Identification of Feature Interactions Through Combinatorial Interaction Analysis

Sabrina Böhm

Master’s Thesis

Institute of Software Engineering and Programming Languages
Sabrina Böhm:
Identification of Feature Interactions Through Combinatorial Interaction Analysis

Advisors:
- Prof. Dr. Thomas Thüm, Dr. Sebastian Krieter
  Department of Software Engineering and Programming Languages
- Prof. Dr. Malte Lochau
  University of Siegen, Department of Electrical Engineering and Computer Science

Master’s Thesis, University of Ulm, 2022.
Abstract

One of the most challenging aspects of software engineering nowadays is to make software easily configurable. In the context of software product lines, developers use the concept of features that can either be selected or unselected for each product, which enables variability. When creating a configuration by selecting different features, it is desired that the configured product works correctly and no feature interaction occurs. A feature interaction describes the situation in which a feature influences one or more other features in that case, when the behavior of the system no longer matches the desired one. As it is practically not possible to test all configurations of a product line, especially for large feature models, sampling strategies have been developed to select small sets of configurations that cover every possible combination of features. Considering combinatorial interaction testing, one can identify faulty configurations but, due to the combinatorial explosion of potentially interacting features, it is currently an open challenge how to identify the feature interaction actually causing the fault. In this thesis, we present a novel concept to identify feature interactions through combinatorial interaction analysis based on a failing configuration. Throughout the analysis, we generate further configurations to gain more information about the occurrence of the feature interaction. We provide two algorithms to create new configurations, one random strategy and one divide-and-conquer strategy, which creates configurations deliberately by including about half of the remaining potential interactions. We evaluate eleven feature models for each algorithm. Our evaluation determines that the random algorithm isolates 82.6% of the pair-wise simulated feature interactions and the split algorithm isolates 91.3%. Our concept of feature-interaction identification forms a good basis to be adapted in future work.
## Contents

5 Evaluation 43

5.1 Research Questions ........................................ 43
5.2 Experiment Design ........................................... 44
5.3 Results ......................................................... 48
  5.3.1 RQ1: How effective can we isolate the feature interaction that leads to the failing configuration? 48
  5.3.2 RQ2: How many configurations are needed to isolate the feature interaction? 51
  5.3.3 RQ3: How long does it take to isolate the feature interaction that leads to the failing configuration? 52
5.4 Discussion .................................................... 54
  5.4.1 Threats to Validity ....................................... 57
5.5 Summary ....................................................... 58

6 Related Work .................................................. 61

7 Conclusion ...................................................... 65

8 Future Work ................................................... 67

9 Acknowledgements ............................................ 71

A Appendix ......................................................... 73
  A.1 Reduced Interactions per Configuration .................... 73
  A.2 Needed Configurations to Isolate the Feature Interaction 73
  A.3 Computation Time to Isolate the Feature Interaction ...... 78
# List of Figures

2.1 Example Feature Model Breakfast Describing a Breakfast Scenario | 6
2.2 Simplified Graphical Representation of Problem and Solution Space | 8
2.3 Compile Error Example For $\neg Water \land Tea$ | 10
2.4 Runtime Error Example For $\neg Water \land Tea$ | 10
2.5 Feature Model Breakfast Shortened | 13
3.1 The Basic Procedure of Combinatorial Interaction Analysis | 18
3.2 The Steps of Combinatorial Interaction Analysis | 21
3.3 The Set of All Pair-Wise Interactions vs. All Pair-Wise Potential Interactions | 22
3.4 Configuration Creation After the Split of Interactions | 26
3.5 Creation of Configuration by Including Interactions Step by Step | 27
4.1 Basic Procedure of Combinatorial Interaction Analysis Algorithm | 36
4.2 Needed Configurations to Isolate the Interaction per Number of Features for Generated Models | 38
4.3 Computation Of Potential Interactions | 39
4.4 Isolating Interaction Through Interaction Analysis | 39
4.5 Splitting the Potential Interactions | 41
4.6 Pseudo Code Create Configuration With Interaction Split | 41
5.1 Proceeding of an Experiment | 45
5.2 Overview of Experiments per Model | 45
5.3 Percentage of Isolated Interactions per Model | 49
5.4 Reduced Interactions per Configuration of GPL | 50
5.5 Reduced Interactions per Configuration of BusyBox | 50
5.6 Reduced Interactions per Configuration of Automotive01 | 51
<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.7</td>
<td>Number of Needed Configurations for GPL</td>
<td>52</td>
</tr>
<tr>
<td>5.8</td>
<td>Number of Needed Configurations for BusyBox</td>
<td>53</td>
</tr>
<tr>
<td>5.9</td>
<td>Number of Needed Configurations for Automotive01</td>
<td>53</td>
</tr>
<tr>
<td>5.10</td>
<td>Computation Time for GPL</td>
<td>54</td>
</tr>
<tr>
<td>5.11</td>
<td>Computation Time for BusyBox</td>
<td>55</td>
</tr>
<tr>
<td>5.12</td>
<td>Computation Time for Automotive01</td>
<td>55</td>
</tr>
<tr>
<td>A.1</td>
<td>Reduced Interactions per Configuration of axTLS</td>
<td>74</td>
</tr>
<tr>
<td>A.2</td>
<td>Reduced Interactions per Configuration of BerkeleyDB</td>
<td>74</td>
</tr>
<tr>
<td>A.3</td>
<td>Reduced Interactions per Configuration of Breakfast</td>
<td>75</td>
</tr>
<tr>
<td>A.4</td>
<td>Reduced Interactions per Configuration of E-Shop</td>
<td>75</td>
</tr>
<tr>
<td>A.5</td>
<td>Reduced Interactions per Configuration of FinancialServices01</td>
<td>76</td>
</tr>
<tr>
<td>A.6</td>
<td>Reduced Interactions per Configuration of Linux</td>
<td>76</td>
</tr>
<tr>
<td>A.7</td>
<td>Reduced Interactions per Configuration of Soletta</td>
<td>77</td>
</tr>
<tr>
<td>A.8</td>
<td>Number of Needed Configuration of Linux</td>
<td>78</td>
</tr>
</tbody>
</table>
## List of Tables

2.1 Occurred Feature-Interaction Bugs by Abal et al. [38] .......................... 11  
5.1 Feature Models Chosen for Evaluation ................................................. 46  
5.2 Number of Variant Features and Potential Interactions for $t = 2$ per Model 47  
A.1 Number of Needed Configurations per Algorithm ................................. 77  
A.2 Computation Time per Algorithm in Seconds ....................................... 79  
A.3 Computation Time per Configuration .................................................. 79
1. Introduction

One of the most challenging aspects of software engineering in the context of software product lines is to make software easy to configure. Software product-line engineering defines a set of systems that share common functionality and code artifacts to achieve high reusability and maintainability [21]. The concept of software product lines is used to manage configurable software where the functionality is determined by configurations [3]. These configurations consist of a set of features that can be either selected or unselected for each software variant. On the basis of a feature selection, all the required artifacts are combined to create the deliverable software product [7]. Therefore, different features can be combined to a configuration under the adherence to certain constraints, which can be represented, e.g., in a feature model [7]. A feature model represents features of a software product line in a tree structure by representing the possible combinations of features that lead to possible configurations [7]. One goal of feature-oriented software product-line engineering is to derive a product as an output automatically from variable code artifacts based on a feature selection input[21].

When different features are combined, it must be ensured that they do not cause software misbehavior and that all product tests pass. A feature interaction describes the situation in which a feature influences another feature such that the combined behavior of two features differs from the individual behaviors of both features in the system [35]. More precisely, the behavior of the system no longer matches the desired behavior. The interaction of multiple features or even one single feature can cause unforeseen faults and security vulnerabilities, which are some of the reasons why it is important to find these feature interactions [35]. It is difficult to determine and fix the unwanted interactions between features, because the exponentially growing number of possible interactions in the number of features makes an exhaustive search unfeasible [21]. Therefore, it is difficult to identify feature interactions due to the potentially large number of interactions. Most works deal with how to generate a suitable sample, but only few works deal with the identification process of the fault. Therefore, the developer is on his own when finding the origin of the fault that leads to the failing configuration. In summary, feature interaction detection is still a open problem, because there is no general, scalable, and product-ready strategy, yet [21]. Basically, the aim of configurable software developers is that every unwanted feature interaction in the software system is fixed.
or documented [46]. This problem of interactions between multiple features has been a challenging issue for decades [39].

Moreover, for a product line with \( n \) optional features there are up to \( 2^n \) possible configurable software products, that can be created. Especially, quality assurance for highly-configurable large systems is challenging because of the large configuration space [35]. Therefore, it is usually not feasible to test all these possible configurations of a system [33], let alone even generate all possible configurations. That is why sampling strategies have been developed to test small representative groups of products [33, 43]. These strategies, for example \( t \)-wise interaction sampling, aim to generate a as small as possible set of configurations that cover every possible configuration options of \( t \) features [43]. When testing these products and analyzing the corresponding configurations, our aim is to find the feature interaction or the feature interactions that lead to the failing configuration.

A widely used strategy is combinatorial interaction testing, which is a technique that constructs these samples for systematic testing of configurations [35]. Instead of testing every possible configuration, all combinatorial possible combinations of \( t \) features should be covered in at least one configuration. Furthermore, the idea is to select a representative subset of all possible configurations where feature interaction failures are more likely to occur [29, 40]. Therefore, the goal is to find the erroneous configurations when exhaustive testing of all possible configurations is not feasible.

While with current testing strategies one can identify failing configurations, it is challenging how to identify the feature interaction actually causing the fault. With the purpose of detecting this feature interaction that leads to the faulty configuration, we want to isolate the feature interaction or at least narrow down the set of potential interactions as far as possible. Often it is not possible to isolate features because some features have dependencies to other features. Besides that, some features are designed to work together and can only occur together [35], like for example when they are part of an atomic sets, which means that these features always occur in the same configurations [7]. Therefore, the search for the feature or the multiple features that cause misbehavior through interacting is an open challenge (i.e., it is difficult to isolate the feature interaction). Moreover, feature interactions may occur only in specific configurations and are thus difficult to detect, for example when a \( t \)-wise feature interaction coverage of the product line is not reachable. Unexpected behavior can also occur when changing code of an unrelated feature at first glance [35]. So, in order to detect the origin of this misbehavior, one can manually search for the fault or create and test suitable configurations to narrow down the features included in the feature interaction. Hence, this thesis wants to tackle the problem of isolating feature interactions based on a underlying feature model and a failing configuration.

