similarity of Configurations*, and *Filter Identical Match Different Configurations*. Based on their measurement results, our metrics are compared against each other. By doing so, we want to find a conclusion, which metric can be used to measure sample stability of product-line evolutions used in industry.

Research Question 3: How does a self developed sampling algorithm, to maximize sample stability, perform compared to established sampling algorithms? We compare results of our self developed sampling algorithm with those produced by IncLing. Therefore, we measure the runtime and testing efficiency while executing the algorithms. In addition, we compare the stability of samples generated by both algorithms. Aim of this research question, is to examine whether our self developed sampling algorithm fulfils it's defined aim or not.

5.1.2 Procedure

Answering Research Question 1, requires measuring the sample stability between samples of a product-line evolution. During the process of measuring sample stability different artefacts are generated. Based on this artefacts we can answer Research Question 2 and Research Question 3. Hence, no special data calculations are needed to evaluate Research Question 2 and Research Question 3.

To calculate the sample stability for different product-line histories, we follow a conceptual chain of data processing. This chain consists of three different phases. Each phase produces data artefacts, which are used by the respective following phase. Figure 5.1 visualises the phases and their artefacts.

Phase 1: Collect Product Line Versions

A product-line evolution history represents the basic data for further calculations. Hence, the first phase of the processing chain is to collect different versions for a specific product-line. Each version of the product-line is represented by a feature model. Consequently, the product-line history is represented by an ordered list of

https://doi.org/10.24355/dbbs.084-201812111412-0

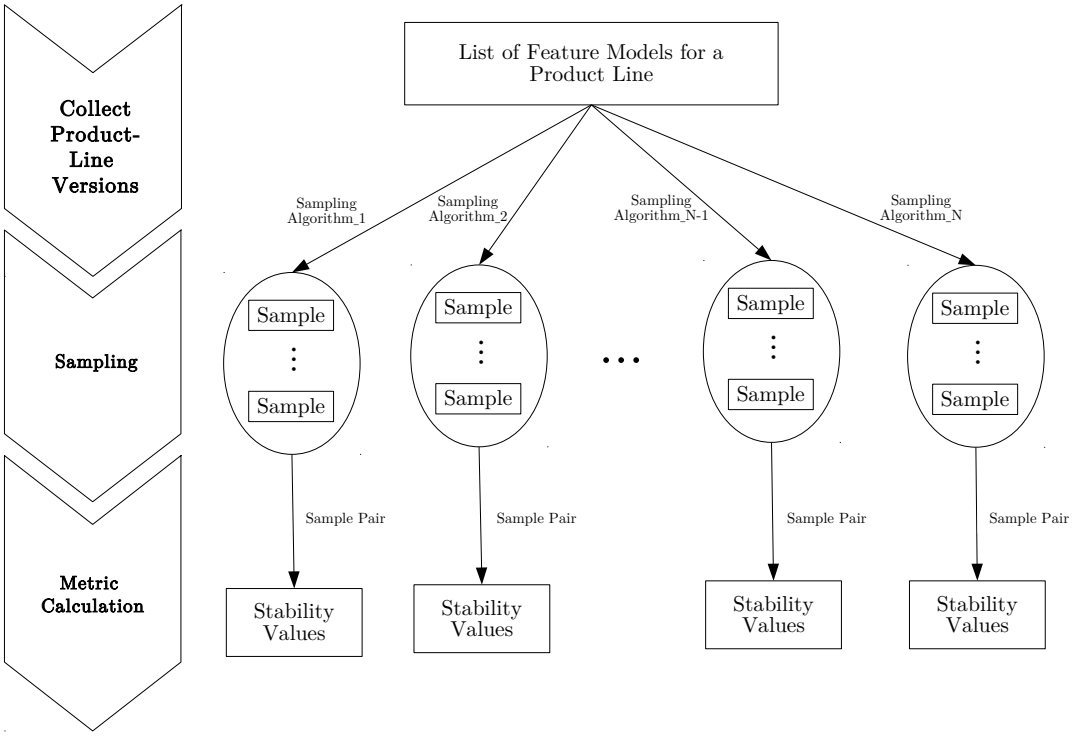

Figure 5.1: Overview Schema: Evaluation Procedure

feature models. To represent the historical evolution of the product line, it must be identifiable which version is represented by a feature model. This can be done by adding a version number or a concrete time stamp to the name of the feature model. Figure 5.1 visualises the data processing chain for analysing one product line. in our evaluation of sample stability, the processing chain is executed multiple times for different product-lines.

To generate relevant data for answering Research Question 1, we use different product lines in our evaluations. Our selection strategy is focused on large scale product lines from an industrial context. To use those product lines promises a realistic base for our evaluation. Based on our selection criteria, we decide to use Automotive02, Financial Services, and Linux kernel as representatives for product lines used in real world applications. Furthermore, we analyse the sample stability for our running example the Graph Library.

According to our defined chain of processing, we need to collect the product-line history for all of the chosen product lines. In respect to Automotive02 and Financial Services. Both product lines and their respective histories are provided as example feature modelling projects in FeatureIDE. Hence, all data needed is provided in the correct format.

FeatureIDE also provides three versions of the Graph Library as project examples. An evaluation on three product-line evolution containing only three evolution steps, does not hold enough significance for this master thesis. Hence, we developed a product-line history by executing different change operations on the first version of

https://doi.org/10.24355/dbbs.084-201812111412-0

Graph Library. For the selection of change operations we considered the typical changes in a product-line evolution, described by Passos et al.[PGT+13]. We use FeatureIDE as tool to create the feature models for the Graph Library history. Therefore, the feature models are in the correct format for further processing.

As for the Linux kernel product line, the product-line history can be acquired via the Linux kernel Git repository.[2] The repository provides the complete history of Linux. However, as described in Section 4.4, we need to convert the variability model of Linux into a processable format before using the models in further processing.

Phase 2: Sampling

The second phase of the processing chain shown in Figure 5.1, deals with creating samples for feature models acquired in phase one. For each feature model of a product-line history, a sample is created. Consequently, this phase results to a list of samples for each sampling algorithm in our evaluation suit. The created lists of samples represent a sample history, on which the sampling stability can be measured.

As initially stated, the aim of this master thesis is to evaluate the stability of sampling algorithms such as Casa, Chvatal, ICPL, and IncLing. Furthermore, we developed our own sampling algorithm (Section 3.6). As base line for our evaluation of sample stability, we use the random sampling procedure implemented in FeatureIDE.

By analysing the possible usage of sampling algorithms such as Casa, Chvatal, ICPL, and IncLing we discovered, that all algorithms could be used for small product lines. However, as evaluated by Al-Hajjaji et al. [AHKT+16], the Casa sampling algorithm shows considerably poorer runtime efficiency compared to the other algorithms. Furthermore, their results show that the Casa algorithm, does not finish sampling within 24 hours for product lines with more than 500 features. Because we aim to use large product lines (Automotive02, Financial Services, Linux) and do not have the time resources to sample a product line, for more than 24 hours, we decided to exclude Casa sampling algorithm from our evaluations. The evaluations done by Al-Hajjaji et al. [AHKT+16], show appropriate runtime efficiency for the algorithms Chvatal, ICPL and IncLing, even for medium and large feature models. Hence, those sample algorithms will be used to create samples for our evaluations. All three algorithms can create samples which fulfil different coverage criteria. The smallest coverage criteria all of them fulfil Pairwise coverage. Therefore, we use them to produce samples with pairwise coverage.

The sampling algorithms Chvatal, ICPL, and IncLing, as well as the random sample procedure are already implemented in FeatureIDE. Originally, we wanted to implement the algorithms as command line tools, by reusing the implementation of FeatureIDE library. By doing so, we aimed to automate multiple sampling executions and respective logging of runtime and testing efficiency statistics for each sampling algorithm. However, we discovered that ICPL and Chvatal are not directly contained in the FeatureIDE library. Therefore, we could not implement a headless version of those two algorithms. Instead we adjusted FeatureIDE itself, so that multiple executions of sampling and logging of statistic values is possible. The

[2] https://github.com/torvalds/linux

https://doi.org/10.24355/dbbs.084-201812111412-0

implemented adjustments are utility specifically tailored for this master thesis and not of general interest. Hence, they will not be integrated into the repository of FeatureIDE. We use the adjusted version of FeatureIDE to execute Chvatal, ICPL, and the random sample procedure. IncLing and our self implemented preservative sampling are executed as Java command line applications. We automate the execution of those tools by using simple bash scripts.

During the execution of the chosen sampling procedures, statistics about runtime and testing efficiency are logged away as *.csv files. We use this statistics for our evaluation of Research Question 3. To mitigate influences of the operating system scheduler and the Java garbage collector we execute each sampling procedure five times and form the mean value between the resulting statistics.

A sampling execution, finishes by itself if a distinct number of features (Pairwise coverage or maximum number of features) is reached. To prevent an algorithm from consuming to much calculation time, we defined a time out to finish its execution forcefully. We define a calculation time of 8 hours as time out for the sampling procedures. That means, algorithms which calculate configurations incrementally and buffer them in an internal queue, can write these configurations to the file system before they are terminated. To implement a time out after 8 hours is necessary, to raise significant results in the limited scope of this master thesis.

Phase 3: Metric Calculation

The third phase of our data processing chain consists of measuring the sample stability for a product-line history. To do so, we use the stability calculation system implemented in Section 4.2. The calculation system, implements the stability metrics *Ratio of Identical Configurations*, *Mean Similarity of Configurations*, and *Filter Identical Match Different Configurations*. We developed all three metrics ourselves in Chapter 3. Each metric is designed to calculate the stability between a pair of samples. To apply the stability metrics to a list of samples, we build a sample pair two consecutive samples contained in the list. This is visualised in Figure 5.2.

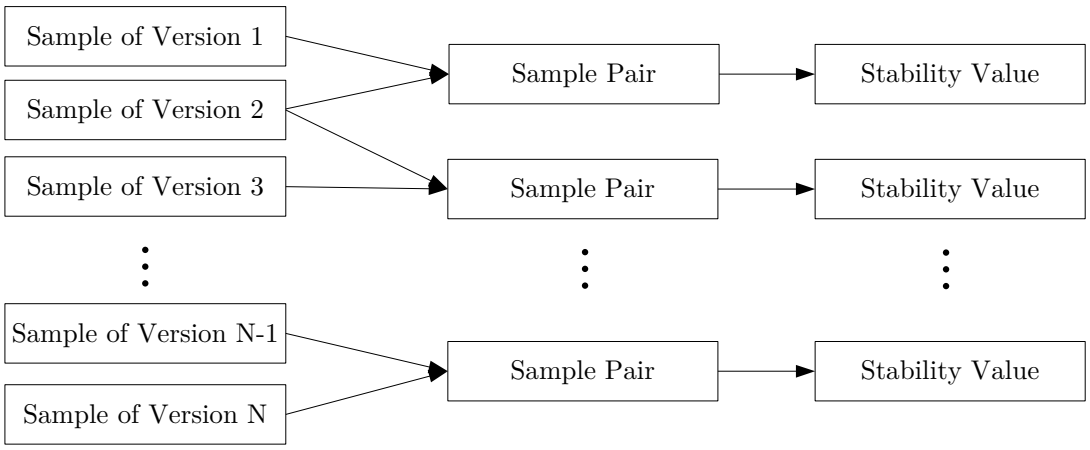

Figure 5.2: Overview Schema: Concept of Pairwise Calculation of Sample Stability

As described above, our metrics are designed to calculate the sample stability between a pair of samples. Therefore, the calculation system expects a pair of samples

https://doi.org/10.24355/dbbs.084-201812111412-0

as input values. We build those pairs following the schema shown in Figure 5.2. As for the calculation system, it is implemented as Java command line application. This way, we can use a bash script to automate the metric calculation.

After executing the calculation for a list of samples, the stability values calculated by the *Ratio of Identical Configurations*, *Mean Similarity of Configurations*, and *Filter Identical Match Different Configurations* metric is created. For each metric a list of stability values, which represent the sample stability between consecutive product-line versions, for a specific sampling algorithm, is logged as *.csv file. The results of our metric calculations are used as base for evaluating Research Question 1. Furthermore, we can use the produced data to discuss Research Question 2.

5.1.3 Execution Environment

Our experiments are conducted on the server of the Institute of Software Engineering and Automotive Informatics. As operating system the server runs a 64 bit CentOS 7.3. The hardware of the server includes a CPU system consisting of 16 cores. Each core has a CPU stroke of about 2400 MHZ. Furthermore the system provides a main memory of 64 GByte. As described above our metrics and sampling procedures are executed on a virtual Java machine. The server provides an Oracle Java Development Kit version 1.8. We execute our metric calculations, the sampling algorithms as well as our conversion tools on this virtual machine with nine GByte of virtual memory. This is necessary, to process the large scale feature models used in our experiment.

5.2 Subject System

The following section describes the product-lines used in our evaluation. They are used as base to create samples, on which sample stability is measured. Beside providing basic information about each product-line, we also discuss how much the number of features and constraints changes during the product-line evolution. To do so the growth of feature and constraint size for the product-line versions, is visualized as bar chart. The bar chart contains the number of features and constraints for different versions of the product line. Each version of the example is presented with two bars. The lower bar represents the number of features and the upper bar represents the number of constraints contained in the respective version. We use the same bar chart structure for Graph Library, Automotive02, Financial Services, and Linux.

5.2.1 GPL

The first product-line we present is the Graph Library example. Graph Library is a small product line contained as feature modelling example in FeatureIDE. Our aim is to use this product line as example to validate our own developments (Metrics, Preservative Algorithm). However, the examples provided by FeatureIDE include only three different versions of the Graph Library product line. Furthermore, the provided versions differ largely between each other, so that they are not applicable as validation basis. Hence, we created our own evolution history based on Graph

https://doi.org/10.24355/dbbs.084-201812111412-0

Library. As described in Section 2.3, our product-line evolution of Graph Library contains our created versions between the smallest and the medium version of Graph Library provided by FeatureIDE. Figure 5.3 indicates the growth of features and constraints for our artificially build evolution history.