Opposed to combinatorial interaction testing, we want to perform a combinatorial interaction analysis, where we want to analyze the potential feature interactions that can cause the software fault. We present two algorithms to create new configurations, a random and a divide-and-conquer strategy. By creating new configurations, new information about the features involved in failing configurations is gathered, which is then used to continue with our analysis until we have isolated the feature interaction. If we have isolated the feature interaction then we get the corresponding feature assignments as result. In this work, the type of fault or error message that occurs is not considered,
and we also limit ourselves to a single feature interaction in a system that we try to isolate.

**Goal of this Thesis**

The goal of this thesis is to present the concept of combinatorial interaction analysis, where we analyze all potential feature interactions until we have isolated the faulty interaction based on the failing input configuration. Furthermore, we provide the first algorithms that isolate the faulty feature interaction or even narrow down the set of potential feature interactions when it is not possible to isolate the interaction. From the current manual process of searching for the feature interaction, we want to present the concept of having an underlying feature model and a failing configuration as an input and delivering the isolated feature interaction causing the fault as an output, as can be graphically seen in on the cover sheet of this work.

Several conceptual and algorithmic research questions will be important, as defined in the following.

- **RQ1**: How effective can we isolate the feature interaction that leads to the failing configuration?
- **RQ2**: How many configurations are needed to isolate the feature interaction?
- **RQ3**: How long does it take to isolate the feature interaction that leads to the failing configuration?

First, we analyze how often we can isolate the feature interaction with our two presented algorithms of configuration creation. Besides that, we are interested in the remaining set of potential feature interactions after our analysis. Did we completely isolate the feature interaction or do we have a set of interactions left? This research question refers to the effectiveness of our approach. Second, we are interested in the number of configurations needed to isolate the interaction in each algorithm. Surely, we want to compare both algorithms in terms of computed configurations and the related output. In particular, we want the check the efficiency of our concept of combinatorial interaction analysis and how the efficiency depends on the algorithm. Finally, we want to compare the run time of our analysis and the time needed for the isolation. Based on this, we want to find out which algorithm can isolate the interaction faster and which algorithm needs less configurations.

The focus of this work is on pair-wise interaction analysis, i.e., we concentrate on interactions between two features, but a generalization to a combination of higher order interactions is possible. The concept describes how it can work with higher feature interaction, but some parts of the implementation and the evaluation only focus on pair-wise feature interactions.

**Structure of the Thesis**

Following this introduction, in Chapter 2, we explain the background of the topic in order to have a basic understanding of the domain and the problem of feature interactions. In Chapter 3, we introduce the concept of combinatorial interaction analysis, the
underlying assumptions, its limitations, and the guarantees we can take. In Chapter 4, we provide an implementation for the concept. After that, in Chapter 5 the evaluations based on the research questions are shown. Furthermore, we discuss the results and we give a summary. In Chapter 6, we conclude related work to our thesis to differentiate this work from the existing one. In Chapter 7, we conclude the results of the thesis. In Chapter 8, we provide possible future work.
2. Background

This chapter gives an overview of the basic knowledge that is necessary for this thesis. General knowledge of the domain as well as special background are presented in order to be able to understand the presented concept of combinatorial interaction analysis.

In the following, we only focus on software product lines dealing with the concept of feature selection from a given feature model. First, we take a close look at the concept of feature models, configuration generation, and the basic idea of feature interactions. After that, sampling strategies and combinatorial interaction testing are presented briefly.

2.1 Feature Models

A feature variable, also called feature, is a characteristic or end-user-visible behavior of a software system [21]. Features are used in the software engineering process to illustrate commonalities and differences between different products in the product line. Customers select features to derive a product that fulfills their needs. Besides that, they are often used for communication between stakeholders, because of the readability. These features are mostly structured in a feature model to set the properties and dependencies of the features. A formal definition of a feature model is given in the following [49].

**Definition 2.1 (Feature Model).** We define a feature model \( M = (F, D) \) as a tuple, consisting of a set of features \( F \) and a set of dependencies (also referred to as constraints) \( D \) over \( F \).

- Let \( F \) be the universe of all possible feature variables.
- The feature set \( F = \{ f_1, \ldots, f_n \} \subset F \) is a finite set, containing all features of the feature model \( M \) with \( n \) being the total number of features.
- Based on \( F \), we define the universe of literals \( \mathbb{L} \), which represents all possible assignments of all feature variables in \( F \).
  
  - We define the functions \( \phi^+ : F \rightarrow \mathbb{L} \) with \( \phi^+(f) = f \) and \( \phi^-(f) = \neg f \) that respectively return the positive and negative literal for a feature (i.e., both possible assignments for a Boolean feature).
Further, we define the function $L : \mathbb{M} \rightarrow 2^L$ with $L(M) = \{\phi^+(f) | f \in F(M)\} \cup \{\phi^-(f) | f \in F(M)\}$ that maps a feature model to a set of literals such that for each feature $f \in F$, it contains a corresponding positive and negative literal. Where, $\mathbb{M}$ describes the set of all possible feature models.

- The set of dependencies $D = \{d_1, \ldots, d_m\} \in 2^L$ is a finite set, containing all dependencies (i.e., constraints) of the feature model with $m$ being the total number of constraints. Every dependency $d$ is a subset of the set of literals $L(M)$ and represents a disjunction of the contained literals.

- We denote the set of features for a given feature model $M$ with $F(M)$ and the set of dependencies for the model with $D(M)$.

A feature model represents a software product line in a hierarchical structure by representing the possible combinations of features or the valid configurations which lead to the final software product [21]. Feature models are often represented in feature diagrams. A feature diagram consists of a set of features represented by nodes and edges, as can be seen in Figure 2.1. A feature can either be concrete, like for example the feature Water, or it can be abstract, like the feature Drinks, which means in most cases that there is no code that belongs to this feature. Most of the time, abstract features act as an overview of the underlying child features. The feature on the top is called the root feature. Furthermore, a feature can either be mandatory, like the feature Drinks, or optional, like for example the feature Cornflakes. Besides that, there are alternative groups represented through a white angle, which means that exactly one of the features must be selected if the parent is selected, as can be seen in Figure 2.1 for feature Toast. Furthermore, a logical OR group can be represented through a filled angle, as can be seen by looking at the feature Drinks and its children. Moreover, a feature model is called void if the model represents no products (e.g., through constraints) [7].

Also, there can be any number of cross-tree constraints, which limit the number of possible configurations and therefore products that can be created in our product line. Cross-tree constraints are logical terms, which restrict some feature combinations. In Figure 2.1, we can see the cross-tree constraint that says whenever the feature Cornflakes is selected, it implies that the feature Milk has to be selected, too. Furthermore, a disabled feature (e.g., Vegan $\Rightarrow \neg$ Milk) or an equivalency can be part of a cross-tree constraint, too.
2.2. Problem Space and Solution Space

Configurations

When creating a software product with specific features of our product line, first we have to select the features we want in our product. We want to set each feature variable, represented through a literal, to either be selected or unselected. Our formal notation of a configuration is based on a set of literals that can be derived from our feature model [49].

**Definition 2.2 (Configuration).** We define a configuration as set of literals \( c = \{l_1, \ldots, l_k\} \in \mathbb{L} \), where \( k \) is the number of literals contained in the configuration.

- If a configuration \( c \) assigns the value true to a feature variable \( f \), it contains the positive literal of \( f \) (i.e., \( \phi^+(f) \in c \)). Analogously, if a configuration \( c \) assigns the value false to a feature variable \( f \), it contains the negative literal of \( f \), i.e., \( \phi^-(f) \in c \).

- A configuration \( c \) may never contain more than one literal of any feature variable, i.e., \( \forall f \in \mathbb{F} : \{\phi^+(f), \phi^-(f)\} \notin c \).

- A configuration \( c \) is complete, iff all features are defined to either true or false, i.e., \( \text{complete}(c, M) = \text{true} \) iff \( |c| = |\mathbb{F}(M)| \).

- A configuration \( c \) is partial, when the configuration is in a state, where not every feature is defined yet, i.e., not \( \text{complete}(c, M) \).

- A configuration \( c \) is valid, iff the configuration is complete and if it satisfies all dependencies \( D \) of the feature model \( M \), i.e., \( \text{valid}(c, M) = \text{true} \) iff \( \text{complete}(c, M) \land \forall d \in D(M) : c \cap d \neq \emptyset \). We denote the set of all valid configurations of a feature model \( M \) with \( \mathbb{C}(M) \).

When creating configurations we select or deselect features, but first we have to consider core and dead features. A feature is called core if there is no valid configuration for the model in which the feature variable is assigned the value false [49]. Analogously, a feature variable is called dead if there is no valid configuration for the underlying feature model in which the feature variable is assigned the value true [49]. Therefore, the core and dead features are always in the same selection state for every possible valid configuration. Besides that, any feature that is not core or dead is called variant feature.

When selecting features together, we can have direct and transitive dependencies between them that lead to so called atomic sets of features, which means, that they always appear in the same configurations and can be seen as a unit [7]. Besides that, features in the same atomic set always occur in the same number of valid configurations, because otherwise there would be at least one configuration containing only a subset of these features [45].

Based on that definition of a configuration, we will state our understanding of problem and solution space in the following.

2.2 Problem Space and Solution Space

The specific construction of software product lines leads to a separation of problem and solution space. When we create a product, the feature selection, that leads to a configuration, is made in the problem space and this configuration is tested in the solution space and forms the product, as can be seen in Figure 2.2.
Problem Space

As problem space we define the representation of the problems, features, requirements and views of stakeholders. So, features can be seen as domain abstractions that characterize the problem space. When creating a software product from our software product line, we first generate a configuration by selecting or deselecting every feature and checking if the configuration is valid. So, a set of selected feature leads to a configuration in our problem space [49].

When checking whether the configuration is valid, we must check every dependency in the feature model. To test partial configurations, for example to check if the current feature selection can lead to a valid configuration, we can also use SAT Solvers which receive as input a propositional formula that represents each feature by a boolean value and also documents all dependencies between the features by conjunctions and disjunctions. The result of the solver gives us the information if the current configuration can lead to a valid configuration, so far. As output we get a boolean value that shows us if our formula is satisfiable or not. For example, for the configuration \{Breakfast, Drinks\}, we would get the output false, because the feature model requires at least one element to be selected from the OR group Drinks.

When we select features in problem space we only need the underlying feature model and no source code or any integration tests, which means that our feature selection does not depend on the used implementation technique or feature code structure [21].

Solution Space

As solution space we define the perspective of developers and vendors with implementation artifacts such as classes and functions. It deals among other things with the design and implementation of the features or software modules. The solution space contains all software artifacts and other documents of the software product line in order to be able to generate the set of all possible products [49].
2.3. Feature Interactions

The mapping from problem space to solution space is represented by a variability mechanism, that generates a software product by combining a set of implementation artifacts derived from a configuration [21]. Note that the number of possible configurations in problem space does not need to match the number of derivable products in the solution space, but the number of products is limited to the problem space [47].

There are several implementation techniques when we want to create a deliverable product, for instance preprocessors or build systems, which assembles the configurable software product based on our configuration. To check if a valid configuration leads to a functional product in solution space, we test it through integration tests. Besides that, we define that we only have to deal with valid configurations in solution space, because before mapping from problem to solution space the valid configuration test is done in problem space [49]. When our tests fail, we have to find the software fault, which leads to the failing configuration.

2.3 Feature Interactions

On testing configurations in the solution space and a faulty product results, it is usually because certain features interact with each other and produce unwanted behavior in the software. Some features might work in some configurations but when combining them with specific other features, they can cause faults. A feature interaction between two or more features can be seen as an emergent behavior that cannot be easily deduced from the behaviors associated with the individual features involved [21]. In the literature it is often not clear whether feature interactions mean the wanted interaction of two or more features or the unwanted interaction that leads to faulty behavior of the software. In the following, we always refer to the unwanted feature interaction.

Interactions of features cause dependencies between features which is a critical point where software faults can occur. These dependencies can be divided in three types [21]. First, there are syntactic dependencies, where features use methods of other features. Second, kind of syntactic dependencies, which means features extend other features and third, semantic dependencies, which rely on a specific behavior specified in a feature. Moreover, features can be classified into dynamic and static feature interactions, where static feature interactions occur when a configuration is compiled and when, for example, a method is called that does not exist, and dynamic feature interactions occur when a compiled product is executed and cause unexpected behavior.

In a product line with \( n \) features, there can be \( \binom{n}{t} \) \( t \)-wise feature combinations. In particular, there are \( \binom{n}{2} = \frac{n(n-1)}{2} \) pair-wise feature combinations. A higher-order interaction, also called \( t \)-wise interaction, is an interaction that occurs when \( t \) features interact. For a specific \( t \) there are \( 2^t \cdot \binom{n}{t} = 2^t \cdot \frac{n!}{(n-t)!t!} \) number of \( t \)-wise interactions of \( n \) features. In particular, there are \( 2^2 \cdot \binom{n}{2} = 2n(n-1) \) pair-wise feature interactions. Note that for each combination of two features, there are four ways to combine them:

\[
A \land B \quad A \land \lnot B \quad \lnot A \land B \quad \lnot A \land \lnot B
\]

Furthermore, only valid combinations of features that appear in at least one valid configuration are of interest, so for example the feature interaction of the feature combination
of Cornflakes and \(\neg\)Milk of our example feature model in Figure 2.1 need not be considered when analyzing feature interactions.

Note that there can also be feature interactions that are connected by disjunctions, thus stating that the feature interaction consists of, for example, feature Water or feature Milk, i.e., Water \(\lor\) Milk, called complex feature interactions. In the following, we restrict ourselves to feature interactions connected by conjunctions.

As an example, we can have a valid configuration \{Breakfast, Drinks, Milk, Tea\} in problem space but a failing software product in solution space because of a feature interaction between Water and Tea, because Water defines a method \(f\) and the feature Tea wants to call it, therefore when feature Water is deselected, Tea cannot use the method and we get a compile error, as we can see in Figure 2.3. Here we have a syntactic feature interaction because of the feature combination Tea and \(\neg\)Water, which cannot be predicted in our feature model in problem space.

In Figure 2.4, we can see a runtime error that occurs when feature Water is deselected and Tea is selected, too. When running the program, method \(f\) divides by zero if the main method sets \(x\) to zero.
2.4. Product Sampling

As a real-world example, in Table 2.1, Abal et al. list the interactions of the bugs they studied. Their research focused on the four models Linux, Marlin, Busybox and Apache [38]. There are 98 bugs, 41 of which involve only one feature, 35 involve two features and 22 involve three or more features. These numbers indicate that many faults occur due to one-wise or pair-wise interactions. This may be an indicator that fewer interactions occur with multiple features involved or that we currently cannot detect higher order interactions well.

### 2.4 Product Sampling

A sample is a set containing configurations of a software product line [49]. Product sampling is a technique that is used to avoid exhaustive testing of products, that is often infeasible because of the huge solution space [40]. So, sampling can be seen as a mapping from problem to solution space. The idea is to create a representative small subset of all products instead of creating and testing all possible products. Choosing the right sampling algorithm for practical applications is challenging because each sampling algorithm fulfills other evaluation criteria to a different degree [44]. The key point is how to select the important and relevant products.

A first sampling strategy is to look at often delivered products, or products requested the most. Another sampling technique include expert and customer knowledge. This sampling algorithm, can include further knowledge in terms of partial configurations, which occur often or which contain security-critical feature interactions [40]. An example of a sampling algorithm, that can incorporate expert knowledge is YASA [43].

In the following we show some coverage criteria and algorithms to generate a representative sample of a product line.

#### 2.4.1 Coverage Criteria

A further sampling strategy is to select only a few representative number of products with a high coverage of some criteria. To name just a few, we list the coverage criteria presented by Apel et al. [21]:

<table>
<thead>
<tr>
<th>Interaction</th>
<th>L</th>
<th>M</th>
<th>B</th>
<th>A</th>
<th>∑</th>
</tr>
</thead>
<tbody>
<tr>
<td>some selected</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A$</td>
<td>5</td>
<td>6</td>
<td>3</td>
<td>7</td>
<td>21</td>
</tr>
<tr>
<td>$A \land B$</td>
<td>10</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>21</td>
</tr>
<tr>
<td>$A \land B \land C$</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>$A \land B \land C \land D \land E$</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>some-selected-one-deselected</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\neg A$</td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>$A \land \neg B$</td>
<td>13</td>
<td>3</td>
<td>4</td>
<td>-</td>
<td>20</td>
</tr>
<tr>
<td>$A \land B \land \neg C$</td>
<td>3</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>4</td>
</tr>
<tr>
<td>$A \land B \land C \land D \land \neg E$</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>other configurations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\neg A \land \neg B$</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>$A \land \neg B \land \neg C$</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>$A \land \neg B \land \neg C \land \neg D \land \neg E$</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
- **Feature Coverage**: select features such that every feature occurs in at least one configuration

- **Pair-wise Feature Coverage**: every pair-wise feature selection occurs in at least one configuration in the sample (look at pair-wise interaction testing)

- **T-Wise Feature Coverage**: every \( t \)-wise interaction of \( t \) features is covered by at least one configuration in the sample (look at \( t \)-wise interaction testing)

The procedure to get every possible combinatorial pair of \( t \) features is first, exclude invalid combinations like for example \( C \) and \( \neg M \) from our example feature model, because the cross-tree constraint \( C \Rightarrow M \). Second, exclude core features as well as dead features. Moreover, \( t \)-wise feature coverage is almost like pair-wise but with a \( t \)-tuple of features instead of only two features. Thus, \( t \)-wise coverage defines that every possible \( t \)-wise interaction of \( t \) features is covered by at least one configuration in the sample. Furthermore, when covering \( t \)-wise, one can automatically cover all possible interactions \(< t\), because they are included in the higher \( t \) combinatorial combinations.

Consequentially, if we only sample with the feature coverage criteria, then we have a lower probability to find feature interactions. In contrast, it is more expensive to cover a higher \( t \), because of the raising sample size.

Furthermore, one can include feature code coverage, which means that at least every feature code fragment from the solution space is included in at least one configuration [21].

### 2.4.2 Combinatorial Interaction Testing

Combinatorial interaction testing systematically samples the possible configuration possibilities and tests only the created configurations. The main idea is that this approach works by first defining a set of valid ways the configurations can be configured. Mostly, this approach contains a model, e.g., a feature model, which determines the configuration options through constraints. On the basis of these model, combinatorial interaction testing compute a small set of valid configurations by a sampling algorithm, mostly represented in a \( t \)-way covering array, in which each combination of \( t \) features appears in at least one [24]. After that, every configuration is tested. As an input model, we will only look at feature models, that represent this array as a set of feature literals, which can either be true or false.

This technique is used to reduce redundant effort in contrast to test all possible configurations and it is applicable to large software product lines. For a large \( t \), one often speaks of combinatorial explosion, because of the increasing combinatorial possibilities to combine features.

### One-Wise Interaction Testing

In Figure 2.5, we can see a simplified version of the feature model from the beginning, to explain the interaction testing ideas by an example. To make it easier to write down the interactions, only the initial letters of the individual features are used as representation in the following.
First, we want to check if one feature causes the unwanted behavior of our software product. Maybe it is the occurrence or the absence of one feature that leads to the fault. So for example if we have \( n \) optional features we need two configurations to select and deselect each feature. Looking at our example feature model, we have to cover the following ten feature interactions, where the interaction is represented through the selection or deselect of one feature with the code base of the core features.

\[
\begin{align*}
W & \quad \neg W \\
M & \quad \neg M \\
J & \quad \neg J \\
T & \quad \neg T \\
C & \quad \neg C
\end{align*}
\]

If we want to cover these interactions, the following two configurations would do so, for example:

**Configuration 1**: \( \{B,D,M,C\} \)

**Configuration 2**: \( \{B,D,W,J,T\} \)

This procedure can be used to test whether a bug is due to the presence or absence of a feature.