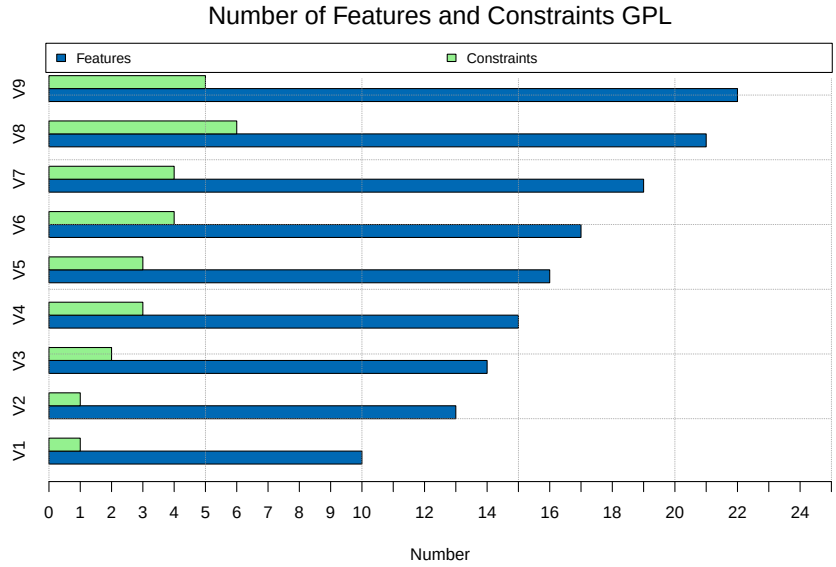

Figure 5.3: Bar Chart: Evolution of GPL

As already described in Section 2.3, we developed nine different version of Graph Library as validation examples. To promote the purpose of the Graph Library product-line history, we realised small steps between the versions. Furthermore, we realised a continuous growth in feature size, as shown in Figure 5.3. Additionally, we used only a small number of constraints, which are continuously growing until the eighth version is reached. In the evolution step from eight to ninth version we restructured the feature model, so that a cross tree constraint is expressed in the model structure. Therefore, the number of constraints, contained in the feature model is reduced.

5.2.2 Automotive02

Automotive02 is an example for a product line used in automotive industry. We can access this example, because it is contained in FeatureIDE as an feature modelling example. FeatureIDE provides a small product-line evolution history for Automotive, which includes four different versions. However, those versions are just named with version number and not with time stamps. Therefore, we can not estimate how much development time passed between one evolution step and another. Figure 5.4 displays how many features and constraints are contained in the respective feature models of Automotive02. Figure 5.4. By looking at the displayed feature numbers, we recognize that Automotive02 is a large scale product line. The minimal number of features amounts to about 14000 in the first version. This amount increases during the product-line evolution to over 18000 features. The increase in feature size is most prominent in the evolution step from version one to version two (about 3000 features). Between the second and third version an increase about 1000 features

https://doi.org/10.24355/dbbs.084-201812111412-0

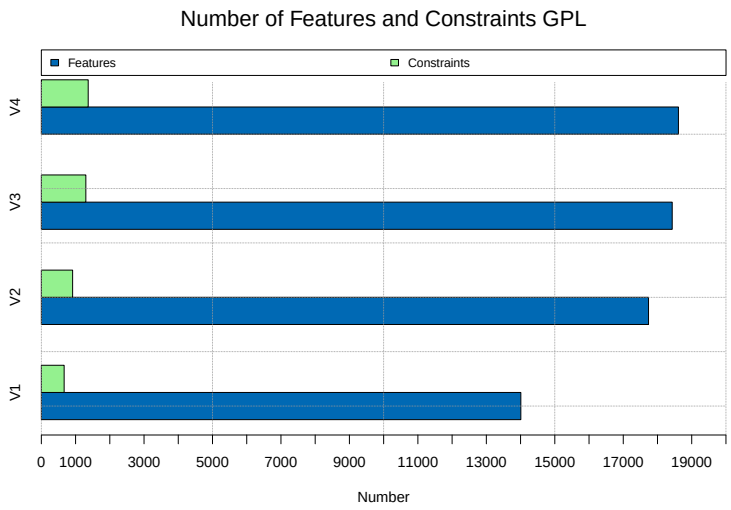

Figure 5.4: Bar Chart: Evolution of Automotive02

can be mentioned. In the final evolution step, the number of features increased only slightly (about 200 features).

Compared to the number of features, Automotive02 contains a lot less constraints. The maximum number amounts to slightly more than 1300 constraints. However, the pattern of evolution is similar to the one for features. This pattern shows the most increase in constraints for the evolution step one (Version 1 to Version 2) and two (Version two to Version three). Between the third and the fourth version of the product line, only a small increase in the number of constraints is to be mentioned.

As previously mentioned, the examples of FeatureIDE do not provide a time frame for the evolution steps of Automotive02. However, we can estimate a possible evolution pattern for the product line. Based on data presented before, we can estimate that the first evolution step (between version one and two) needed the largest time frame or the most effort. Furthermore, we can estimate, that more time passed (or more effort was spent) for the second evolution step compared to the third.

Even though Automotive02 provides only a small evolution history, it is a product line used in industrial context. Hence, we can use Automotive02 to produce relevant result with measuring the stability between its evolution steps. Beside the usage in industry, the size of Automotive02 is another reason we chose to use it in our evaluation. However, the size Automotive02 can lead to scalability problems for our used sampling algorithms. As evaluated by Al-Hajjaji et al.[AHKT+16], the Chvatal sampling algorithms needs more than 24 hours to finish for product lines with more than 5000 features. Additionally Al-Hajjaji et al. point out that other sampling algorithms such as ICPL and IncLing, also need long calculation times for larger product lines. As for Automotive02 we need our chosen algorithms to handle feature models with minimal 14000 features. Therefore, we defined the time out for sampling previously described in Section 5.1.

https://doi.org/10.24355/dbbs.084-201812111412-0

5.2.3 Financial Services

Financial Services is an industrial product line, from an financial application background. It is provided by FeatureIDE as feature modelling example. The example project of FeatureIDE contains ten different versions of the Financial Services product line. These versions are named with time stamps. Therefore, we can see how much time between each version has passed. The time span between two versions can be used as an indicator for possible changes in the feature models of the product line. Figure 5.5 visualises the product-line evolution of Financial Services, in context of the growth of features and constraints.

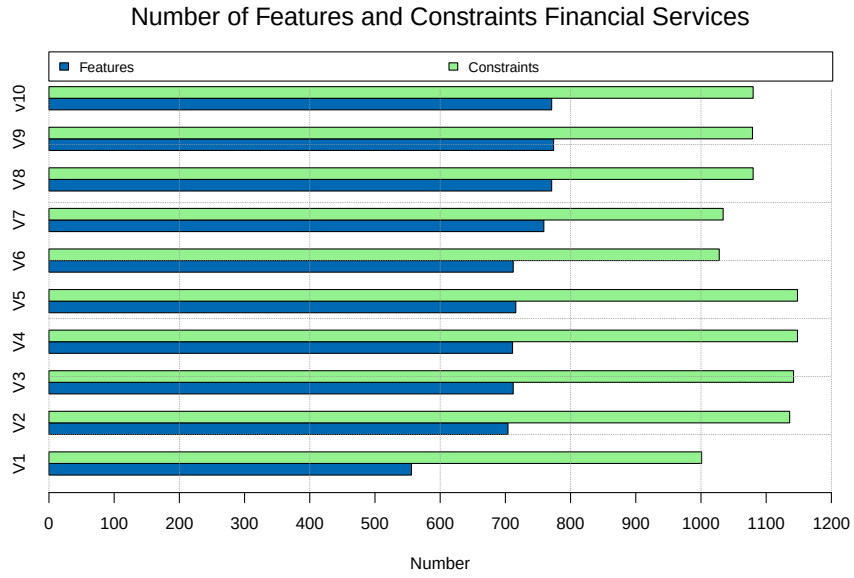

Figure 5.5: Bar Chart: Evolution of Financial Services

By looking at the data we can assess that most of the product-line versions are roughly a month apart of each other. Only the time span between version one and two is significantly longer with about four months. Additionally, we can note that all versions of Financial Services contain more constraints than features.

As for the growth of features we can assess a general increase in numbers. Only between version 5 and version 6, a small decrease can be noted. All things considered, the number of features grows from about 550 to slightly less than 800 features. The most significant increase in feature size can be noted between version one and two. This growth is in accordance, to the longer time span of the evolution step.

As for the constraints, we cannot note a continuous increase in size. The maximum number of constraints is reached at version five, with slightly less than 1200 constraints. However, in the evolution from version five to six this number decreases again. Even though the number of constraints increases from version six until version 10, the maximum number of constraints is never reached again. The strongest increase of constraint size is visible between version one and two. This growth is in accordance to the longer time span between those versions.

https://doi.org/10.24355/dbbs.084-201812111412-0

All things considered, Financial Services provides a solid base to generate relevant results for our evaluation of sample stability. One reason for this statement, is the large product-line history with nine evolution steps. Additionally, we know how many time passed between an evolution step. As for the constraints, we can find an interesting evolution pattern, where the number of constraints decreases between version five and six. This pattern indicates a restructuring process. Hence, we expect large changes between those versions, which could lead to reduced similarity between them. Besides, Financial Services is an example product line directly extracted from industry. Hence, we strengthen the relevance of our evaluation results, in context of real world applicability.

5.2.4 Linux as Product Line

In respect to Linux kernel, all product-line versions are open-source. So the variability models of all Linux version can be acquired by downloading them from the Linux kernel Git repository. [3] However, the format of Linux variability models cannot be processed directly by our implemented tool chain. In Section 4.4 we describe different methods to convert the Linux variability model into a processable feature model format. All variants rely on file formats produced by KConfigReader [Käs17] (*.dimacs and *.model).

The first step of creating the Linux product-line history is to download Linux variability models and process them with KConfigReader. To do so, Section 4.4.3 presents an automated workflow to download and convert Linux variability models with the KconfigReader. By using this workflow we created a history of Linux models in the time span from 06.11.2013 to 14.01.2018. Between this time span 420 commits changed the feature model Linux X86 architecture, so that we acquired 420 converted variability models of Linux. We provide those models on our Git repository for data used in this master thesis.[4]

As described in Section 4.4.2, we chose KConfigReader over Undertaker and LVAT as conversion tool for Linux variability models based on the analysis of El-Sharkawy et al. [ESKS15]. They stated that KConfigReader, produces more reliable results than the other tools. However, the conversion of KConfigReader does not represent all concepts of Linux correctly. Some corner cases such as, choices and prompt combinations, choices and if combinations, or choices and hierarchies are translated incorrect. For a complete list of incorrect translated constructs of Linux variability models, we refer to [ESKS15]. Based on the conversion output of KConfigReader (*.dimacs and *.model files), we defined two possible procedures to generate processable feature models in Section 4.4.

One procedure to convert the Linux variability model into a processable file format uses the FeatureIDE dimacs converter. As described in Section 4.4 this converter expects a *.dimacs file as input and creates a FeatureIDE feature model as XML file. By applying this method to the generated *.dimacs files, we can create FeatureIDE feature models that contain about 60,000 features. By trying to calculate samples for this feature models, we discovered that sampling algorithms such as ICPL, Chvatal

[3] https://github.com/torvalds/linux
[4] https://github.com/PettTo/Master_Thesis_Data

https://doi.org/10.24355/dbbs.084-201812111412-0

and IncLing do not scale for product lines with this many features. Hence, we cannot use FeatureIDE's dimacs import to generate processable feature models, for this master thesis.

As described in Section 4.4, the high number of features result from transforming the Kconfig file into *.dimacs format. To overcome this challenge, we developed an alternative conversion procedure (see Section 4.4) based on *.model files created by KconfigReader. However, the method itself shows scalability issues when applied to the original Linux variability model. The reason behind those issues are complex constraints contained in the *.model files. The conversion of those constraints, with our implementation is to time consuming, to be used in this master thesis. In our tests the conversion of *.model files did not finish in 16 hour. To loose more than 16 hours of calculation time, is not acceptable in context of the limited time for this master thesis. Hence, this conversion method is not usable to produce processable feature models, for this master thesis.

Based on the challenges described above, we decided to use the Linux variability model not in its full scale, but only a small part of it. To do so, we developed a procedure to select only a small number of features and constraints from the original feature model. The procedure is based on the feature models created by the dimacs conversion of FeatureIDE. Section 4.4 describes the implementation of our procedure. For this master thesis, we use this procedure to create ten partial feature models from the Linux history. We decided to restrict the number of feature models to ten, in consideration to the limited time frame of this master thesis. As base we chose the ten latest Linux versions in the available history. Thereby, we cover a time span of about three month from 15.11.2017 to 14.01.2018. Moreover, we decided to use only 1.5 percent of features provided by the original models. This results to about 900 features contained in the created feature models. Figure 5.6 visualises the constructed product-line evolution of our partially used Linux product line.

The bar chart shown in Figure 5.6 visualises the growth of constraints and features contained in our generated feature models of Linux history. Similar to the previous bar charts, Figure 5.6 visualises the number of features and constraints. In addition, it visualises the number of reused features, as bar between features and configurations. This bar is used as an indicator for changes between two versions of our generated feature models. As described in Section 4.4, we use a randomly chosen Linux feature model as base, before generating the first Linux model for our evolution history.

By looking at the data we can see, that all of our generated feature models contain more constraints than features. While the number of features (about 900) stays roughly the same during the whole product-line evolution, the number of constraints changes from version to version. The maximum number of constraints is reached at version six with about 5700 constraints. Thereafter, a decrease in the number of constraints can be noted. Based on the number of reused features, we can assess that for each version the majority of features from the previous version could be reused. Only about 100 features needed to be exchanged by our algorithm.

https://doi.org/10.24355/dbbs.084-201812111412-0

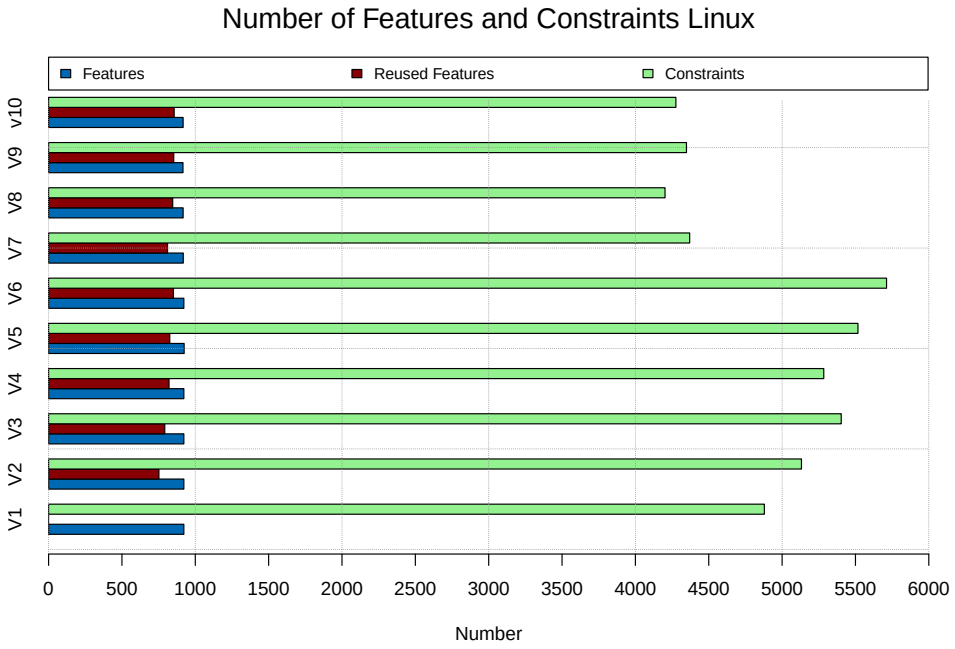

Figure 5.6: Bar Chart: Evolution of Linux

The consistent number of features in each version, can be explained by the procedure used to create the partial Linux feature models. As described in Section 4.4 the procedure defines a maximum number of features. Furthermore, the procedure tries to reuse as much features as possible from the previous version. This behaviour is indicated by our bar chart. That not all features are reused from version to version can indicate changes between the original Linux models. As for the evolution pattern seen for constraints, the decrease in size of constraints, could indicate a restructuring of the Linux variability model between version six and seven.

Based on the method used to create the partial Linux feature models, we cannot be sure, that the evolution patterns visualised by Figure 5.6 represent the real evolution of Linux.

5.3 Results

In this section we present the evaluation results of measuring sample stability for different sampling algorithms. We perform our assessments on evolutions of different product lines. Therefore, we present the sample stability for each product line separately. Before presenting the sample stabilities, we perform an assessment on sampling and testing efficiency for the sampling algorithms used. Focus of the assessment is set to our self developed sampling algorithm (preservative sampling). After presenting the results of our measuring, we discuss the previously defined research questions.

5.3.1 Sampling Efficiency

The sampling efficiency of a sampling algorithm describes its aggregated computation time to achieve pairwise coverage [VAHT+18]. We measure sampling efficiency

https://doi.org/10.24355/dbbs.084-201812111412-0

of different sampling algorithms to compare them with the Preservative Sampling. Doing so, we gather information to answer Research Question 3.

The results of measuring the sampling efficiency of our used product lines is visualised in the box plot shown in Figure 5.7. The box plot shows, results for the following product lines from left to right: GPL, Linux, Finincial Services, and Automotive02. Each product line, is sampled by the following sampling algorithms: Chavatal, ICPL, Random, IncLing, and Preservative Sampling, as shown on the x-axis of the box plot. The y-axis shows the amount of time needed to calculate the samples in minutes. This axis is scaled logarithmic to visualise short calculation times as well as long calculation times in the same diagram. As previously described, we implemented a time out at 8 hours of calculation time. This time out is represented in the box plot by the line at 480 minutes. A box of the box plot, represents the data distribution of over the whole product-line evolution.

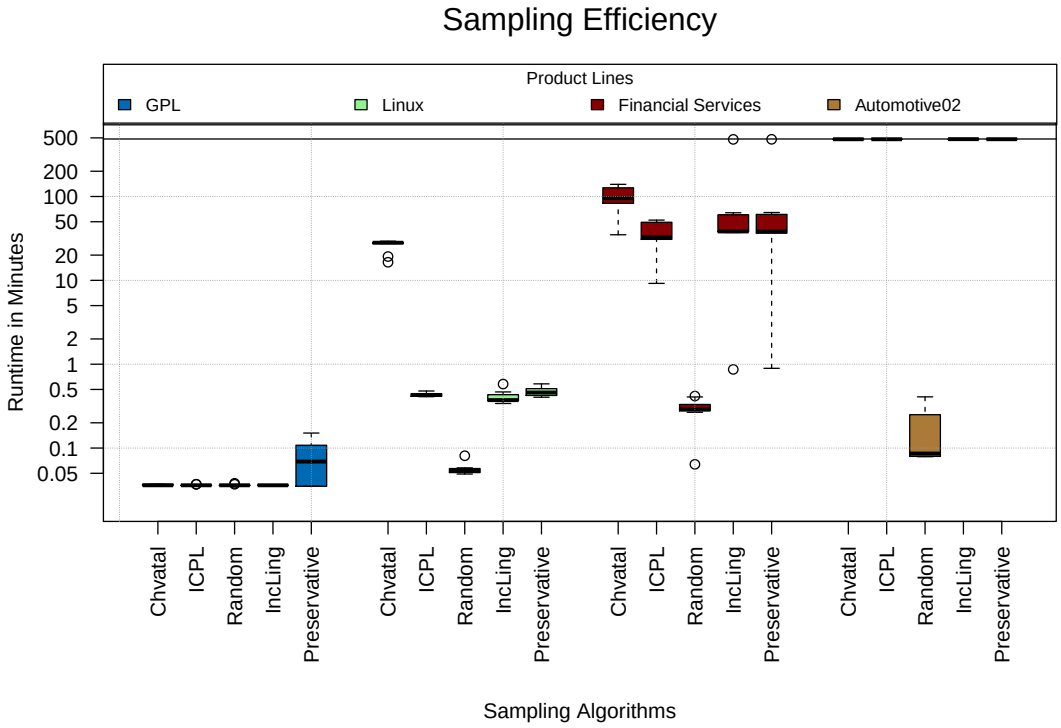

Figure 5.7: Box Plot: Results Sampling Efficiency

Looking at the data shown in Figure 5.7, we can observe that between all product lines, the sampling times for the Graph Library are the shortest and that the longest calculation times appear for Automotive02. In regard to Automotive02 our time out is reached by four of five sample procedures. Only the random procedure finishes in time (no pairwise coverage). As for random sampling, we can observe throughout all product lines, the shortest calculation times. In addition we can see, that the Chvatal algorithm always takes the longest time to cover pairwise coverage. With respect to our self developed algorithm (Preservative Sampling), we can asses, that it shows for most product lines the same or slightly higher sample times as IncLing. The only exception occurs for Financial Services, where the minimal sampling time is significantly lower as the minimal sampling time for IncLing.

https://doi.org/10.24355/dbbs.084-201812111412-0

With respect to the implementation of Preservative Sampling, faster calculation times as IncLing could be possible. However, this can only be the case if enough configurations of a previously calculated sample can be reused. Our experiment shows this is only the case for one version of Financial Services and for the Graph Library example. As for Financial Services, this visualises as the observed outlier at about one minute. In regard to Graph Library the profit of reusing configurations cannot be seen, because the needed preprocessing takes to long, compared to simply calculating a complete new sample with IncLing. Based on the same reason, we can observe a slightly higher calculation time of Preservative Sampling compared to IncLing for Linux.

The diagram shows, that Chvatal, ICPL, IncLing, and Preservative Sampling do not finish in 8 hours of sampling time. This behaviour meets our initially taken assumption, that those sampling algorithms do not scale for product lines with more than 14,000 features. Another observation seen in the box plot, is that random sampling shows faster calculation times than other algorithms. The reason behind this behaviour, is that random sampling does not try to fulfil pairwise coverage. Instead, random sampling creates configurations randomly, until a maximum number is reached. As maximum number of configurations we chose the configuration number created by IncLing after 8 hours of sampling. Furthermore, we observed, that Chvatal is the slowest of the algorithms. This is in accordance to the evaluation of Al-Hajjaji et al. [AHKT+16].

5.3.2 Testing Efficiency

As described by Al-Hajjaji et al.[VAHT+18], testing efficiency considers how many configurations a sampling algorithm generates to reach pairwise configuration coverage. We take this metric to compare our self developed Preservative Sampling against established sample algorithms such as Chvatal, ICPL, and Incling. By doing so, we gather insights to answer Research Question 3.

The box plot shown in Figure 5.8, shows the measured testing efficiency for Graph Library, Linux, Financial Services, and Automotive02. For each product line we measure the testing efficiency for the five sampling algorithms in our evaluation suit. These algorithms are visualised on the x-axis. On the y-axis the number of configurations is displayed. To display all results in the same box plot we use a logarithmic scaling for the y-axis. The boxes visualized in the box plot, represent the distribution of the number of configurations, for a sampling algorithm, over the whole product-line evolution.

By looking at the data provided in Figure 5.8, we can ascertain, that the used sampling algorithms generate the fewest number of configurations for Graph Library. The highest number of configurations is generated for Financial Services. Linux and Automotive02 build the mid range. For Linux and Financial Services we can ascertain the tendency of Incling and Preservative Sampling to produce more configurations than Chvatal and ICPL. As for Preservative Sampling, this tendency can also be seen in the Graph Library example. Except for the Graph Library product line, IncLing and Preservative Sampling produce about the same number of configurations for the product-line evolutions in our experiment. With respect

https://doi.org/10.24355/dbbs.084-201812111412-0

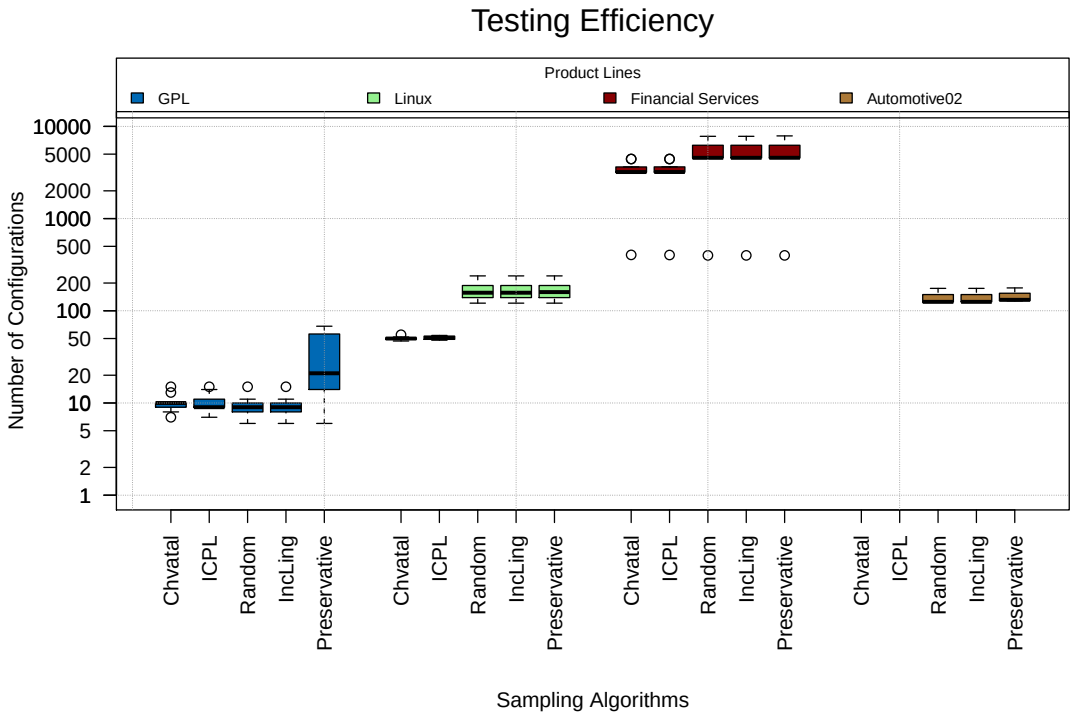

Figure 5.8: Box Plot: Results Testing Efficiency

to Automotive02, no configurations are generated by Chvatal and ICPL. Furthermore, the number of configurations created by IncLing and Preservative Sampling are only mid-range with about 150 configurations, even though the feature models for Automotive02 are the largest in our experiment suit.

The small number of configurations generated for Automtive02 can be explained by the size of this product line. The sampling algorithms Chvatal, ICPL, IncLing and Preservative Sampling, need more than 8 hours to calculate pairwise configuration coverage. Hence, they reach our defined time out before finishing the complete calculation. Thanks to the incremental nature of IncLing and Preservative Sampling, at least some configurations are created. Chvatal and ICPL only create samples if the whole sampling process is finished. Therefore, no data can be produced for both algorithms.

Based on the implementation of Preservative Sampling, we expect it to produce larger samples than IncLing throughout the product line history. However, this behaviour is only visible for the Graph Library example. For all other product lines, Preservative Sampling creates about the same number of configurations for each product-line version as IncLing. The reason behind this behaviour is, that the feature models of product lines used in industry change allot between each version. Therefore, only a few (or no) configurations can be reused. Hence, both algorithms show the same distribution of stability values. For Graph Library this is not the case, so Preservative Sampling creates a larger sample.

https://doi.org/10.24355/dbbs.084-201812111412-0

5.3.3 Measuring Sample Stability

To measure the sample stability of different algorithms, we use our metrics defined in Chapter 3. As previously describe, we conduct our stability analysis for the product lines Graph Library, Linux, Financial Services, and Automotive02. We use the data produced by our metrics as basis to answer Research Question 1. Furthermore, we can use the generated stability values as base to answer our Research Question 2.

To present the results of our measuring, we visualise the stability values as box plot, such as the one shown in Figure 5.9. A box plot contains results for our three metrics. They are visualized in the following order from left to right: *Ratio of Identical Configurations* (ROIC), *Mean Similarity of Configurations* (MSOC), and *Filter Identical Match Different Configurations* (FIMSC). For each metric the samples produced by Chvatal, ICPL, Random, IncLing, and Preservative Sampling are examined. The names for those algorithms are displayed on the x-axis of our box plots. On the y-axis, we display the stability value in percent. A box contained in the box plot represents the distribution of stability values throughout the whole product-line evolution. We provide the same box plot structure for the product lines: Graph Library, Linux, Financial Services, Automotive02.