**Pair-Wise Interaction Testing**

When looking at pair-wise interaction testing, we want to test a sample set with all valid configurations with pair-wise coverage [8]. That means, that every pair-wise interaction is covered by at least one configuration in the sample. The following pairs should be covered, based on the feature model example from Figure 2.5.
The combination of $\neg M \land C$ is not possible, because of the cross-tree constraint. This are possible six valid configurations of our feature model example that cover each pair-wise feature interaction:

**Configuration 1:** \{B, D, W, M, J, T, C\}

**Configuration 2:** \{B, D, W\}

**Configuration 3:** \{B, D, M, J, T\}

**Configuration 4:** \{B, D, M, C\}

**Configuration 5:** \{B, D, J\}

**Configuration 6:** \{B, D, T\}

Looking at the pair-wise coverage, the blue marked pairs are first covered for configuration 1 and so on:

A simple heuristic for pair-wise interaction testing is a greedy procedure that always chooses the configuration that covers the most missing pairs next. This is not necessarily optimal because we do not always have the smallest number of configurations.


T-Wise Interaction Testing

When considering \( t \)-wise interaction testing, we first consider \( t = 1 \) as interaction size, then every feature is selected and deselected in at least one configuration in the sample. Second, for the interaction size \( t = 2 \) is referred to as pair-wise interaction testing and last, we can assume a higher value of \( t \), so \( t \)-wise interaction testing is a generalization of one-wise and pair-wise interaction testing. For \( t = 3 \) every combination of three features has to be covered, for example, the features \( W, M, \) and \( J \) have eight potential 3-wise interactions:

\[
\begin{align*}
    W \land M \land J \\
    \neg W \land M \land J \\
    W \land \neg M \land J \\
    W \land M \land \neg J \\
    \neg W \land \neg M \land J \\
    \neg W \land M \land \neg J \\
    W \land \neg M \land \neg J \\
    \neg W \land \neg M \land \neg J
\end{align*}
\]

Therefore, we have \( 2^t \) possible \( t \)-tuples for \( t \) independent optional features. Note that for \( t \)-wise coverage, all smaller than \( t \) interactions are covered as well, as already mentioned before. In generally, the larger the selected \( t \), the more faults can theoretically be found. In particular, if the \( t \) is equal to the number of features, then we can find all faults with suitable test cases.
2 Background
3. Combinatorial Interaction Analysis

In this chapter, we present combinatorial interaction analysis as our approach to isolate faulty feature interactions based on known correct and failing configurations. Under the assumption that there is only one faulty feature interaction, we analyze the set of all potential feature interactions and eliminate candidate interactions step by step until the feature interaction causing this fault is isolated. Our elimination procedure utilizes our knowledge of correct and failing configurations, we assume that feature combinations are not faulty when they appear in a correct configuration. We want to use the concept of a black-box setting with perfect fault information.

In contrast to combinatorial interaction testing, we provide an approach to analyze the potential feature interactions and narrow down the set of potential interactions until we have isolated the feature interaction. In the concept of combinatorial interaction testing, one tests configurations that cover combinations of features for a given interaction size \( t \) to see if there are configurations that fail in solution space. In comparison, we present the first concept to isolate the origin of the fault. We want to investigate combinatorial feature interactions that are likely to cause the software fault and identify the origin of the fault by systematically excluding the remaining feature interactions from the set of all potential feature interactions. In summary, based on a failing configuration, we want to analyze the feature interactions to find the smallest feature combination that is at least necessary to cause the fault, as can be seen graphically in Figure 3.1. With this analysis we want to provide a concept to support developers in debugging.

This chapter is structured in six sections. First, we present a definition of a feature interaction in Section 3.1, to share our understanding of a feature interaction. With some assumptions presented in Section 3.2, we narrow down the use cases to which our approach is applicable. In Section 3.3, we present the concept of combinatorial interaction analysis and two strategies for creating new configurations. Besides a random strategy, we want to show a divide-and-conquer strategy to split the set of potential feature interactions and create new configurations containing about half of the interactions. In Section 3.4, we extend our approach by including further information, such as expert knowledge, in the analysis. Next in Section 3.5, we discuss our approach and give some guarantees and limitations. In Section 3.6, we provide a summary about our approach of
isolating the feature interaction causing the failing configuration through combinatorial interaction analysis.

\section*{3.1 Definition of Feature Interactions}

In our application case, we work in the context of software product lines with the concept of features represented in a feature model. Therefore, configurability is presented in form of combined features when creating new software products. Here, we can have unwanted interactions between several features that can lead to software faults. As there are different definitions of feature interactions in the literature and none of them meets exactly our requirements to consider only interaction with conjunctions of literals, we give a new definition in the following.

\begin{definition}[Feature Interaction] A \(t\)-wise feature interaction \(I\) is a partial configuration that causes unwanted software behavior. It can be represented through a set of literals \(c = \{l_1, \ldots, l_t\} \in \mathbb{L}\) for the feature interaction size \(1 < t \leq n\), whereby \(n\) represents the number of all features \(F\) of the underlying feature model \(M\).

- Each literal stands for a feature in a feature interaction \(I\), which can either be selected (positive literal) or deselected (negative literal).
- Each interaction represents a conjunction of the represented literals.
- We define the set of all possible feature combinations of \(t\) features as the set of all possible \(t\)-wise feature interactions \(\mathcal{I}\).

For the interaction size \(t = 1\), we define a one-wise feature interaction of a feature model \(M\) containing one selected or deselected feature (i.e., one positive or negative literal). Therefore, for the interaction size \(t = 2\) we define a pair-wise feature interaction of a feature model \(M\), e.g., \(I = \{l_1, \neg l_2\}\). Analogously, for a interaction size \(t \geq 3\) we define a higher-order feature interaction of a feature model \(M\) as a set of \(t\) positive or negative literals connected with a conjunction. Of course, due to the underlying feature model,
particular feature combinations may not be valid. In particular, some interactions from \( \mathcal{I} \) cannot be covered, as they conflict with the dependencies \( D(M) \) from the feature model \( M \). We give a definition of an valid feature interaction in the following.

**Definition 3.2 (Valid Feature Interaction).** An interaction \( I \) is valid, if there exists a valid configuration \( c \) for the feature model \( M \), which contains \( I \), i.e., \( \exists c \in C(M) : i \subseteq c \).

Additionally, regarding feature interactions, we limit our considered feature set to not contain core and dead features. That is because we assume that a pair-wise feature interaction between a core or dead feature and a variant feature could be analogously represented through a one-wise interaction by the variant feature. For example, if we have an unwanted interaction between a core feature \( A \) and variant feature \( B \), then we do not consider the occurring interaction as a pair-wise feature interaction \( A \land B \), but as a one-wise feature interaction with \( B \). Furthermore, one can assume that core or dead features can be omitted in problem space, due to the non-existing variability of these features in the product line.

Furthermore, features can have dependencies between each other. For example, when \( A \) is equivalent to \( B \), \( B \) is equivalent to \( C \), and \( C \) is equivalent to \( A \). Thus, the assignments of these three features are dependent, which means that this partial configuration is an atomic set, as described in Section 2.1. Therefore, these features selected in an atomic set cannot be assigned to true or false independently from each other in any valid configuration. Therefore, we consider all features of an atomic set as one participant in an interaction.

### 3.2 Assumptions

Before introducing our approach, we aim to make some conceptual assumptions under which conditions our approach is feasible. Our concept of combinatorial interaction analysis is built on the following assumptions, which we make explicit.

**A1:** The feature model has at least one valid configuration that contains the feature interaction.

**A2:** The feature model has at least one valid configuration that does not contain the feature interaction.

**A3:** There is exactly one feature interaction in our system and the occurring fault is exactly reproducible.

The first assumption **A1** ensures that our underlying feature model is not void. This ensures that at least one valid configuration of our product line can be automatically generated that contains the fault and therefore the feature interaction. Automatic generation and testing of the configurations is important, otherwise our approach cannot be performed without further effort, for example if one would have to write glue code first [21].
Moreover, the assumption \textit{A2} refers to the fact that we require at least one valid configuration of our product line to not contain the fault. More precisely, the fault does not occur for every configuration. This offers us the possibility to not consider core and dead features, since these are included in every valid configuration. Therefore, the set of features we have to consider contains only variant features.

Besides that, we assume in our analysis that the occurring fault is deterministic and does not occur sporadically, which means that the fault only occurs under the same general conditions and is reproducible. Therefore, the assumption \textit{A3} is important, to be able to identify the faulty feature interaction exactly. The fault could be a compiler or a runtime fault, as long it is the same failing test at the end. In particular, we assume that there are no race conditions. In this thesis, we will not go further into the different types of faults. Furthermore, by this assumption, we want to constrain that there is only one uniquely identifiably feature interaction that causes a software fault. This is important to be able to exclude feature interactions that are not covered in the considered configurations from our set of the remaining potential feature interactions. In summary, we can reliably determine for a given configuration whether it contains the fault or not.

### 3.3 Concept of Combinatorial Interaction Analysis

In the following, we show the concept of combinatorial interaction analysis, where we want to isolate the feature interactions (short interactions) in the following, based on a failing configuration. During the analysis, new configurations are created which are then used for further analysis to gain more information about when the software fault occurs. Our analysis takes place entirely in problem space, whereby the configurations needed for the analysis are tested in solution space.