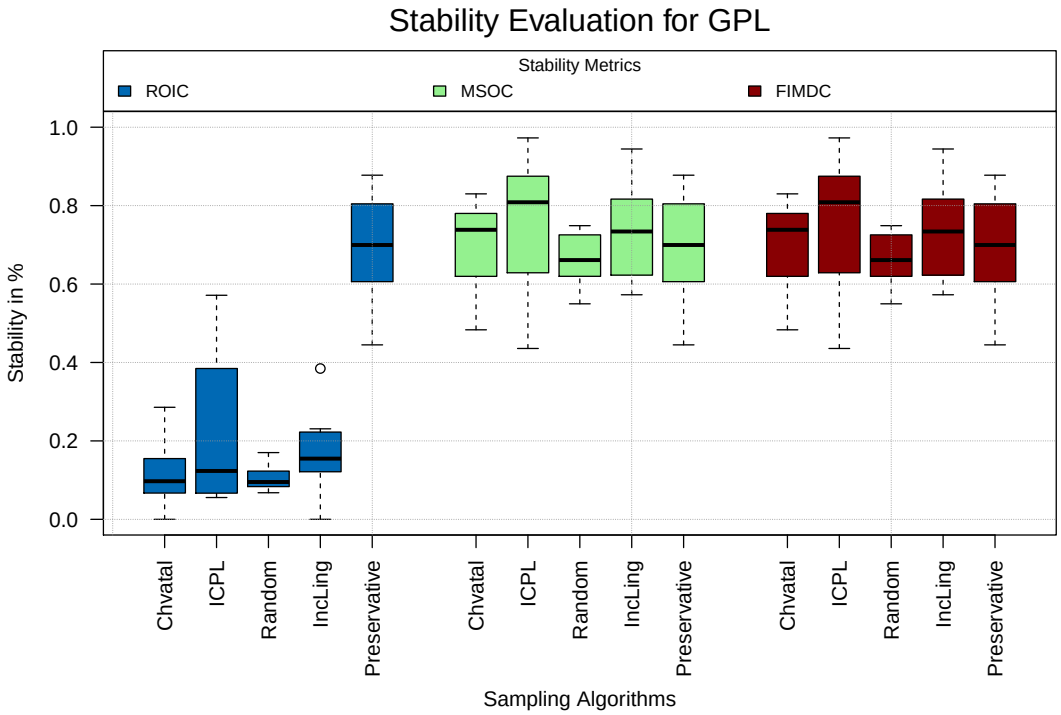

Figure 5.9: Box Plot: Results of Calculating Sample Stability for GPL

Figure 5.9 visualises the sample stability for the Graph Library product-line evolution. By looking at the data we can see, that for all sampling algorithms the *Ratio of Identical Configurations* generates lower stability values compared to the other two metrics. *Mean Similarity of Configurations* and *Filter Identical Match Different Configurations* produce for this example about the same sample stability values for all sampling algorithms. Between the sampling algorithms, seen for *Mean Similarity of Configurations* and *Filter Identical Match Different Configurations*, ICPL

https://doi.org/10.24355/dbbs.084-201812111412-0

produces samples with the highest stability values. However, those samples also provide a large spread between minimum and maximum stability. In case of *Ratio of Identical Configurations* samples with the highest stability values over the product line evolution are produced by the Preservative Sampling. Nonetheless, in regard to the other two metrics Preservative Sampling produces even less stable samples than IncLing. Seen over all metrics, the Random sampling procedure produces samples with the lowest stability.

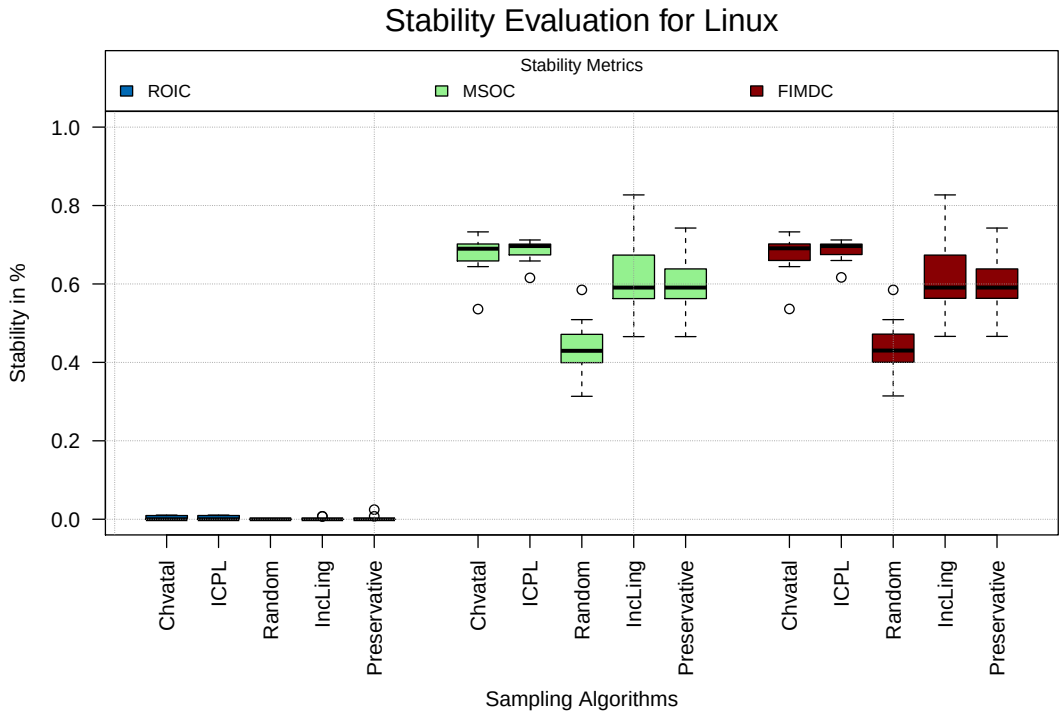

Figure 5.10: Box Plot: Results of Calculating Sample Stability for Linux

Figure 5.10 shows the results of our sample stability measure for samples of the Linux product line. First thing to note is, that *Ratio of Identical Configurations* produces only small sample stability values close to, or equal to zero. Furthermore, the distribution of sample stability for the other two metrics is equal for all sampling algorithms. As for *Mean Similarity of Configurations* and *Filter Identical Match Different Configurations* the Random sampling procedure creates samples which are the least stable throughout the product-line evolution. In regard to the mean stability over the product-line evolution, the best values are measured for Chvatal and ICPL, although Chvatal shows a higher distribution between all values. IncLing reaches the highest maximum sample stability when measured with *Mean Similarity of Configurations* and *Filter Identical Match Different Configurations*. As seen previously, Preservative Sampling reaches again lower results than IncLing, when measured with *Mean Similarity of Configurations* or *Filter Identical Match Different Configurations*.

The box plot used to visualise the result of measuring sample stability for Financial Services is shown in Figure 5.11. Similar to the results for Linux, *Ratio of Identical Configurations* produces in average only sample stabilities equal to zero. However,

https://doi.org/10.24355/dbbs.084-201812111412-0

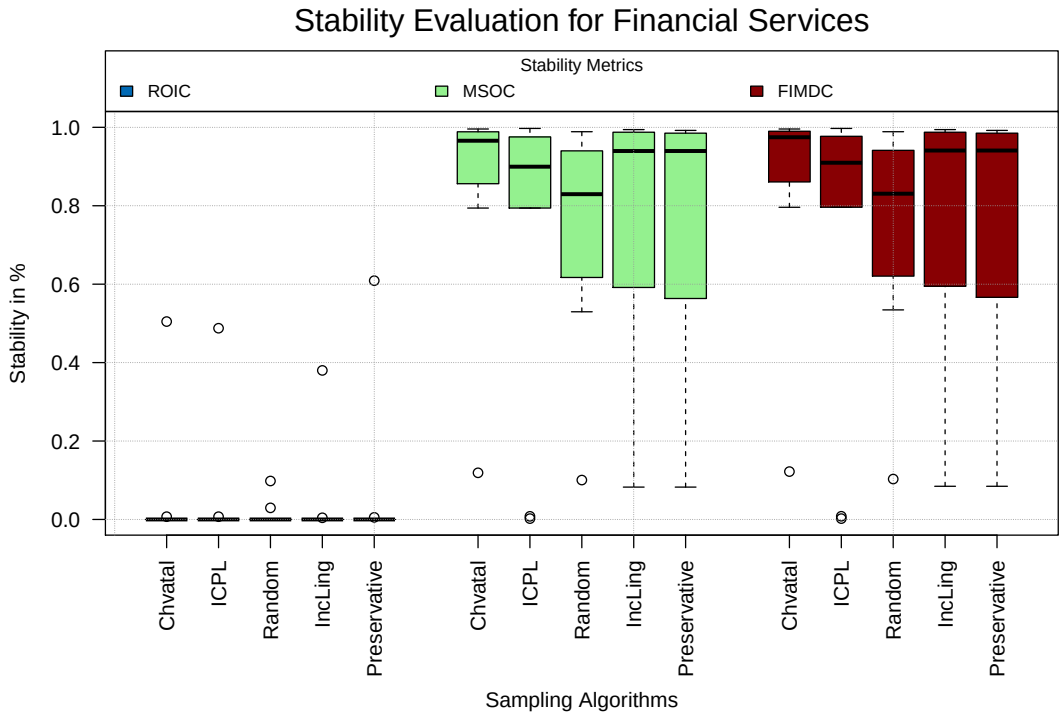

Figure 5.11: Box Plot: Results of Calculating Sample Stability for Financial Services

for Financial Services some outliers can be noted. For Preservative Sampling one outlier occurs for example at 60 percent stability. By viewing the results of these metrics, we note the same distribution of sample stability for all algorithms. Furthermore, we can observe that *Mean Similarity of Configurations* and *Filter Identical Match Different Configurations* follow the same distribution of sample stability values. Therefore, the following descriptions apply for both metrics. With respect to the data shown in Figure 5.11, we see that for all sampling algorithms a maximum stability score close to 100 percent are calculated. The highest average score is calculated for the samples generated by the Chvatal algorithm and the Random procedure. Furthermore, we can note, that the distribution of stability values of IncLing and Preservative Sampling are almost equal to each other. Moreover, the samples generated with those algorithms show largest data distribution for the Financial Services product-line evolution.

Figure 5.12 represents the sample stability values of samples calculated for Automotive02. As described previously, the sampling algorithms Chvatal and ICPL would have needed more than 8 hours to compute pairwise configuration coverage for Automotive02. Hence, our defined time out cancelled the sampling process before finishing. Therefore, no samples to calculate sample stability are available. IncLing and Preservative Sampling also reached our defined time out during calculation of samples. However, their incremental nature produced at least some configuration. So that, measuring sample stability was possible.

As shown in Section 5.2, that Automotive02, provides four versions in its evolution history. Hence, we can calculate only three stability values for this product line. Which means, that only the minimum, the first quartile, and the average value

https://doi.org/10.24355/dbbs.084-201812111412-0

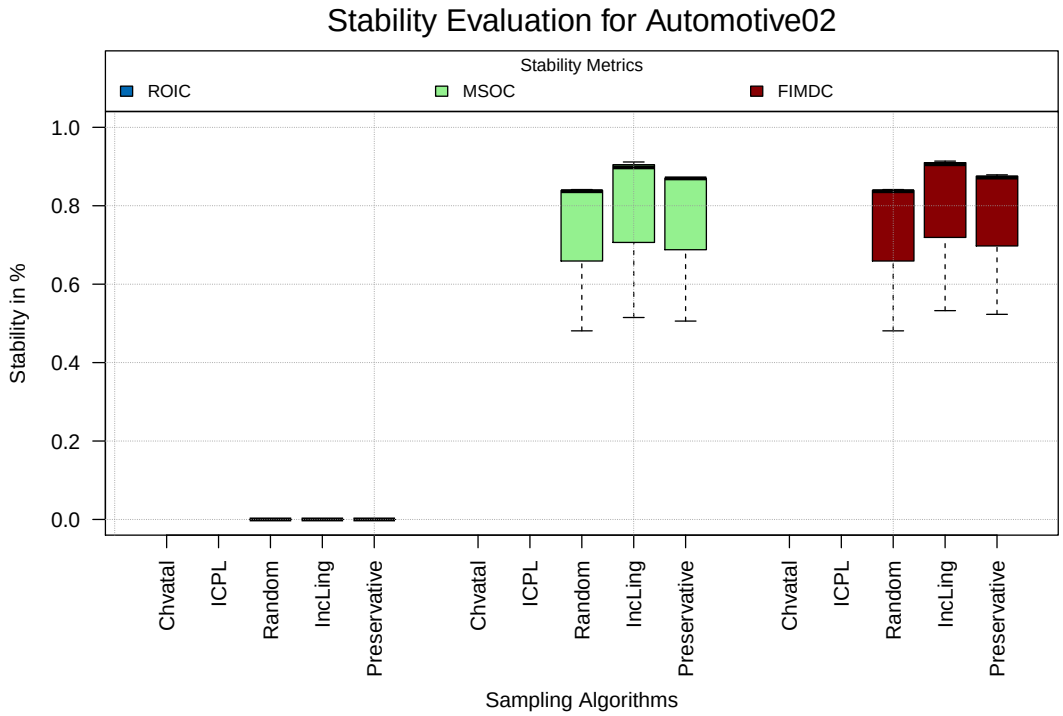

Figure 5.12: Box Plot: Results of Calculating Sample Stability for Automotive2

of our box plots can be set. As for the *Ratio of Identical Configurations* metric we can see stability values equal to zero, for all sampling algorithms used. *Mean Similarity of Configurations* and *Filter Identical Match Different Configurations* calculate nearly the same distribution of sample stability values for all sampling algorithms shown in Figure 5.12. Therefore, the following descriptions apply for both metrics. Both metrics measure the highest average sample stability (about 90%) for samples calculated by the IncLing algorithm. Even though, Preservative Sampling uses IncLing as base algorithm, it stays behind with an average stability value of about 85%. The lowest average sample stability is calculated for samples of the Random sample procedure. All algorithms show about the same data distribution with about 40 to 50 percent between the minimum and average value.