To apply our concept we require an initial sample containing one failing configuration $c_{fail}$ and one correct configuration $c_{corr}$ from our feature model as input, as depicted in Figure 3.2 in Step 1. This is intended to address the assumptions \textit{A1} and \textit{A2}. When searching for the feature interaction we start by the interaction size $t = 1$ to search for the feature interaction causing the fault (Step 2). In Step 3, we can proceed by creating a set containing all potential feature interactions for the interaction size $t$. Next, we create (Step 4) and test new configurations, where we analyze the interactions contained in the considered configuration and update (Step 5) the failing and correct configurations. If the new configuration fails, we can exclude all interactions that are not included because of \textit{A1}. Similarly, we can exclude all interactions that are included in correct configurations. After going through the loop of Step 3 till 5 and after we considered all potential interactions of the interaction size $t$ and no faulty feature interaction is identified, we increase the interaction size to the next higher $t$ (Step 7). Then we continue this loop until we have identified the interaction, run into a timeout, or reached our maximum interaction size $t_{max}$, which is $|F(M)|$ at most. Then, the output is either the isolated interaction or a set of remaining interactions (Step 6). Moreover, we can set this maximum interaction size for example at $t_{max} = 4$, because there are very few known interactions so far that have contained more features [38]. Furthermore, we could only proceed the analysis for a specific interaction size $t$ if we want to search for a specific $t$-wise interaction. Thus, we can leave out the Step 7.
3.3. Concept of Combinatorial Interaction Analysis

Figure 3.2: The Steps of Combinatorial Interaction Analysis
In the following, we show how we compute all potential interactions considered in our analysis. After that, we will explore two strategies of configuration creation when performing combinatorial interaction analysis. First, we provide a random strategy that analyzes random configurations to get more information about which feature interactions can be responsible for the occurring fault. Second, we present a divide-and-conquer strategy, that systematically creates new configurations containing about half of the potential remaining interactions to eliminate potential interactions step by step.

3.3.1 Compute All Potential Feature Interactions

Instead of calculating all possible interactions provided by the whole feature model, we take advantage of the assumptions from the previous section and start our analysis based on the failing configuration $c_{\text{fail}}$ and reduce the set of potential interactions significantly, as shown in Figure 3.3. Here, we see the comparison of the set of all possible pair-wise interactions and the actual potential pair-wise interactions we have to consider. Moreover, this tendency is adaptable for higher-order interactions. This gives us the possibility of excluding about 75% of all interactions in our analysis from the beginning. Note, that dependencies of feature models are omitted here. Therefore, the all-interaction curve represents an upper bound.

The correct configuration $c_{\text{corr}}$ from our input data can also be seen as a further reduction of the set by removing all interactions contained in this configuration from the set of all potential interactions. If the correct configuration has no common interactions with the failing configuration, then the starting set corresponds to the lower curve in Figure 3.3. If the correct configuration contains common interactions with the faulty configuration, we can exclude these interactions initially and start with a lower curve, i.e., a smaller set of potential interactions. Therefore, the lower curve can even be considered as the upper bound of the starting set. For an interaction size $t \in \{1, \ldots, n\}$ we have therefore the set
of all potential $t$-wise interactions for all features $n$. This set cannot be empty, because at least the interaction we are looking for is in $c_{fail}$ and not in $c_{corr}$, as defined in the assumptions in Section 3.2 (i.e., A1 and A2). In particular, for $t = 1$ we have all literals contained in the failing configuration and not in the correct configuration.

### 3.3.2 Random Strategy

In this section, we explain the combinatorial analysis when choosing randomly generated configurations. Based on these configurations we want to examine and narrow down the set of all potential interactions by looking in which configurations the interaction occurs and in which ones it does not. When creating a new configuration, we ensure that it is different from all previously generated configurations.

In the Example 3.1, we show the random approach based on our running feature-model example that can be seen in Figure 2.5 from Section 2.4.2. Note that when we write down configurations in an example, we only write down the variant feature names abbreviation by their first letter and leave out the core and dead features because they are independent of the individual configuration.

**Example 3.1: Random Strategy**

We start the analysis for the feature interaction size $t = 1$ by our failing input configuration

$$c_{fail} = \{M, J, T\}$$

and create the set of all potential interactions

- $W \not\in W$
- $M \not\in M$
- $J \not\in J$
- $T \not\in T$
- $C \not\in C$

Next, we consider our correct input configuration

$$c_{corr} = \{W\}$$

With this information we have the set of all possible one-wise interactions reduced to
Assume that the first randomly generated configuration is

\[ c_1 = \{M, J, T, C\} \]

and that it does not fail. Therefore we have randomly covered all remaining interactions and no interaction isolated. Now we have to increase the interaction size \( t \).

Therefore, we increase the interaction size to \( t = 2 \). Because the feature \( W \) is deselected in our failing configuration \( c_{\text{fail}} \) and we have the assumption that we have exactly one interaction that causes the fault, the fault cannot be caused by an interaction in which \( W \) is selected, therefore we can exclude all interactions containing feature \( W \), and so on. Furthermore, we can look at the correct configuration \( c_{\text{corr}} \) and exclude all interactions contained, because these cannot be the faulty feature interaction. In this case, we cannot exclude more interactions because our \( c_{\text{corr}} \) does not have any interactions with \( c_{\text{fail}} \) in common. Therefore, we can establish the following ten feature interactions of size \( t = 2 \) that could be the reason for the fault:

\[
\begin{align*}
\neg W \land M & \quad \neg W \land J & \quad \neg W \land T & \quad \neg W \land \neg C & \quad M \land J \\
M \land T & \quad M \land \neg C & \quad J \land T & \quad J \land \neg C & \quad T \land \neg C
\end{align*}
\]

In the next step, we generate a new randomly created configuration, to see if the fault still occurs. Assume we generate the following configuration

\[ c_1 = \{M, T\} \]

and we assume \( c_1 \) fails. Thus, we can exclude the grayed out feature interactions to not be the faulty one, because the considered configuration \( c_1 \) do not contain these feature interactions. For example the interaction \( M \land J \) is not part of \( c_1 \), thus we can exclude it for our further analysis. So, the black feature interactions remain to be the faulty interaction:

\[
\begin{align*}
\neg W \land M & \quad \neg W \land J & \quad \neg W \land T & \quad \neg W \land \neg C & \quad M \land J \\
M \land T & \quad M \land \neg C & \quad J \land T & \quad J \land \neg C & \quad T \land \neg C
\end{align*}
\]

Next, we generate another configuration to analyze if we can exclude more interaction pairs. Assume, we have the configuration

\[ c_2 = \{M, J, C\} \]
and assume that this one does not fail. Now, we can exclude one further interac-
tion pair, i.e., $\neg W \land M$, because it is part of our new correct configuration and can
therefore not be the feature interaction we are searching for.

$$
\begin{align*}
\neg W \land M & \quad \neg W \land T \\
M \land T & \quad M \land \neg C \\
T \land \neg C
\end{align*}
$$

Next, we check another configuration. For example, we generate the configuration

$$
c_3 = \{M, W, T\}
$$

and we assume that this one fails. Therefore, we can exclude further interaction
pairs from our set of potential feature interactions:

$$
\begin{align*}
\neg W \land T & \quad \neg W \land \neg C \\
M \land \neg C & \quad T \land \neg C
\end{align*}
$$

Next, we take a further randomly generated configuration

$$
c_4 = \{M, C\}
$$

and assume that it does not fail. Here, we cannot exclude any further interaction
pair, because $c_4$ does not contain any of the remaining interaction pairs. After that,
we continue to generate a new randomly generated configuration

$$
c_5 = \{M\}
$$

and check if it fails. We assume it fails and so we can exclude two out of three
remaining interactions:

$$
\begin{align*}
M \land T & \quad M \land \neg C \\
T \land \neg C
\end{align*}
$$

Now we have isolated one potential interaction, we can assume that the interaction
we are searching for is a pair-wise interaction. Based on our interaction analysis we
now have isolated the pair-wise feature interaction $M \land \neg C$.

This approach can potentially require many configurations and thus, take a long time to
isolate the feature interaction, since some configurations (e.g., $c_4$) do not deliver new
knowledge to exclude feature interactions from the potential feature interaction set.
More precisely, one can be unlucky when choosing the configurations and in the worst
case no further interactions can be eliminated. This means that one may have to test
an amount of configurations, which is not feasible to test because of the large configu-
ration space. Because of this problem, we want to present a further strategy to create
configurations more deliberately.
3.3.3 Divide-And-Conquer Strategy

In this section, we introduce a further strategy, which uses the principle of divide-and-conquer to isolate the feature interaction. When creating new configurations, we want to create them deliberately instead of randomly. The newly-created configuration should contain about half of all potential feature interactions each step. Here, we want to take advantage of the fact that if we create a new configuration that still fails, we can exclude all feature interactions that are not included in this configuration, because of the assumption A3 from Section 3.2 that defines that we have only one uniquely identifiable feature interaction. Analogously, if we create a new configuration that does not contain the interaction we can exclude all feature interactions that are included in this configuration. Hence, in each case, we can exclude about half of the remaining feature interactions.

Creation of Configurations

When creating the new configuration, we can proceed as follows. First, we set all core features to true and all dead feature to false since these must be set in every valid configuration. After that, we split the remaining potential feature interactions in two sets that each contain about half of the interactions. This procedure can be done in several ways, for example, we can first merge all features of the same atomic set and after that include only interactions that contain the same features. In this way, we want to make sure that we do not merge interactions that lead to invalid configurations. After splitting the remaining interactions, we want to create a configuration which contains about half of the interactions, as shown in Figure 3.4.

Another way to create the new configuration is by including a first feature interaction from our set of the remaining interactions and check if the resulting partial configuration is still valid, as displayed in Figure 3.5. This step also sets all other dependent features that this feature interaction selection implies in our partial configuration. It is possible that we include other feature interactions greedy from our set automatically, that is why we have to update our interactions contained in our partial configuration. Furthermore, it is possible that a selected feature interaction would lead to an invalid configuration, which is when we exclude this feature interaction to be part of the configuration that we
create at the moment. After that, we proceed by including feature interactions greedy from our set until we have covered about half of the interactions. Here we have to consider the problem of adding any interaction to the configuration that implies too many further interactions. If this happens, we may think of adapting the greedy procedure by excluding a previously selected interaction and select other interactions until we get to about half.

In Example 3.2, we show the divide-and-conquer approach using the same feature model from Figure 2.5 for the random strategy. Note that, as in the previous example, the core features are omitted from the example configurations. As before, we assume that we did not found a faulty feature interaction for \( t = 1 \), therefore we continue to search for a feature interaction for the next higher interaction size \( t = 2 \).

**Example 3.2: Divide-And-Conquer Strategy**

We start the analysis for the interaction size \( t = 2 \) by our failing input configuration

\[
c_{\text{fail}} = \{M, J, T\}
\]

and our correct input configuration

\[
c_{\text{corr}} = \{W\}
\]

Because the feature \( W \) is deselected in our failing configuration \( c_{\text{fail}} \) and we have the assumption A3 that we have exactly one interaction that causes the fault, the fault cannot be caused by an interaction in which \( W \) is selected, therefore we can exclude all interactions containing feature \( W \), and so on. Therefore, we can establish the following ten feature interactions of size \( t = 2 \) that could be the reason for the fault:

\[
\neg W \wedge M \quad \neg W \wedge J \quad \neg W \wedge T \quad \neg W \wedge \neg C \quad M \wedge J \\
M \wedge T \quad M \wedge \neg C \quad J \wedge T \quad J \wedge \neg C \quad T \wedge \neg C
\]
As a next step we want to create a new configuration that contains about half of these potential feature interaction pairs. Let us take the configuration

\[ c_1 = \{J, T\} \]

and test if it fails. We assume that \( c_1 \) does not fail, therefore we can exclude all interaction pairs that are contained in our considered configuration:

\[
\begin{align*}
\neg W \land M & \quad \neg W \land J \\
M \land T & \quad M \land \neg C \\
\neg W \land \neg C & \quad M \land J \\
\end{align*}
\]

Next, we create a new configuration containing half of the remaining interactions

\[ c_2 = \{W, M, T\} \]

and test if it fails, too. We suppose \( c_2 \) fails, so we can exclude all remaining feature interactions that are not contained in this configuration:

\[
\begin{align*}
\neg W \land M & \quad M \land \neg C \\
\neg W \land T & \quad J \land T \\
\end{align*}
\]

Next, based on the remaining two pairs, we have to create a further configuration which contains one interaction, so for example

\[ c_3 = \{M\} \]

Suppose, we get the result that it fails, then we can exclude the other interaction:

\[
\begin{align*}
M \land T & \quad M \land \neg C \\
\end{align*}
\]

Now we have isolated the feature interaction. Thus, as we can see, the only feature interactions that remains is \( M \land \neg C \). In summary, we have isolated the feature interaction with three new configurations through the strategy of including about half of the interactions per new configuration.

In this example, we can see that we can exclude about half of the potential interactions in each step. If the remaining interactions can always be divided into two sets each containing about half of the interactions, then the strategy is similar to an algorithm that follows a binary-search principle and thus provides a good approach to reduce the configurations to be considered. Note that the creation of configurations depends on dependencies between the feature interactions and it is possible that we have to create more configurations than desired.
3.4 Including Further Information

In this section, we discuss the possibility to have more input knowledge than before to analyze the failing configuration. In Section 3.4.1, we show the advantages of a whole sample as input at once instead of just one failing configuration. We can include these other configurations that fail or succeed to get more information about the interactions that can be responsible for the software fault. We can also prioritize feature interactions, which is presented in Section 3.4.2. So, if we want to create new configurations to isolate the faulty feature interaction, instead of just examining the individual potential \( t \)-wise interactions, we can also prioritize the respective interactions with some further knowledge.

3.4.1 Including More Than Two Configurations

Instead of an input of one failing configuration and one correct configuration, we can consider a sample \( S \) containing at least one failing configuration and one correct configuration with more than two configurations (i.e., \(|S| > 2\)). This offers us more information about several feature interactions in our system, which can be used for our analysis to reduce the set of the potential feature interactions.

There is the possibility that more than one configuration in the sample fails or only one. When we have a sample \( S \) containing a set of configurations \( C \) that contains at least two failing configurations, we can build the set of feature interactions that can possibly be the faulty feature interaction. We build a subset of all feature interactions for each interaction size \( t \) that occurs in all failing configurations, i.e., the intersection of these configurations. It follows that if we have more than one configuration that fails, we can exclude all \( t \)-wise feature interactions that are not in all of these failing configurations. Moreover, we can have a sample \( S \) containing more than one correct configuration. This offers the possibility of excluding further interactions from the set of all potential interactions, i.e., all interactions that are contained in any correct configuration.

With this concept, we have the possibility to use the combinatorial interaction approach to find the feature interaction without even having to construct further configuration. This can happen in the simplest case when we have two configurations that both fail but have only one feature interaction in common. So, in the particular case where there is only one feature interaction that is included in each faulty configuration, we can identify the faulty feature interaction directly. Thus, in the best case we do not need to create any new configurations at all. Furthermore, when increasing the interaction size \( t \), we can include the configurations that are generated for all lower interaction sizes. This provides a first optimization.

In Example 3.3, we present a short pair-wise interaction example in which the advantage of more configurations as input is presented.

Example 3.3: Input Sample

Let us take a look at the failing configuration from the example below with the size \( t = 2 \). With the failing configuration:

\[
 c_{\text{fail}} = \{M, J, T\}
\]
these are the potential interaction pairs which can cause the fault:

\[
\begin{align*}
\neg W & \land M \\
W & \land T
\end{align*}
\]

We assume we have a sample containing \( c_{\text{fail}} \) and the following two configurations

\[
c_{\text{corr}1} = \{ J, T \} \quad \text{and} \quad c_{\text{corr}2} = \{ M, J, T, C \}
\]

We can assume that the interaction we are searching for is a pair-wise interaction. Therefore, we can exclude all pairs but one through analyzing the sample without even creating a new configuration:

\[
\begin{align*}
\neg W & \land M \\
W & \land T
\end{align*}
\]

### 3.4.2 Including Expert Knowledge

In the following, we show some advantages of including expert knowledge to our analysis. As expert knowledge, we consider more detailed information about the individual features and the scenarios where features can interact with each other. This offers us more information about several interactions, which can be used for reducing the set of all potential interactions even further.

When creating new configurations, we can take advantage of the fact that software bugs occur more often with a combination of selected and deselected features. Hence, this can be beneficial, as Abal et al. discovered that most interactions occur in these two interaction groups [38], as presented in Section 2.3. So, we can prioritize feature interactions with at least one selected feature, i.e., we can pay less attention the feature interactions with only deselected features at the start of analysis.

Moreover, for creating new configurations, feature interactions can be chosen that are included in many products or specific features can be included that are chosen frequently. In addition, safety-critical features can be included in the analysis first to be part of the early creation of new configurations. This offers the possibility to exclude these features to be contained in the interaction before the analysis is finished.

In Example 3.4, we present a short pair-wise feature interaction example in which the prioritization of at least one selected feature in a feature interaction is presented.

**Example 3.4: Prioritize Feature Interactions**

We can assume the scenario of the failing configuration from our running feature-model from the previous examples. Let us suppose, the following interactions pairs are the remaining potential feature interactions:
Here, we can assume that we do not split the half of the interactions, but only take a look at the one-selected interactions. We have the possibility to isolate the feature interaction with creating only one further configuration containing either the interaction $\neg W \land M$ or the interaction $M \land \neg C$. Thus, if it is a pair-wise interaction that we are looking for, we can isolate the faulty interaction with only one further configuration.

\[\neg W \land M \quad M \land J \quad M \land T \quad M \land \neg C\]

3.5 Discussion

In the following, we discuss our approach of combinatorial interaction analysis and its applicability. We explain the weaknesses and strengths of the two strategies to create configurations and why the assumptions were defined this way.

With our given definition of a feature interaction we want to deliver a generally applicable definition that defines the feature interaction as an unwanted interaction. We use it to replace the many different previous definitions of feature interactions in the domain of software product lines, because some researchers also refer to the wanted interaction between features in the context of feature interactions. Another controversial definition is the minimal feature interaction size. Here, most researchers start with a feature interaction between at least two features to call it feature interaction. However, we also want to consider feature interactions between a feature and the code base as a one-wise feature interaction. Therefore, we can omit the core and dead features in our analysis, since an interaction between them and a variant feature can be expressed by a one-wise interaction of the variant feature. For this rational, our assumption \textbf{A2} is important, which says that there is a configuration in which the fault does not occur, because this allows us to state that the core and dead features are out of scope. Interactions that occur caused by these features can be detected for example by white-box analyses.

Furthermore, when checking for feature interactions, we consider both abstract and concrete features. We could have the possibility to exclude more features from our analysis if we would only take a look at the set of concrete features because abstract features do not need to be handled, since typically no code exists for them [16].

When defining our assumptions \textbf{A1} and \textbf{A2} it was important to show that our underlying feature model is not void (i.e., to be able to derive products from the model) and that we have at least one correct configuration to exclude all interactions with core and dead features to be faulty. With the assumption \textbf{A3}, we restrict the use case in which we can have several faults in our system, which cannot be guaranteed. This restriction is important to proceed in our analysis like in a binary-search strategy to exclude sets in which the fault is not contained. Thus, we want to be able to make statements about the considered interactions and the not considered interactions whether the fault is still contained.

Note that we do not verify the found feature interaction, which is outside the scope of this thesis. However, this can be possible by, for example, checking the next higher interaction size \(t\).
Guarantees

Our concept of combinatorial interaction analysis offers certain guarantees, which are presented in the following. With respect to the assumptions from Section 3.2, we can guarantee that we get at least a set of interactions that contains the faulty feature interaction in our system and that the previously excluded interactions work correctly. In the worst case we can only exclude the interactions occurring in the correct configuration from our input. In particular, we sometimes cannot create new configurations that include or exclude certain interactions due to dependencies between features. However, if all features are independent and optional we can guarantee that we can always isolate the interaction that is responsible for the fault.

For the random strategy of creating configurations, we have the guarantee that at least when we have tried all configurations we have fully narrowed down the set of potential interactions. Testing all possible configurations is exactly what we do not want to do and is usually infeasible [21]. In particular, we might test configurations that do not give further evidence. That is why we only generate new configurations that give us new information about at least one remaining potential interaction in the split algorithm, compared to the random algorithm.

Limitations

Now, we show the limitations of our idea of combinatorial interaction analysis. Our approach is only applicable under the assumptions defined in Section 3.2. With the restriction to only have one deterministic fault in our system, we are currently not able to guarantee that the concept is applicable to scenarios in which multiple feature interactions cause a software fault.

Moreover, if we restrict our analysis to a maximum interaction size $t_{\text{max}}$ then we cannot find the faulty interactions with a higher interaction size $t > t_{\text{max}}$. For example, the feature interactions $A \land B$, $B \land C$, and $A \land C$ remain, but the actual feature interaction was $A \land B \land C \land D$. Thus, we get a smaller subset of interactions. Besides that case, it could also be possible that we retain an empty set of potential feature interactions for the considered feature interaction size $t < 4$, because the previous $4$-wise feature interaction only shows up for the feature interaction size $t = 4$ and can therefore exclude the smaller subset interactions.