5.3.4 Interpretation of Stability Results

Throughout the product-line evolutions of Graph Library, Linux, Financial Services, and Automotive02 we discovered, that *Ratio of Identical Configurations* calculates the lowest stability values compared to the other metrics. Furthermore, *Ratio of Identical Configurations* calculates, for all product lines used in industry (Linux, Financial Services, and Automotive02), only sample stability values around or equal to zero. This behaviour can be explained with the implementation of this metric. It only considers identical configurations when calculating stability. Hence, if two configurations differ only slightly, they will contribute a value of zero to the stability calculation of *Ratio of Identical Configurations*.

As for *Mean Similarity of Configurations* and *Filter Identical Match Different Configurations*, we discovered that both metrics result to a very similar distribution of

https://doi.org/10.24355/dbbs.084-201812111412-0

sample stability for all product-line evolutions in our experiment. For this behaviour we can raise two different explanations. The first explanation considers the matching order for the heuristic used in *Mean Similarity of Configurations*. If *Mean Similarity of Configurations* matching finds all perfect matches (similarity value of one), it works the same as *Filter Identical Match Different Configurations*, which saves those perfect matches during a preprocessing. The other reason why those metrics show an equal distribution of sample similarity considers the absence of identical configurations. As described in Chapter 3, if no identical configurations can be found during the preprocessing of *Filter Identical Match Different Configurations*, this metric works the same as *Mean Similarity of Configurations*. Based on the low stability values calculated by *Ratio of Identical Configurations*, we can promote the second explanation.

Throughout all product lines, considered in our experiment, we observed that samples produced by Preservative Sampling reach slightly lower similarity values as for IncLing. The only exception can be seen in the Graph Library example, measured with the *Ratio of Identical Configurations* metric. To explain this behaviour, we need to consider the concept of Preservative Sampling given in Section 3.6. Preservative Sampling extents the IncLing algorithm by reusing configurations of a previously calculated sample, if those are still valid. Hence, a similar distribution of sample stability can be expected, if no configurations can be reused between evolution steps.

However, the aim of Preservative Sampling was to increase the stability of samples throughout the product-line evolution. With respect to the data, a fulfilment of this aim can only be confirmed for the samples produced for Graph Library and only when measured with *Ratio of Identical Configurations*. Based on the implementation of Preservative Sampling, two explanations for the observed behaviour can be raised. The first explanation considers the increase of sample size, which results from reusing previous configurations. As described in Chapter 3, a larger sample size can have negative influence on the sample stability calculated by our metrics, if its growth is not proportional to the number of similar configurations contained in the sample.

The second explanation considers the validity check performed by preservative sampling. As described in Section 3.6, previous configurations are only valid for the new feature model, if they can be reused without adjustments. This kind of implementation makes the algorithm fragile against small changes between product-line versions. For example if an evolution step includes the insertion of a new core feature, all previously calculated configurations will be invalid on the feature model. In this case, the preprocessing of Preservative Sampling does not bring any benefits and a standard sampling with IncLing is performed. Hence, a similar distribution between IncLing and Preservative Sampling can be expected for product lines such as Linux, Financial Services, and Automotive02, which potentially change allot between two versions.

5.4 Answering Research Questions

In this section we answer our previously defined research questions. The answers to those are based on our stability evaluation of the product lines Graph Library, Linux,

https://doi.org/10.24355/dbbs.084-201812111412-0

Financial Services, and Auomotive02. We evaluate, which sampling algorithms produce most stable samples for those product lines. Furthermore we evaluate which sample algorithms produce, most unstable samples. This evaluation, leads to answering Research Question 1. In the second subsection we discuss performance of our stability metrics in context of calculating sample stability. By doing so we answer Research Question 2. In the third subsection Preservative Sampling is discussed. By comparing the performance of our self implemented sample algorithm against the performance of IncLing, we answer Research Question 3.

5.4.1 RQ1: Evaluating Stability of Sampling Algorithms

Our first research question aims to provide insights about the stability of different sampling algorithms. To provide the needed data we used sampling algorithms such as Chvatal, ICPL, IncLing and Preservative Sampling. As base line we used the Random sampling procedure integrated in FeatureIDE. We expect this procedure to produce most unstable samples, because of its random selection of configurations.

Our evaluation results from Section 5.3, show that the highest stability values can be calculated for samples produced by ICPL and Chvatal. Furthermore, we observed for both only a small spread in the stability distribution throughout the product-line evolutions. However, we also discovered a scalability problem for product lines with a high number of features. This discovery is in accordance to previous research results[AHKT+16].

As for IncLing and Preservative Sampling, we discovered that both algorithms follow nearly the same distribution of stability values. However, Preservative Sampling shows always a bit lower values as IncLing. Compared to ICPL and Chvatal, the stability values measured for IncLing and Preservative Sampling show the lowest average stability. Furthermore, IncLing and Preservative Sampling show a high spread in their distribution of stability values.

Based on our observations we evaluate, that the sampling algorithms ICPL and Chvatal can produce more stable samples in comparison to IncLing and Preservative Sampling. Hence, ICPL and Chvatal can be used in application areas where stable samples are needed. As introduced, such an area would be the performance testing of software product lines. Nonetheless, we confirmed previous research results[AHKT+16], in context of scalability problems of ICPL and Chvatal. Hence, the size and complexity of product lines need to be considered, if ICPL and Chvatal are used to create samples.

As for Incling and our Preservative Sampling, we evaluate that they can be used to produce unstable samples. This statement is based on the low values for sample stability throughout our experiment. Furthermore, both algorithms show a high distribution of stability values, which indicates an unstable behaviour during the sampling process. Hence, they can be used in areas where exploring different aspects of the product line is of importance.

5.4.2 RQ2: Evaluating the Relevance of Stability Metrics

Our second research question considers the applicability of *Ratio of Identical Configurations*, *Mean Similarity of Configurations*, and *Filter Identical Match Different*

https://doi.org/10.24355/dbbs.084-201812111412-0

Configurations for large product lines used in real world industries. To answer this question, we consider the calculation results for all product lines contained in our experiment suit.

The data shown in Section 5.3 indicate, that the sample stability calculated by *Ratio of Identical Configurations* results to values about zero, for larger product lines like Linux, Financial Services and Automotive02. The only product line, for which *Ratio of Identical Configurations*calculates reaches higher sample stability values, is the Graph Library. However, those values are still the lowest compared to the calculation results of *Mean Similarity of Configurations* and *Filter Identical Match Different Configurations*. As discussed in Section 5.3.4, this behaviour results from only considering identical configurations in the stability calculation. Hence, we can conclude that *Ratio of Identical Configurations* is not applicable for real world product lines used in industry.

As for *Mean Similarity of Configurations* and *Filter Identical Match Different Configurations* the data shown in Section 5.3 indicate, that both metrics follow about the same distribution of sample stability values for all considered sampling algorithms. Section 5.3.4 identifies two reasons for this occurrence. The first reason could be that no identical configuration exists in the considered samples. The other reason suggests that the heuristic used in *Mean Similarity of Configurations* finds all identical configuration by itself. Since both *Mean Similarity of Configurations* and *Filter Identical Match Different Configurations* follow the same distribution of sample stability, we cannot reach a conclusion on which metric is to be preferred, based on our calculation results. However, by considering the concepts of both metrics we can discuss some advantages and disadvantages of both metrics.

Filter Identical Match Different Configurations extends the *Mean Similarity of Configurations* metric by filtering all identical configurations that exist between the sample pair. This leads to a small overhead of calculation time, but ensures that identical configurations will be considered in the stability calculation. The heuristic used by *Mean Similarity of Configurations* does not provide such an assurance, but promises potentially faster calculation times. Hence, we conclude, that *Filter Identical Match Different Configurations* should be used to ensure higher stability values.

5.4.3 RQ3: Evaluating Preservative Sampling

Our self implemented sampling algorithm, Preservative Sampling, aims to maximize the sample stability of calculated samples. To do so, it extends IncLing by a preprocessing, which loads previous calculated configurations, checks them for validity and uses them as base sample for the incremental calculation of new configuration with Incling. Research Question 3 considers the performance of Preservative Sampling in context of runtime and testing efficiency compared to IncLing.

As shown in Section 5.3.1, the Preservative Sampling shows in average slightly higher calculation times as IncLing. The overhead is most clearly visible for the Graph Library product line. For the larger product lines, this overhead takes only small portion of the total runtime, so it cannot be observed as clearly. We determined that

https://doi.org/10.24355/dbbs.084-201812111412-0

this runtime overhead results from the preprocessing needed to check the validity of previously calculated configurations.

With respect to the testing efficiency of Preservative Sampling, we ascertained that the algorithm produces about the same number of configurations as IncLing for the product lines Linux, Financial Services, and Automotive02. The number of configuration differs only for the Graph Library product line. Based on this finding, we concluded that the reuse mechanism of Preservative Sampling is fragile against small changes in the feature models of different product line versions. An example would be the insertion of a core feature. Hence, we concluded that the reuse mechanism of Preservative Sampling only works for product lines which do not change allot between evolutions steps.

During the stability analysis conducted in Section 5.3.3, we discovered that Preservative Sampling generates samples slightly lower stability than IncLing. This pattern can be seen for all product lines and all metrics except for Graph Library measured by *Ratio of Identical Configurations*. In our interpretation, conducted in Section 5.3.4 we identified the fragile reuse mechanism of Preservative Sampling as one reason for this pattern. Additionally, we assume that the increase in sample size is another reason for the discovered pattern.

Based on the arguments provided above, we conclude that Preservative Sampling does perform slightly worse than IncLing in regard to runtime efficiency and shows the same results for testing efficiency. However, it does not increase the sample stability of calculated samples, compared to IncLing. On the contrary, we observed even lower values for sample stability. Hence, Preservative Sampling misses its original aim to maximize the sample stability throughout the product-line history.

Moreover, we conclude that the reason behind missing our aim is the fragile reuse mechanism implemented in Preservative Sampling. To improve the reuse mechanism the implemented validity check for constraints need to be adjusted. The adjustment need to assure that small changes between the previous and the current feature model do not result in invalid configurations. As small changes to feature models we identify, for example, the insertion and removal of core and normal features.

5.5 Threats to Validity

In this section we discuss threats to the validity of our evaluation results. First we analyse factors that threaten in internal validity of the results. To do so, we check if our results can be influenced by factors not considered during the experiment set up and execution. Thereafter, we analyse the external validity of our results, by checking whether the results can be generalized to similar examples in the application area.

5.5.1 Internal Validity

To answer Research Question 3, we compare performance of Preservative Sampling with IncLing based on runtime and testing efficiency. To do so, computation time and number of configurations produced during execution of the algorithms is measured. The computation time can be influenced by the schedulers of the operating

https://doi.org/10.24355/dbbs.084-201812111412-0

system or the CPU. In addition the Java garbage collector can have an influence to the measured values. Variations based on the named causes are called a computational bias. To mitigate the influence of this bias, we executed the sample generation for all sampling algorithms considered in our experiment five times. Each time we measured computation time and configurations. For the collected data, we calculated the average. By doing so we mitigate the bias, but cannot assure the exactly same results for new executions of our experiment.

The used metrics to calculate sample stability and evaluate it for Research Question 1, are self developed and implemented by ourselves. Even though we carefully planed and implemented our metrics as described in Chapter 3 and Section 4.2, we can not assure that no errors occurred during both processes. To mitigate the threat of errors in concept and implementation, we tested our metrics on the Graph Library product line. As described before, this product line is small and artificially created by ourselves. Hence, we were able to comprehend the behaviour of metrics theoretically and check its validity. Nonetheless, we cannot assure that our small example covers all corner cases included in large product lines used in real world industry. Consequently, the threat of invalid concept and implementation still exists. Furthermore, our evaluation faces the threat that our metrics are not the most appropriate way to measure stability of samples. To mitigate this threat, we discussed different designs of metrics in Chapter 3. Out of those ideas, we chose the most promising for our evaluation.

Beside implementing our own metrics, we also implemented the Preservative Sampling algorithm ourselves. The validity of this implementation is threatened by the threats described for our metrics. To mitigate these threats, we performed a qualitative validation process by using the Graph Library. However, we can not assure the correctness of the sampling for other product lines.

Another threat to internal validity exits for results based on the Linux product line used in our evaluation. To be exact, the used Linux models pose two threats to validity. The first threat consists of not using the original Linux kernel model described in KConfig, but rather a transformation done by the KConfigReader. As described in Section 4.4, KConfigReader uses its own procedures to transform Linux variability models into propositional logic. Even though, El-Sharkawy et al. [ESKS15] state that KConfigReader is a reliable tool for transforming Linux variability models into propositional logic, they also state that KConfigReader cannot cover all corner cases of Linux. The second threat is based on the fact, that we do not use the complete model created by KConfigReader, but rather a randomly selected part of it. Hence, we cannot assure the applicability of our results to the complete model created by KConfigreader or to the original Linux model.

5.5.2 External Validity

Our evaluation results are threatened by external influence in regard of a generalisation to product lines, not contained in our experiment set up. We cannot ensure that all of our results hold for different product lines other than those used in our experiment. Especially large scale product lines, used in real world industry can contain corner cases not considered in our evaluation. To mitigate this threat we

https://doi.org/10.24355/dbbs.084-201812111412-0

conducted our evaluation on the following four product lines: Graph Library, Linux, Financial Services and Automotive02. By analysing the evolution histories of those four product lines, we cover a range from small to large scale product lines. Additionally we use two product-line evolution histories with origins in real world industry (Financial Services and Automotive02). In regard to the limited time frame of this master thesis, we were not able to conduct experiments for more product-line evolutions. Hence, our result may not be generalisable for all product line versions in the world. However, our evaluation provides insights on how stable different sampling algorithms for different product-line evolutions work.

Furthermore, the results shown in this chapter, are based on the sampling algorithms Chvatal, ICPL, Random, IncLing and Preservative Sampling. All of this algorithms consider the feature model as sole parameter for calculating samples. However, different classes of sampling algorithms can be used to generate samples for product lines. Varshosaz et al. [VAHT+18] identified the following classes of input parameters for sampling algorithms: Feature Model, Expert Knowledge, Implementation Artefacts, and Test Artefacts. Using only sampling algorithms from one of those classes poses the threat that our results are not applicable in general. However, in the short time span of our thesis, we were not able to conduct experiments with algorithms from all defined classes. Hence, we focused to keep our results valid for one class of algorithms, by using the established sampling algorithms for this class.

5.6 Summary

In this chapter we evaluated the sample stability of four established sampling algorithm, Chvatal, ICPL, Random sampling, and IncLing, and one self developed algorithm (Preservative Sampling). The Preservative Sampling is already described in Section 3.6. The named sample algorithms were used to create samples for the following product-line evolutions: Graph Library, Linux, Financial Services, and Automotive02. We presented the evolution histories for the named product lines in this chapter, with respect to the growth in feature and configuration size. To determine how stable the named sample algorithms work throughout the evolution history of a product line, we use our self developed metrics. The metrics are described in Chapter 3.

To lead our evaluation, we defined three research questions. The first research question is directly concerned with evaluating which sampling algorithm produces stable samples for a product-line evolution and which algorithms produce unstable samples. With the second research question we want to evaluate how relevant our metrics for sample stability are, when applied to large scale feature models. The last question is concerned with the performance of our self developed Preservative Sampling, in comparison to the performance of IncLing.

To answer the research questions we sampled the product-line evolutions for all product lines contained in our experiment. To sample a product-line evolution, we executed all sampling algorithms contained in our experiment, on all the product-line versions contained in the respective product-line evolution history. While executing the sampling algorithms, we measured the sampling and testing efficiency. For the resulting samples we measured the sample stability by using our metrics *Ratio of*

https://doi.org/10.24355/dbbs.084-201812111412-0

Identical Configurations, *Mean Similarity of Configurations*, and *Filter Identical Match Different Configurations*.

After generating the needed data, we visualized, described and interpreted them. Doing so revealed that the sampling algorithms ICPL and Chvatal produce most stable samples for a product-line evolution, while IncLing and Preservative Sampling, produce the most unstable samples. In addition we discovered that the *Ratio of Identical Configurations* metric cannot be used for product-line evolutions where the feature models change strongly between versions. As for *Mean Similarity of Configurations* and *Filter Identical Match Different Configurations*, we saw nearly the same stability distribution in our data. Hence, we concluded that it does not matter which metric is used to measure sample stability. By comparing the sample stability results for Preservative Sampling with those for IncLing, we discovered that both algorithms show nearly the same stability distribution. Hence, we concluded that Preservative Sampling does not fulfil its initial aim to produce most stable samples for a given product-line evolution.

At the end of this chapter we discuss possible threats to the validity of our results. This discussion shows, that we could mitigate some of the existing threats. However, other researchers may come to other results, by using different product-line evolutions as base for the sample stability analysis.

https://doi.org/10.24355/dbbs.084-201812111412-0

6. Related Work

In this master thesis we analyse different sampling algorithms in accordance to their sample stability. Previous research conducted by Varshosaz et al. [VAHT+18], also analysed sampling algorithms in accordance to different attributes. Varshosaz and his team, did an extensive literature search about sampling algorithms. They classified the accumulated knowledge about sampling algorithms in context to informations used for sampling, the kind of sampling algorithm, and the achieved coverage criteria. There research does not consider the stability of samples created by a sampling algorithm. One part of our thesis concentrates on classifying different sampling algorithms as stable or unstable. Even though, our experiment included only a small number of sampling algorithms, we achieved valuable information for those few. The knowledge we achieved in this master thesis, can complement the extensive work of Varshosaz et al.

Another research area we tackle in this master thesis is the evolution of product lines. This research area was a topic in previous work as well [TBK09, AHC+12, BKL+15]. Previous research on product-line evolution mostly focuses on the deduction of changes between feature models. For example, Bürdek et al. [BKL+15] conducted a real-world case study to observe complex change operations during the product-line evolution. They define a detailed set of complex edit operations based on their observations. Furthermore, Bürdek and his team present an approach for reasoning about impact of feature model changes, based on their defined edit operations.

Similar to Bürdek and his team, Thüm et al. [TBK09] also conducted research on changes occurring during the evolution of product lines. However, they take a more general approach and classify feature model changes based on the effects to the configuration space, of a product line. Thüm and his team define the following categories for feature model changes: *Refactoring*, *Generalization*, *Specialization*, and *Arbitrary Edit*. *Refactoring* edits do not change the configuration space of a product line, but could change its behaviour. *Generalizations* extend the configuration space of a product line, while *Specializations* reduce it. *Arbitrary Edit*, is the category for complex feature model edits, which cannot be classified in any other category. Be-

https://doi.org/10.24355/dbbs.084-201812111412-0

side defining those categories, Thüm et al. [TBK09] also present a tool to reason about feature model edits.

Unlike Bürdek et al. [BKL+15] and Thüm et al. [TBK09], we do not analyse feature model changes during the evolution of product lines directly. Instead we evaluate the sample stability of different sampling algorithms throughout the product-line evolution. However, our sample creation depends directly on the feature model. Hence, changes to the feature models between two product-line versions will be reflected in the samples. This provides a possible research direction for future work. Based on the results of Thüm et al. [TBK09], our results for sample stability could be qualified in context to the kind of feature model changes.

Based on our definition of sample stability, we need to measure the similarity between configurations. The research area of measuring similarity between configurations was previously tackled by Al-Hajjaji et al. [AHTL+18] and Henard et al. [HPP+14]. Both use the similarity between configurations to implement an ordering of configurations contained in a sample. Thereby, they propose techniques to raise the effectiveness of software product line testing. Our master thesis was mostly influenced by the approach of Al-Hajjaji et al. [AHTL+18], therefore we focus on their work.

Al-Hajjaji et al. [AHTL+18] propose a similarity based prioritisation of configuration to improve software product line testing. They reorder the existing sample stepwise. In each step, the configuration which is most dissimilar to the already selected configurations is chosen next. The implementation is based on the assumption that most errors can be found in the most dissimilar configurations. The evaluation results of Al-Hajjaji et al. show, that their prioritisation ordering performs most of the time better than a standard ordering.

Unlike Al-Hajjaji et al. [AHTL+18], we do not use the similarity between configurations to establish an ordering of samples. Instead, we realise a matching between configurations, based on the similarity between them. Moreover, we aggregate the calculated similarity values to achieve a value for sample stability. Even though we use the same basic metric to determine similarity between configurations (Hamming similarity), our matching is not based on the most dissimilar but on the most similar configurations. Furthermore, Al-Hajjaji and his team conduct their similarity analysis only on one product-line version, while we examine the similarity between configurations of two consecutive product-line versions. Nonetheless, the ideas of Al-Hajjaji et al. [AHTL+18] provided us with the initial idea to use the Hamming similarity as metric for similarity between configurations.

https://doi.org/10.24355/dbbs.084-201812111412-0

7. Thesis Summary

With software product lines, products can be generated from a large number of features. However, to test all possible products is often infeasible, because the number of products grows exponentially with the number of features. To cope with this challenge, sampling algorithms are used to generate product samples, which can be tested. Currently, sampling algorithms are evaluated with regard to their sampling efficiency and testing efficiency for one version of the product line. With this master thesis, we introduce another criterion to evaluate sampling algorithms, which is the sample stability of product sampling under product line evolution. The stability of samples under product line evolution was not defined before we started researching this topic. Hence, we deducted our own definition. We defined the stability of samples under product line evolution, as similarity between two samples.

Based on our definition we developed three concepts to measure the stability between two product samples. The first metric (*Ratio of Identical Configurations*) considers the ratio of identical configurations in both samples, to calculate their stability value. Our second concept of measuring sample stability is called *Mean Similarity of Configurations*. This metric is based on the similarity of configurations contained in two samples. We use a heuristic to match the most similar configurations between both samples. The similarity values of matched configurations, are aggregated to generate the sample stability between the respective samples. However, the used heuristic does not ensure that always the best matches are found. Therefore, we developed a third metric called *Filter Identical Match Different Configurations*. This metric uses the same heuristic approach as *Mean Similarity of Configurations* to match similar configurations. However, it contains a preprocessing which filters identical configurations. Hence, we can ensure that at least identical configurations contribute with the highest similarity value to the sample stability. We implemented our concepts as Java command line applications.

As already mentioned, sample stability under product-line evolution is currently not an evaluation criterion for sampling algorithms. Therefore, no established sampling algorithms consider the product-line evolution in their sampling procedures. As part of this master thesis, we present the concept for Preservative Sampling, a self devel-

https://doi.org/10.24355/dbbs.084-201812111412-0

oped sampling algorithm. Our algorithm aims to maximize the stability of samples throughout the product-line evolution by reusing configurations of previously calculated samples. To implement our algorithm, we use the IncLing sampling algorithm as base and extend it by a preprocessing. The preprocessing checks configurations of a previously generated sample, whether they are still valid for the new product line version. If this is the case they are reused as base for the new sampling process. Preservative Sampling is implemented as Java command line application.

To evaluate concepts and implementations we generate samples with the following established sampling algorithms: Chvatal, ICPL, Random, and IncLing. Furthermore, we use Preservative Sampling to generate samples. We use the named sampling algorithms to generate samples for evolutions of the following product lines: Graph Library, Linux, Financial Services, Automotive02. To use the Linux product line, we needed to convert the original variability model into a feature model in the FeatureIDE XML format. We evaluated different possibilities to do so. Our evaluation has shown, that we could only use a partial Linux model. Hence, we implemented a Java tool to generate a product-line history of partial Linux models. During the sampling we measured the calculation time as well as the sample size, for each algorithm. Based on the collected information we evaluate part of Research Question 3, by comparing the runtime and testing efficiency of Incling and Preservative Sampling. The produced samples are used to measure sample stability for the respective algorithm. To measure the stability of samples, we use our developed metrics.

Based on the measured stability values we evaluate the sample stability between samples contained in a product-line evolution. Our results show that ICPL and Chvatal show the tendency to produce more stable samples for product lines used in industry. IncLing and Preservative Sampling, show a higher distribution between stability values, compared to ICPL and Chvatal. Hence, we concluded that both algorithms produce more unstable samples, throughout a product-line evolution. Based on this conclusion, we can evaluate for Research Question 3 that Preservative Sampling misses its original goal. As reason we identified the naive validity check for previous configurations.

By evaluating the calculated stability values, we also discovered that *Ratio of Identical Configurations* calculates very low sample stability compared to the other two metrics. Moreover, we discovered a similar distribution between *Mean Similarity of Configurations* and *Filter Identical Match Different Configurations*. Based on our observations, we concluded that *Ratio of Identical Configurations* is not applicable for large product lines used in industry, because only few identical configurations appear for those product lines. Furthermore we concluded, that the difference in sample stability between *Mean Similarity of Configurations* and *Filter Identical Match Different Configurations* is small enough that both metrics could be used as stability measures.

All in all, we used this master thesis to conduct a first evaluation of sample stability for different sampling algorithms. We implemented those concepts as tools, which were used to conduct the evaluation on sample stability. Our evaluation results provide first insights in how to measure sample stability and which sampling algorithms create the most stable samples. Even though not all of our concepts achieved the

https://doi.org/10.24355/dbbs.084-201812111412-0

expected results, we obtained valuable information on how to adjust those concepts to work as expected. By doing so, we provide ideas and tools to encourage further research in the field of stability of product sampling under product-line evolution.

https://doi.org/10.24355/dbbs.084-201812111412-0

https://doi.org/10.24355/dbbs.084-201812111412-0

8. Future Work

This thesis provides first insights on how sample stability can be measured and how stable some sampling algorithms work when applied to product-line evolutions. However, in the limited scope of this master thesis, we were only able to cover a small share of the large research topic of stability of product sampling under product-line evolution. Therefore, many ideas for further research can be identified, which are presented in the following subsections.

Alternative Metric Concepts and Implementations

Based on the limited time frame of this master thesis, we implemented only three concepts of stability metrics. However, in Chapter 3 we presented more conceptual ideas to calculate the stability of product samples. In future research different concepts for metrics to calculate sample stability could be implemented and compared to the results presented in our thesis. In Chapter 3, we provide some leads for this further research by presenting design decisions not implemented in our thesis. For example, we describe in Chapter 3 that either the Hamming distance or the Jacard metric can be used to implement the *Mean Similarity of Configurations* metric. We chose the Hamming similarity to implement *Mean Similarity of Configurations* in our thesis, to consider selected as well as deselected configurations. The influence of this choice could be researched by implementing a *Mean Similarity of Configurations* version which uses the Jacard metric as similarity measure.

Another research direction could be to improve the matching procedure used in our *Mean Similarity of Configurations* metric. Currently, the matching process is based on a heuristic to find best matches between configurations. As described in Chapter 3, this heuristic does not ensure that all the best matches will be found. To improve this, an optimisation algorithm could be used. We already discovered, that the matching used in *Mean Similarity of Configurations* can be described as matching problem on bipartite graphs. For those problems, different optimisation algorithms exist in mathematics, such as the Hungarian algorithm [Kuh56]. By implementing a *Mean Similarity of Configurations* version with this algorithm, best matches between configurations would be ensured. This assurance could lead to higher and more realistic sample stability values.

https://doi.org/10.24355/dbbs.084-201812111412-0

Context Free Evaluation of the Stability of Sampling Algorithms

In this master thesis, we evaluated how stable different sampling algorithms work, in context of product-line evolution. To do so, we created samples for the evolution and measured the stability between sample pairs of different versions. To measure the stability of sampling algorithms in this master thesis, holds relevance for regression and performance testing of product lines. However, the assertions about sample stability of the algorithms are influenced by changes in the product line. An interesting research direction would be to measure the sample stability for samples on one version of the product line.

A possible approach would be, to sample the same product-line version multiple times with the same sampling algorithm. Thereafter, the sample stability between all runs will be measured. The resulting stability values can be analysed and accumulated to a global stability value for the algorithm. An algorithm which produces stable samples, should reach results close to a stability value of one, while unstable sampling algorithms should reach a stability value close to zero. By using this approach, the product-line evolution as factor for change is eliminated. Hence, a context free evaluation of the stability of sampling algorithms would be possible.

Analysis of Sampling Algorithms from Different Categories

In this master thesis, we evaluated the sample stability of different sampling algorithms. However, our evaluated algorithms take only the feature model as input parameter for the sampling process. As Varshosaz et al. [VAHT+18] describe, some sampling algorithms also take other product line artefacts into account. Varshosaz et al. [VAHT+18] define the following categorise based on the input parameters of sampling algorithms: Feature Model, Expert Knowledge, Implementation Artefacts, and Test artefacts.