Furthermore, if we have dependencies or constraints between features that are included in an interaction then we cannot continue to separate the remaining interactions with new configurations because we cannot create a configuration without violating dependencies. In this case, we can only get a set of remaining interactions as an output instead of the isolated feature interaction (i.e., the remaining potential interactions cannot be chosen separately in a configuration and therefore cannot be eliminated from the set of remaining potential interactions). For example, the feature interactions $A \land B$, $B \land C$, and $A \land C$ remain, but the actual feature interaction is only $A \land B$, but through constraints of the underlying feature model we cannot create a new configuration that contains less interactions. Thus, we get a bigger subset of remaining interactions.

Besides the considered feature interactions, we can think of more complex feature interactions than only conjunctions of selected and deselected features. As a more complex
3.6 Summary

In this chapter, we have introduced our concept of combinatorial interaction analysis. That is, we provide an approach to isolate a feature interaction or at least reduce the set of potential feature interactions that could cause the software fault. Based on the input feature model and a failing and a correct configuration, we generate the set of all potential interactions by including all interactions contained in the failing configuration and exclude all interaction contained in the correct configuration. We then create new configurations to gain more information about the feature selections under which the fault still occurs. We continue to create new configurations until we have isolated the potential interactions or until we cannot reduce the set of potential interactions even further by gaining information whether the new configurations fail or not.

To create new configurations during the analysis, we have presented two strategies. In the random approach, the configurations are generated randomly, while in the divide-and-conquer approach, the configurations are created by trying to include about half of the remaining potential interactions. Assuming that we have only one fault (i.e., one feature interaction) in our system we can take advantage here to make statements about the interactions contained and not contained in the considered configuration. Furthermore, we can include further knowledge, for example expert knowledge, to reduce the set of all potential interactions we have to consider.
4. Implementation

In this chapter, we describe the implementation of our concept of combinatorial interaction analysis. In addition to the libraries used for the implementation, we show how we compute the set of all potential interactions and how we apply our analysis concept. Furthermore, we present the implementation for the two strategies to create new configurations throughout the analysis, presented in Chapter 3. First, we present an approach that generates new configurations randomly. Second, we explain the split algorithm, which generates configurations purposefully based on the set of all potential interactions remaining to be the interaction we search for.

To give the possibility to use and re-implement the algorithms, we provide two packages. First, we provide the source code of the algorithms in one implementation package. Second, we provide one test package that includes the test class and the test models for our evaluation.

This chapter is structured in four sections. In Section 4.1, we list all libraries used throughout our implementation. Moreover, the technical details and the used tools are presented. In Section 4.2, we show the concept of combinatorial interaction analysis from the programming point of view. After that, in Section 4.3, we show the random algorithm and the split algorithm to create configurations, as well as detailed steps in the split algorithm. In Section 4.4, we discuss our implementation decisions and we present the limits of the current implementation.

4.1 Libraries & Tools

Our concept is implemented in Java [2]. For feature-model handling and configuration creation we use framework support from FeatureIDE [6], i.e., we use the new FeatureIDE architectureFeatJAR. In particular, we use FeatureIDE for special functionality on feature models such as computing the number of core features or computing

---

1https://github.com/holaadiewaldfeee/formula/releases/tag/masterthesis
2https://github.com/holaadiewaldfeee/formula-analysis-sat4j/releases/tag/masterthesis
3https://github.com/FeatureIDE/FeatJAR
the atomic sets of a model. Besides that, the used libraries enable functionality to check partial configurations through a SAT Solver, which is needed when creating new configurations in the split algorithm. FeatureIDE is an integrated development environment used for developing feature-oriented software [30]. It is based on the eclipse IDE but also offers a headless library for operating with feature modelling. The source code for the eclipse plugins is available at the GitHub repository of the project.\footnote{https://github.com/FeatureIDE}

### 4.2 Procedure of Combinatorial Interaction Analysis

In this section, we show the general procedure of our analysis starting with the input data, as can be seen in Figure 4.1. Then we show how the potential interactions are calculated and how information from other configurations is used for the further analysis. Finally, we show the termination criteria of our algorithm and the possible outputs after performing the analysis.

#### 4.2.1 Combinatorial Interaction Analysis

Initially, we load the feature model to create our input for the analysis. We load the feature model via a method from FeatureIDE, which offers us the feature-model formula and the variable set. After that, we perform an atomic set analysis which replaces all literals of an atomic set with one representative literal. In our case, this analysis is important to have the possibility to isolate further interactions that are contained in an atomic set. Therefore, one literal represents an entire atomic set, which is important since we cannot have interactions containing features from atomic sets without the other features of the atomic set. Hence, the interaction is simulated after the atomic set flattening. Furthermore, we create two configurations from our model and simulate an interaction, which is important for our evaluation. It is also possible to use an already identified pair-wise interaction bug from the underlying model as input.

After the initially phase, we continue by computing the set of all potential feature interactions and create and test new configurations to get more information about the
occurrence of the interaction we search for. We update the set of potential interactions until we have either isolated the interaction or have reached our defined maximum of configurations we take for our analysis. How we calculated this maximum and how we compute the set of potential interactions is explained in the following subsections.

After we have isolated the interaction or reached the maximum of configurations, we finish our analysis. Therefore, the output of our procedure is a set of all remaining potential interactions, which only consists of the interaction we searched for in the best case. In the worst case, we could not reduce our set of potential interactions at all. Therefore, the possible outcome of our analysis is the isolated interaction or the set of remaining potential interactions, as already discussed in Chapter 3.

### 4.2.2 Computing Maximum Configuration Limit

During the implementation we noticed that in some cases all possible configurations of a feature model have to be run, which is the reason why we calculate an upper bound to limit the maximum number of configurations we consider in our analysis. This maximum number of configurations we have estimated using our random algorithm to create configurations run for 98 feature models with \( n \in \{3, \ldots, 100\} \) optional features. We compute the median and average number of configurations that are needed to isolate the feature interaction by performing 100 iterations for each model using the random algorithm, which can be seen in Figure 4.2. Here, we can see that for the increasing number of features the average of configurations needed to isolate the interaction does increase logarithmically. Because it is infeasible to test all possible configurations we approximate the maximum number of configurations we consider to get a random algorithm that scales. With respect to the approximation, we want to estimate the configurations by the features of the underlying model. We see a logarithmic curve which is why we estimate the maximum number of configurations by the logarithm of the number of features. Additionally, we double this estimate and add a constant of 100 to generate an upper bound to ensure that our approximation is not too small. It follows

\[
\text{maxConfig} = 2 \cdot \text{round}(3 \cdot \log_2(\text{numberOfFeatures})) + 100,
\]

which is used to set an upper bound in our random algorithm.

Setting this maximum number of configurations is important for the evaluation to perform in reasonable time especially for large feature models. Therefore, our computation of a maximum value of configurations offers us the possibility to finish instead of going through every possible configuration of our product line. Unfortunately, this introduces the possibility to finish our analysis before we exclude all interactions. As we can see in Figure 4.2, we never need more than 20 configurations for 98 iterations on average for generated models, which leads us to set the upper bound like this.

### 4.2.3 Computing Potential Feature Interactions

As a next step, we present the procedure of computing the set of all potential interactions that could possibly be responsible for the fault. On the basis of a failing configuration, we can compute all potential interactions that could be faulty for an interaction size \( t \).

In the following, we only search for interactions with an interaction size \( t = 2 \), but the generalization to a higher value of \( t \) is possible.
As described in the assumptions in Section 3.2, we assume that only one uniquely identifiable fault exists that is contained in the failing start configuration, which means that we only need to look at the literals contained in the faulty configuration as explained in Section 3.3.1. This allows us to exclude about 75% of all interactions to consider in our analysis from the beginning. An example of the computation of all potential interactions can be seen in Example 3.1. When calculating the set of all potential interactions, we start by listing the faulty literals from the failing configuration. Second, we remove every core or dead feature to exclude them from further analysis. Based on the remaining set of literals we compute the list of all potential interactions. After that, we reduce this set with the interactions contained in the correct configuration. This forms the initial set of potential interactions and therefore the basis of our interaction analysis.

4.2.4 Interaction Analysis to Isolate The Interaction

In combinatorial interaction analysis, we continually create new configurations, test them, and then use this information to reduce our set of all potential interactions. In Figure 4.4, we can see the general procedure of the analysis concept to isolate the interaction causing the fault. A new configuration is created and tested until the number of potential interactions is reduced to one remaining interaction or the maximum number of configurations is reached. If the configuration fails, the interactions contained in the configuration are intersected with all interactions from failing configurations. Otherwise the interactions are removed from the set of all potential interactions.
4.2. Procedure of Combinatorial Interaction Analysis

function computePotentialInteractions
    failingLiterals ← Compute all literals ∈ failingConfig
    failingLiterals ← Remove all core and dead features
    foreach lit1, lit2 ∈ failingLiterals do
        interactionsAll.add(lit1, lit2)
    end
    interactionsAll ← Remove all interactions contained in correctConfig
    return interactionsAll
end

Figure 4.3: Computation Of Potential Interactions

function isolateInteraction
    configCount ← 0
    maxCount ← computeMax()
    interactionsAll ← computePotentialInteractions()
    for interactionsAll.size() > 1 AND configCount < maxCount do
        configuration ← getConfiguration()
        if configuration.failTest() then
            failingConfigs.add(configuration)
            interactionsAll ← Intersect all interactions from failingConfigs
        else
            correctConfigs.add(configuration)
            interactionsAll ← Remove all interactions contained in correctConfigs
        end
        configCount ++
    end
    return interactionsAll
end

Figure 4.4: Isolating Interaction Through Interaction Analysis
4.3 Algorithms to Create Configurations

In the following, we present the two algorithms to create configurations. While performing the combinatorial interaction analysis, new configurations are needed, on the basis of which the analysis is then continued. The first algorithm follows a random approach that generates random configurations. The second algorithm works on a divide-and-conquer approach that splits the set of remaining interactions in two sets that each contain about half of the interactions and creates a new configuration including one set and excluding the other.