In future research, the stability of samples produced by sampling algorithms from the categories presented by Varshosaz et al. [VAHT+18] could be evaluated. The new results, could be compared against the results presented in this master thesis. By doing so, the influence of different artefacts on sample stability could be researched. Based on this research, it could also be possible to achieve knowledge about which category of sampling algorithm can produce the most stable samples. Moreover, it would be possible to identify the sampling algorithm which produces the most stable samples or most unstable samples throughout all categories.

Improve Selection of Partial Linux Models

As described in Section 4.4 we use a random feature selection strategy to build partial feature models for Linux kernel. By using these models as base for our analysis, we introduce a self constructed bias to our evaluation. Therefore, the result achieved for the Linux product line, are only partially relevant for the original product line. As future work, we could consider approaches to select the partial feature model differently, to make our results more relevant.

https://doi.org/10.24355/dbbs.084-201812111412-0

One approach could be, to create a partial Linux feature model, based on our random selection procedure, and build an evolution based on it. To build an evolution history, we could use mutation operators proposed by Reuling et al. [RBR+15]. We would use the presented operators randomly to model the evolution of the product line. By using mutation operators to evolve the product line, we could control how much the feature model changes from version to version. Furthermore, we could control how often a mutation operator is used, compared to others. If the application of mutation operators, follows the typical distribution of changes in feature models, analysed by Passos et al. [PGT+13], we could build an evolution history close to reality. By doing so, our analysis results would be more relevant for real world applications.

Use Import of *.model Files to Convert Linux Feature Models

During this master thesis we discovered, that our implemented import mechanism for *.model files, face scalability problems when applied to *.model files produced from Linux kernel. As reason we identified complex boolean constraints which are contained in the files. To translate those constraints a lot of computation time is needed. As future work we could address this challenge by implementing a time out, when converting a boolean constraint, from the *.model file into FeatureIDE's XML format. If the application needs to long to convert a constraint, this constraint is skipped. Therefore, we would introduce a small bias in our evaluation of Linux history. However, this method could enable us to analyse a large proportion of the original Linux model. Implementing the adjustment of our import functionality was not in the scope of this master thesis.

Improve Sample Stability of Preservative Sampling

Our evaluation of Preservative Sampling has shown, that it does not fulfil its original aim, to maximize sample stability throughout the product line evolution history. We discovered that the validity check of Preservative Sampling is to restrictive for larger product lines used in industry. The feature models of those product lines change allot between each version. Therefore, previously calculated configurations are natively invalid for the new feature models. This could be observed in most of our evaluated data. To face this challenge and improve the stability of generated sample, future work could consider adjusting the validity check of Preservative Sampling.

An entry point to improve Preservative Sampling could be to integrate a configuration correction into the algorithm. This means, if an invalid configuration is discovered, the algorithm tries to correct it automatically. The auto correction of configurations is a difficult task, so only some heuristics can be used.

One of heuristic, could be to filter all core and dead features from the new feature model. All core features, which are found during the analysis are added to the previous configurations, if they are not already contained. All dead features are removed from previous configurations. By doing so, a valid configuration for the new feature should be generated from an old one. Hence, we would secure the reuse of previously calculated configurations.

https://doi.org/10.24355/dbbs.084-201812111412-0

The presented idea is a first approach to improve the sample stability of Preservative Sampling. By conducting further research in this direction, new ideas to improve sample stability of Preservative Sampling can be revealed.

Take Changes between Feature Models into Account

The aim of this master thesis was to provide first insights into the topic of sample stability under product-line evolution. Therefore, we only measured the stability of samples, without analysing the influences of changes between product lines. Future research could analyse which change operations were used between two product line versions and use this knowledge to interpret the measured stability values.

To analyse the changes occurring in the product-line evolution, ideas presented by Thüm et al. [TBK09] could be used. They present a categorisation system of feature model edits. Furthermore, Thüm and his team developed an algorithm, which supports the reasoning about feature model edits, based on their categorisation system. Future research could use the given algorithm to analyse which category of edit has happened between two feature models. Thereafter, the stability between samples of those feature models can be measured by our metrics. By applying this procedure to different product lines, new insights can be gathered. For example, knowledge about the influence of feature model edits to sample stability could be achieved. Based on this knowledge, we could possibly deduce a general statement on how sample stability is influenced by different categories of feature model edits.

https://doi.org/10.24355/dbbs.084-201812111412-0

A. Appendix

A.1 Sampling Efficiency Data

This section provides the aggregated sampling efficiency data for the product lines Graph Library, Linux, Financial Services, and Automotive02. The data shown in the tables bellow, are visualised in Figure 5.7. Each value provided in the tables, represents the computation time needed to reach pairwise coverage in minutes, for the respective algorithm.

Graph Library

	Chvatal	ICPL	Random	Incling	Preservative
V1	0.036	0.036	0.036	0.036	0.035
V2	0.037	0.036	0.036	0.036	0.035
V3	0.036	0.036	0.037	0.036	0.035
V4	0.036	0.036	0.036	0.036	0.069
V5	0.036	0.037	0.036	0.036	0.038
V6	0.037	0.036	0.036	0.036	0.071
V7	0.037	0.036	0.038	0.036	0.108
V8	0.036	0.037	0.036	0.036	0.147
V9	0.036	0.036	0.036	0.036	0.151

https://doi.org/10.24355/dbbs.084-201812111412-0

Linux

	Chvatal	ICPL	Random	Incling	Preservative
V1	16.44	0.414	0.081	0.58	0.583
V2	19.233	0.445	0.056	0.433	0.529
V3	28.094	0.445	0.051	0.367	0.439
V4	28.328	0.434	0.058	0.467	0.511
V5	28.821	0.471	0.052	0.373	0.473
V6	29.302	0.479	0.049	0.34	0.405
V7	27.982	0.424	0.054	0.373	0.446
V8	27.575	0.41	0.053	0.367	0.424
V9	28.076	0.426	0.057	0.427	0.488
V10	28.374	0.413	0.05	0.34	0.404

Financial Services

	Chvatal	ICPL	Random	Incling	Preservative
V1	34.959	9.196	0.064	0.867	0.893
V2	96.039	34.658	0.289	37.176	37.466
V3	82.896	30.808	0.267	38.34	38.514
V4	80.766	30.86	0.278	38.029	36.566
V5	85.368	31.063	0.281	38.256	9.206
V6	94.872	30.084	0.29	37.845	37.832
V7	127.046	49.251	0.322	480.038	480.882
V8	110.781	43.143	0.331	51.181	52.552
V9	139.581	52.281	0.407	63.972	64.396
V10	139.326	51.823	0.417	60.388	61.141

Automotive02

	Chvatal	ICPL	Random	Incling	Preservative
V1	481.667	481.667	0.408	481.62	480.296
V2	481.667	481.667	0.092	482.989	483.639
V3	481.667	481.667	0.079	483.113	482.019
V4	481.667	481.667	0.08	482.963	482.013

A.2 Testing Efficiency Data

This section provides the aggregated testing efficiency data for the product lines Graph Library, Linux, Financial Services, and Automotive02. The data contained in the tables below represents the number of configurations needed to reach pairwise coverage for the respective sampling algorithm. We visualised the data of the tables in Figure 5.8.

https://doi.org/10.24355/dbbs.084-201812111412-0

Graph Library

	Chvatal	ICPL	Random	Incling	Preservative
V1	7	7	6	6	6
V2	8	9	8	8	11
V3	10	11	9	9	14
V4	10	11	9	9	21
V5	9	9	8	8	18
V6	10	9	10	10	25
V7	9	9	10	10	56
V8	13	14	11	11	68
V9	15	15	15	15	65

Linux

	Chvatal	ICPL	Random	Incling	Preservative
V1	51	53	239	239	239
V2	49	51	164	164	164
V3	51	52	133	133	133
V4	48	51	188	188	188
V5	52	54	148	148	148
V6	55	54	121	121	121
V7	47	48	159	159	172
V8	49	48	155	155	155
V9	49	50	190	190	190
V10	51	49	139	139	139

Financial Services

	Chvatal	ICPL	Random	Incling	Preservative
V1	404	403	399	399	399
V2	3199	3203	4560	4560	4560
V3	3194	3197	4563	4563	4563
V4	3190	3190	4550	4550	4568
V5	3196	3193	4546	4546	4539
V6	3189	3190	4542	4542	4542
V7	3632	3638	7802	7802	7885
V8	3551	3555	4975	4975	4975
V9	4434	4440	6392	6392	6392
V10	4427	4419	6245	6245	6245

Automotive02

	Chvatal	ICPL	Random	Incling	Preservative
V1	0	0	175	175	177
V2	0	0	124	125	128
V3	0	0	125	125	133
V4	0	0	125	125	130

https://doi.org/10.24355/dbbs.084-201812111412-0

A.3 Stability Data for the Graph Library Product Line

This section provides the aggregated stability data for the Graph Library product-line evolution. Each table below, represents the data for one of the following sampling algorithms: Chvatal, ICPL, IncLing, Random, and Preservative Sampling. The stability values are measured by our three metrics: *Ratio of Identical Configurations* (ROIC), *Mean Similarity of Configurations* (MSOC), and *Filter Identical Match Different Configurations* (FIMDC). Each column of the tables represents stability values measured by the respective metric. As described in Chapter 3 the stability between two samples is measured as a values between 0 and 1. A value of 0 represents the lowest possible stability, while 1 represents the highest possible stability. We visualised the data shown in the tables below in Figure 5.9.

Chvatal

	ROIC	MSOC	FIMDC
V1 to V2	0.071	0.766	0.766
V2 to V3	0.286	0.711	0.711
V3 to V4	0.111	0.830	0.830
V4 to V5	0.000	0.483	0.483
V5 to V6	0.133	0.788	0.788
V6 to V7	0.063	0.608	0.608
V7 to V8	0.177	0.631	0.631
V8 to V9	0.083	0.772	0.772

ICPL

	ROIC	MSOC	FIMDC
V1 to V2	0.067	0.681	0.681
V2 to V3	0.539	0.778	0.778
V3 to V4	0.571	0.973	0.973
V4 to V5	0.056	0.436	0.436
V5 to V6	0.231	0.875	0.875
V6 to V7	0.067	0.875	0.875
V7 to V8	0.167	0.577	0.577
V8 to V9	0.080	0.839	0.839

Random

	ROIC	MSOC	FIMDC
V1 to V2	0.119	0.616	0.616
V2 to V3	0.170	0.731	0.731
V3 to V4	0.089	0.749	0.749
V4 to V5	0.127	0.550	0.550
V5 to V6	0.094	0.627	0.627
V6 to V7	0.079	0.720	0.720
V7 to V8	0.097	0.695	0.695
V8 to V9	0.068	0.624	0.624

https://doi.org/10.24355/dbbs.084-201812111412-0

IncLing

	ROIC	MSOC	FIMDC
V1 to V2	0.167	0.625	0.625
V2 to V3	0.214	0.741	0.741
V3 to V4	0.125	0.833	0.833
V4 to V5	0.231	0.573	0.573
V5 to V6	0.143	0.727	0.727
V6 to V7	0.385	0.944	0.944
V7 to V8	0.118	0.800	0.800
V8 to V9	0.000	0.621	0.621

Preservative Sampling

	ROIC	MSOC	FIMDC
V1 to V2	0.545	0.545	0.545
V2 to V3	0.786	0.786	0.786
V3 to V4	0.667	0.667	0.667
V4 to V5	0.560	0.679	0.679
V5 to V6	0.720	0.720	0.720
V6 to V7	0.421	0.445	0.445
V7 to V8	0.824	0.824	0.824
V8 to V9	0.727	0.878	0.878

A.4 Stability Data for the Linux Product Line

Similar to the previous section, this section presents detailed information in regard to our stability measurement. The contents of the tables below show the stability values for Linux product-line evolution, which are visualised in Figure 5.10.

Chvatal

	ROIC	MSOC	FIMDC
V1 to V2	0.000	0.659	0.660
V2 to V3	0.000	0.702	0.702
V3 to V4	0.000	0.644	0.644
V4 to V5	0.000	0.672	0.673
V5 to V6	0.009	0.690	0.691
V6 to V7	0.010	0.536	0.536
V7 to V8	0.011	0.699	0.699
V8 to V9	0.000	0.733	0.733
V9 to V10	0.000	0.707	0.707

https://doi.org/10.24355/dbbs.084-201812111412-0

ICPL

	ROIC	MSOC	FIMDC
V1 to V2	0.000	0.659	0.660
V2 to V3	0.000	0.687	0.687
V3 to V4	0.000	0.711	0.711
V4 to V5	0.000	0.698	0.699
V5 to V6	0.010	0.674	0.675
V6 to V7	0.010	0.616	0.617
V7 to V8	0.011	0.712	0.712
V8 to V9	0.000	0.697	0.697
V9 to V10	0.000	0.702	0.702

Random

	ROIC	MSOC	FIMDC
V1 to V2	0.000	0.314	0.314
V2 to V3	0.000	0.416	0.416
V3 to V4	0.000	0.436	0.436
V4 to V5	0.000	0.399	0.400
V5 to V6	0.000	0.430	0.430
V6 to V7	0.000	0.472	0.472
V7 to V8	0.000	0.585	0.585
V8 to V9	0.000	0.509	0.509
V9 to V10	0.000	0.355	0.355

IncLing

	ROIC	MSOC	FIMDC
V1 to V2	0.000	0.466	0.466
V2 to V3	0.000	0.591	0.591
V3 to V4	0.000	0.638	0.638
V4 to V5	0.000	0.563	0.563
V5 to V6	0.007	0.583	0.583
V6 to V7	0.007	0.674	0.674
V7 to V8	0.000	0.827	0.827
V8 to V9	0.000	0.743	0.743
V9 to V10	0.000	0.498	0.498

https://doi.org/10.24355/dbbs.084-201812111412-0

Preservative Sampling

	ROIC	MSOC	FIMDC
V1 to V2	0.000	0.466	0.466
V2 to V3	0.000	0.591	0.591
V3 to V4	0.000	0.638	0.638
V4 to V5	0.000	0.563	0.563
V5 to V6	0.007	0.583	0.583
V6 to V7	0.024	0.626	0.626
V7 to V8	0.000	0.717	0.717
V8 to V9	0.000	0.743	0.743
V9 to V10	0.000	0.498	0.498

A.5 Stability Data for the Financial Services Product Line

In this section, we present stability values for the Financial Services product-line evolution. The tables presented in this section follow the same structure as the tables shown in the previous sections. We visualised the table contents in Figure 5.11.

Chvatal

	ROIC	MSOC	FIMDC
V1 to V2	0.000	0.119	0.122
V2 to V3	0.000	0.989	0.990
V3 to V4	0.007	0.992	0.992
V4 to V5	0.505	0.996	0.996
V5 to V6	0.000	0.966	0.975
V6 to V7	0.000	0.856	0.861
V7 to V8	0.000	0.900	0.910
V8 to V9	0.000	0.794	0.796
V9 to V10	0.000	0.982	0.983

ICPL

	ROIC	MSOC	FIMDC
V1 to V2	0.000	0.008	0.008
V2 to V3	0.000	0.002	0.002
V3 to V4	0.007	0.991	0.991
V4 to V5	0.488	0.997	0.997
V5 to V6	0.000	0.968	0.977
V6 to V7	0.000	0.855	0.860
V7 to V8	0.000	0.900	0.910
V8 to V9	0.000	0.794	0.796
V9 to V10	0.000	0.976	0.977

https://doi.org/10.24355/dbbs.084-201812111412-0

Random

	ROIC	MSOC	FIMDC
V1 to V2	0.000	0.101	0.103
V2 to V3	0.000	0.830	0.831
V3 to V4	0.030	0.937	0.937
V4 to V5	0.098	0.989	0.989
V5 to V6	0.000	0.965	0.974
V6 to V7	0.000	0.617	0.620
V7 to V8	0.000	0.530	0.534
V8 to V9	0.000	0.771	0.773
V9 to V10	0.000	0.940	0.941

IncLing

	ROIC	MSOC	FIMDC
V1 to V2	0.000	0.082	0.085
V2 to V3	0.000	0.990	0.991
V3 to V4	0.004	0.988	0.988
V4 to V5	0.380	0.994	0.994
V5 to V6	0.000	0.969	0.978
V6 to V7	0.000	0.592	0.595
V7 to V8	0.000	0.463	0.468
V8 to V9	0.000	0.771	0.773
V9 to V10	0.000	0.940	0.941

Preservative Sampling

	ROIC	MSOC	FIMDC
V1 to V2	0.000	0.082	0.085
V2 to V3	0.000	0.990	0.991
V3 to V4	0.005	0.992	0.992
V4 to V5	0.609	0.985	0.985
V5 to V6	0.000	0.970	0.979
V6 to V7	0.000	0.564	0.566
V7 to V8	0.000	0.430	0.435
V8 to V9	0.000	0.771	0.773
V9 to V10	0.000	0.940	0.941

A.6 Stability Data for the Automotive02 Product Line

This section presents the detailed data of our stability measurement for Automotive02 product-line evolution. The tables below follow the structure described Section A.3. Figure 5.12, visualises the data contained in the tables below as box plot.

https://doi.org/10.24355/dbbs.084-201812111412-0

Random

	ROIC	MSOC	FIMDC
V1 to V2	0.464	0.481	0.481
V2 to V3	0.829	0.837	0.837
V3 to V4	0.839	0.841	0.841

IncLing

	ROIC	MSOC	FIMDC
V1 to V2	0.000	0.515	0.533
V2 to V3	0.000	0.898	0.906
V3 to V4	0.000	0.912	0.914

Preservative Sampling

	ROIC	MSOC	FIMDC
V1 to V2	0.000	0.506	0.523
V2 to V3	0.000	0.871	0.879
V3 to V4	0.000	0.870	0.872

https://doi.org/10.24355/dbbs.084-201812111412-0

https://doi.org/10.24355/dbbs.084-201812111412-0

Bibliography

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013. (cited on Page 1, 5, and 51)

[AGM+06] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos José Pereira de Lucena. Refactoring Product Lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 201–210. ACM, 2006. (cited on Page 9)

[AHC+12] Mathieu Acher, Patrick Heymans, Philippe Collet, Clément Quinton, Philippe Lahire, and Philippe Merle. Feature Model Differences. In *Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 629–645. Springer, June 2012. (cited on Page 105)

[AHKT+16] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. IncLing: Efficient Product-line Testing Using Incremental Pairwise Sampling. pages 144–155. ACM, 2016. (cited on Page 1, 2, 5, 7, 8, 9, 45, 54, 60, 82, 86, 92, and 99)

[AHMK+16] Mustafa Al-Hajjaji, Jens Meinicke, Sebastian Krieter, Reimar Schröter, Thomas Thüm, Thomas Leich, and Gunter Saake. Tool Demo: Testing Configurable Systems with FeatureIDE. pages 173–177. ACM, 2016. (cited on Page 52)

[AHTL+18] Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. Effective Product-Line Testing Using Similarity-Based Product Prioritization. 2018. To appear. (cited on Page 106)

[AHTM+14] Mustafa Al-Hajjaji, Thomas Thüm, Jens Meinicke, Malte Lochau, and Gunter Saake. Similarity-Based Prioritization in Software Product-Line Testing. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 197–206. ACM, 2014. (cited on Page 13, 22, and 23)

[BKL+15] Johannes Bürdek, Timo Kehrer, Malte Lochau, Dennis Reuling, Udo Kelter, and Andy Schürr. Reasoning about Product-Line Evolution Using Complex Feature Model Differences. 23(4):687–733, 2015. (cited on Page 9, 105, and 106)

https://doi.org/10.24355/dbbs.084-201812111412-0

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–708, 2010. (cited on Page 6, 9, and 21)

[Chv79] Vasek Chvatal. A Greedy Heuristic for the Set-Covering Problem. *Mathematics of operations research*, 4(3):233–235, 1979. (cited on Page 1, 2, and 7)

[CLRS09] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009. (cited on Page 34)

[DvDP17] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. Analysing the linux kernel feature model changes using fmdiff. *Software & Systems Modeling*, 16(1):55–76, 2017. (cited on Page 63)

[ESKS15] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. Analysing the kconfig semantics and its analysis tools. In *ACM SIGPLAN Notices*, volume 51, pages 45–54. ACM, 2015. (cited on Page 63, 65, 88, and 102)

[GCD11] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. Evaluating Improvements to a Meta-Heuristic Search for Constrained Interaction Testing. *Empirical Software Engineering (EMSE)*, 16(1):61–102, 2011. (cited on Page 1, 2, and 7)

[Goo18] Google. Google Guava. Website, 2018. Available online at https://github.com/google/guava; visited on September 12th, 2018. (cited on Page 58)

[Ham50] Richard W Hamming. Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160, 1950. (cited on Page 13)

[HPP+14] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. *IEEE Transactions on Software Engineering (TSE)*, 40(7):650–670, July 2014. (cited on Page 13 and 106)

[Jac12] Paul Jaccard. The distribution of the flora in the alpine zone. *New phytologist*, 11(2):37–50, 1912. (cited on Page 20, 22, and 55)

[JHF11] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. pages 638–652. Springer, 2011. (cited on Page 1, 2, and 7)

[JHF12] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. An Algorithm for Generating T-Wise Covering Arrays from Large Feature

https://doi.org/10.24355/dbbs.084-201812111412-0

Models. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 46–55. ACM, 2012. (cited on Page 1, 2, and 8)

[Kä18a] Christian Kästner. KConfig Reader. Website, 2018. Available online at https://github.com/ckaestne/kconfigreader; visited on August 29th, 2018. (cited on Page 65 and 67)

[Kä18b] Christian Kästner. Typchef Git. Website, 2018. Available online at http://ckaestne.github.io/TypeChef/; visited on September 2nd, 2018. (cited on Page 65)

[KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 311–320. ACM, May 2008. (cited on Page 9)

[Käs17] Christian Kästner. Differential testing for variational analyses: Experience from developing kconfigreader. *arXiv preprint arXiv:1706.09357*, 2017. (cited on Page 65 and 88)

[KAuR+09] Christian Kästner, Sven Apel, Syed Saif ur Rahman, Marko Rosenmüller, Don Batory, and Gunter Saake. On the Impact of the Optional Feature Problem: Analysis and Case Studies. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 181–190, 2009. (cited on Page 1)

[KGR+11] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 805–824. ACM, 2011. (cited on Page 65)

[KPK+17] Sebastian Krieter, Marcus Pinnecke, Jacob Krüger, Joshua Sprey, Christopher Sontag, Thomas Thüm, Thomas Leich, and Gunter Saake. FeatureIDE: Empowering Third-Party Developers. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 42–45. ACM, 2017. (cited on Page 52 and 60)

[KST+16] Sebastian Krieter, Reimar Schröter, Thomas Thüm, Wolfram Fenske, and Gunter Saake. Comparing Algorithms for Efficient Feature-Model Slicing. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 60–64. ACM, September 2016. (cited on Page 68)

[KSTS16] Sebastian Krieter, Reimar Schröter, Thomas Thüm, and Gunter Saake. An Efficient Algorithm for Feature-Model Slicing. Technical Report FIN-001-2016, University of Magdeburg, Germany, April 2016. (cited on Page 68)

https://doi.org/10.24355/dbbs.084-201812111412-0

[KTS+09] Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. FeatureIDE: A Tool Framework for Feature-Oriented Software Development. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 611–614. IEEE Computer Science, May 2009. Formal demonstration paper. (cited on Page 51 and 52)

[Kuh56] Harold W Kuhn. Variants of the hungarian method for assignment problems. *Naval Research Logistics Quarterly*, 3(4):253–258, 1956. (cited on Page 111)

[LHFRE15] Roberto E. Lopez-Herrejon, Stefan Fischer, Rudolf Ramler, and Aalexander Egyed. A first systematic mapping study on combinatorial interaction testing for software product lines. pages 1–10. IEEE Computer Science, April 2015. (cited on Page 7)

[LSB+10] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. Evolution of the linux kernel variability model. In *International Conference on Software Product Lines*, pages 136–150. Springer, 2010. (cited on Page 9)

[lva18] Linux Variability Analysis Tool (LVAT). Website, 2018. Available online at https://gsd.uwaterloo.ca/node/313; visited on August 29th, 2018. (cited on Page 65)

[McG01] John McGregor. Testing a Software Product Line. Technical Report CMU/SEI-2001-TR-022, Carnegie Mellon University, 2001. (cited on Page 1)

[MTS+17a] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. Featureide in a nutshell. In *Mastering Software Variability with FeatureIDE*, pages 19–29. Springer, 2017. (cited on Page 9)

[MTS+17b] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability with FeatureIDE*. Springer, 2017. (cited on Page 52)

[NC07] Linda Northrop and P. Clements. A Framework for Software Product Line Practice, Version 5.0. *SEI*, 2007. (cited on Page 1)

[OMR10a] Sebastian Oster, Florian Markert, and Philipp Ritter. Automated Incremental Pairwise Testing of Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 196–210. Springer, 2010. (cited on Page 1 and 43)

[OMR10b] Sebastian Oster, Florian Markert, and Philipp Ritter. Automated incremental pairwise testing of software product lines. In *International Conference on Software Product Lines*, pages 196–210. Springer, 2010. (cited on Page 45 and 60)

https://doi.org/10.24355/dbbs.084-201812111412-0

[OZML11] Sebastian Oster, Ivan Zorcic, Florian Markert, and Malte Lochau. Moso-polite: tool support for pairwise and model-based software product line testing. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, pages 79–82. ACM, 2011. (cited on Page 60)

[PGT+13] Leonardo Passos, Jianmei Guo, Leopoldo Teixeira, Krzysztof Czarnecki, Andrzej Wąsowski, and Paulo Borba. Coevolution of Variability Models and Related Artifacts: A Case Study from the Linux Kernel. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 91–100. ACM, 2013. (cited on Page 9, 10, 63, 82, and 113)

[POS+12] Gilles Perrouin, Sebastian Oster, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Pairwise Testing for Software Product Lines: Comparison of Two Approaches. *Software Quality Journa l (SQJ)*, 20(3-4):605–643, 2012. (cited on Page 1 and 43)

[PSK+10] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Automated and Scalable T-Wise Test Case Generation Strategies for Software Product Lines. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 459–468. IEEE Computer Science, April 2010. (cited on Page 1 and 43)

[RBR+15] Dennis Reuling, Johannes Bürdek, Serge Rotärmel, Malte Lochau, and Udo Kelter. Fault-Based Product-Line Testing: Effective Sample Generation Based on Feature-Diagram Mutation. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 131–140. ACM, 2015. (cited on Page 113)

[SLB+10] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. The variability model of the linux kernel. *VaMoS*, 10:45–51, 2010. (cited on Page 63)

[TBK09] Thomas Thüm, Don Batory, and Christian Kästner. Reasoning about Edits to Feature Models. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 254–264. IEEE Computer Science, May 2009. (cited on Page 105, 106, and 114)

[TKB+14] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming (SCP)*, 79(0):70–85, January 2014. (cited on Page 5, 51, 52, and 67)

[TSK06] P.N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Always learning. Pearson Addison Wesley, 2006. (cited on Page 20 and 55)

https://doi.org/10.24355/dbbs.084-201812111412-0

[VAHT+18] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammadreza Mousavi, and Ina Schaefer. A Classification of Product Sampling for Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*. ACM, 2018. To appear. (cited on Page 2, 6, 7, 45, 90, 92, 103, 105, and 112)

[VAM18] CADOS / VAMOS. Undertaker. Website, 2018. Available online at http://vamos.informatik.uni-erlangen.de/trac/undertaker; visited on August 29th, 2018. (cited on Page 65)

[Wei08] David M. Weiss. The Product Line Hall of Fame. In *Proceedings of the International Software Product Line Conference (SPLC)*, page 395. IEEE Computer Science, 2008. (cited on Page 1)

[Zip17] Roman Zippel. KConfig Documentation. Website, 2017. Available online at http://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt; visited on May 10th, 2017. (cited on Page 63)

https://doi.org/10.24355/dbbs.084-201812111412-0

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Braunschweig, den 08.11.2018

https://doi.org/10.24355/dbbs.084-201812111412-0