Throughout the whole process of creating configurations in each algorithm, we need SAT calls and other computations. These further operations when generating configurations are abstracted through a black box. We use this black box to check the feature-model dependencies to see if several partial configurations can be fulfilled. These satisfiability results of a partial configuration are needed in the configuration creation process to see which interactions can be included together in configurations and which cannot be part of the same configuration.

Random Algorithm

In the random approach, we generate new configurations randomly by a random sample algorithm from FeatureIDE. When creating the configurations a random seed is used, which can be set beforehand to enable reproducibility. Furthermore, the algorithm avoids to create the exact same configuration that has already been created once in this run to ensure that we do not test the same configurations twice.

Split Algorithm

In the split algorithm, we want to improve the random algorithm and create the configurations in such a way that we gain as much new information as possible for our analysis through including about half of the potential interactions. We can consider several procedures to create a configuration that contains about half the interactions. In the following we choose a splitting technique of the potential interactions to be able to apply a divide-and-conquer strategy in the interaction analysis process.

In Figure 4.5, we see the procedure of splitting the remaining interactions in about half. First, the set of all literals is divided into two sets, the left set containing about $\frac{1}{\sqrt{2}}$ literals and the right set containing about $1 - \frac{1}{\sqrt{2}}$ literals. This is important because we want the left set to contain all interactions that consist of literal pairs that only occur in the left literals set. Therefore, the right set contains all other interactions, i.e., all interactions consisting of literal pairs of the left literal set and additionally the pairs containing one literal from the left set and one literal from the right set.

The split of the potential interactions can in some cases not be performed because of dependencies between the features. If the split is successful, we can create a configuration by including the left set and exclude the right set, as described in Figure 4.6. After creating a new configuration, we check if the newly created configuration already exists in our sample of all configurations. If we are not able to create a configuration, we do the splitting step once again by covering one potential interaction and excluding all other interactions. This ensures us to gain more information about at least one further
4.4 Guarantees & Limitations

In this section, we show the limitations of the current implementation of our concept of combinatorial interaction analysis.

When considering the random algorithm, we have the possibility to generate configurations unlucky in the worst case. That is because we include no further knowledge about already excluded interactions. Therefore, the random algorithm cannot finish if more than one interaction remains, even if all remaining interactions cannot be part of different configurations because of constraints. Currently, we finish our analysis if the set of remaining interactions contains only one remaining interaction or the maximum number of configurations is reached. Hence, another termination criterion could help

---

Figure 4.5: Splitting the Potential Interactions

```
function findConfiguration
    while configuration is ∅ do
        interactionsLeft, interactionsRight ← split()
        configuration ← Include interactionsLeft AND exclude interactionsRight
    end
    return configuration
end
```

---

Figure 4.6: Pseudo Code Create Configuration With Interaction Split

potential interaction. When it is no longer possible to create a configuration through the split procedure, the algorithm terminates.

If the split cannot be created, by default one of the available potential interactions is selected and based on it a configuration is created that contains only this interaction and excludes all other remaining interactions. This guarantees that at least one interaction can be excluded for each new configuration. Hence, one interaction from all potential remaining can be a very low percentage, which is mostly beaten by a randomly generated configuration.

---

4.4 Guarantees & Limitations

In this section, we show the limitations of the current implementation of our concept of combinatorial interaction analysis.

When considering the random algorithm, we have the possibility to generate configurations unlucky in the worst case. That is because we include no further knowledge about already excluded interactions. Therefore, the random algorithm cannot finish if more than one interaction remains, even if all remaining interactions cannot be part of different configurations because of constraints. Currently, we finish our analysis if the set of remaining interactions contains only one remaining interaction or the maximum number of configurations is reached. Hence, another termination criterion could help
to exclude all interactions we can detect by our analysis if this upper bound prevents to exclude all.

The split algorithm still has potential for improvement w.r.t. configuration creation and the split procedure. In the current implementation the main issue of the split algorithm is to create the split of all potential interactions because of dependencies between the features contained in these interactions. When splitting does not work, we guarantee that we gain at least more information about one further interaction if we can create one configuration containing one potential interaction.

Due to the constraints of the model, some interactions cannot be chosen independently of each other, as for example in the case with interactions contained in an atomic set. We tackle this problem by the atomic set analysis at the beginning of our analysis. The atomic set analysis that we perform at the beginning of the procedure is an expensive operation in the current implementation that also leaves room for improvement. However, there are other cases where new configurations cannot be created, such as transitive dependencies between features that are part of different interactions, as explained in Chapter 3. In the current implementation, it is the case that no further new configurations can be created that can exclude further interactions.

4.5 Summary

In this chapter, we provide the details of the implementation of our concept of combinatorial interaction analysis. Besides listing the libraries and tools used in our implementation, we present the procedure of our concept with a technical view. Especially, we explain how we load or generate our input data required for the analysis. To perform the analysis, we have to load a feature model and generate two configurations, one containing the fault and one correct configuration. Furthermore, we present how to compute the set of all remaining potential interactions as well as how to update it with the information of a new configuration. Next, we provide two algorithms to create new configurations throughout the analysis. We can choose between the random and the split algorithm, where the random algorithm generates new configurations randomly without looking at the set of potential interactions and the split algorithm uses the information of the remaining potential interactions.
5. Evaluation

In this chapter, we will evaluate the concept with different feature models to answer the research questions presented in Chapter 1. We limit the evaluation to simulate only pair-wise feature interactions and therefore try to isolate only pair-wise interactions. For our evaluation, we consider eleven different feature models, i.e., ten industrial feature models and our running example.

At first, we discuss the research questions in Section 5.1, which compare our two algorithms from Chapter 4 to check which algorithm can isolate the feature interaction in terms of effectiveness and testing efficiency. After that, we take a look at the experiment design given in Section 5.2. In Section 5.3, we present the results of our experiment according to our research questions. After delivering our results, we discuss them in Section 5.4. To conclude, we provide a summary of the evaluation in Section 5.5.

5.1 Research Questions

In the following, we explain the research questions introduced in Chapter 1 in more detail, which we want to answer with the evaluation.

**RQ1: How effective can we isolate the feature interaction that leads to the failing configuration?**

When performing our concept of combinatorial interaction analysis, we want to take a look at how effective we can isolate feature interactions. When considering the effectiveness of an approach, we want to address the quality of the results we get. Besides that, we are interested in the remaining set of potential feature interactions after our analysis. Have we completely isolated the feature interaction or do we have a set of interactions remaining? Thus, it is also important to know if we can no longer reduce a set of interactions. How good is the approximation to the feature interaction and therefore the remaining potential interactions?
RQ2: How many configurations are needed to isolate the feature interaction?

In the second research question, we want to analyze the efficiency of our approach for each feature model for each algorithm. We want to look at the number of new configurations that must be created and tested in both of our algorithms to perform our analysis. In particular, we want to compare both algorithms in terms of the amount of the needed configurations. Further, we are interested in whether we have to create new configurations at all. The fewer configurations needed to perform our analysis, the more efficient we can perform identifying the interaction through combinatorial interaction analysis.

Moreover, we are interested in how effective we can eliminate the potential interactions for each new configuration in an experiment. So we want to be able to make a prediction about how many interactions we can eliminate for each new configuration on average.

RQ3: How long does it take to isolate the feature interaction that leads to the failing configuration?

In the third research questions, we want to check if our analysis using the two algorithms scales and how much time the procedure of interaction analysis needs in our experiments. So, we want to compare the two algorithms according to the computation time of one experiment containing the creation of new configurations and the analysis of the remaining potential interactions. Thus, we want to check which algorithm can isolate the feature interaction faster. In particular, we are also interested in knowing for which feature-model sizes our algorithms scales.

5.2 Experiment Design

In the next section, we define the experiments we want to perform for evaluation. Initially, we give an overview of the experiment design, where we describe our input feature models we evaluate. Next, we show the procedure of the experiment execution and last, we specify the technical setup we work on. As $t = 1$ is easier to detect and as $t = 2$ occur more frequent [38], in this experiment we restrict ourselves to simulating feature interactions with the interaction size $t = 2$ (i.e., pair-wise interactions). In particular, we simulate exactly one pair-wise feature interaction in each experiment in order to consider the assumption A3.

Our approach of combinatorial interaction analysis takes place in the problem space. When creating new configurations, we use many predefined methods, such as for example checking whether a partial configuration can still lead to a valid configuration. These methods are all represented in a black box, as depicted by Figure 5.1. In particular, when testing configurations we test them in a oracle, which acts as a test that tells if the faulty interaction is included in the considered configuration or not. This test simulates the process in the solution space that builds the finished product from the configuration and checks whether it works as intended. Therefore, the inner structure of the whole solution space and therefore any implementation techniques of our product line and the code structure is ignored.

We perform each experiment 100 times with new input configurations and a new simulated feature interaction. Each experiment is based on pseudo-random numbers to ensure we can reproduce the same experiment and we can also run each model on the same
configurations and the same simulated interactions. This is important to compare the results of the different algorithms for each model.

The procedure of the experiments is as follows. We want to perform several experiments, using the random and split algorithms for each feature model to create new configurations. For each of our experiments, we generate a starting configuration on which we simulate a faulty feature interaction, which can be seen graphically in Figure 5.2. Based on the first configuration and the simulated interaction, we create the second configuration that does not contain the interaction. Therefore, our independent variables are the underlying feature model, our chosen algorithm, the feature interaction size, which, in our setup, is always $t = 2$, and the seed for the pseudo-random number generator. The dependent variables we consider are the number of configurations we need to isolate the interaction and the time that is needed for one experiment.

Because of the fact that it is not easy to test all possible configurations, we approximate the maximum number of configurations we want to consider in terms of the number of features contained in our underlying feature model for the random algorithm. Regarding
the approximation presented in Chapter 4, we set the maximum number of configurations to $\text{max} = 2 \cdot \text{round}(3 \cdot \log_2(\text{numberOfFeatures})) + 100$.

## Input Feature Models

For our evaluation, we use feature models of different sizes to investigate how our analysis works depending on these models. The first feature model we want to look at is our running example feature model Breakfast from this thesis, used in Chapter 3 to explain the concept. As a further model we consider the Automotive01 and Automotive02 models, where Automotive01 has more constraints than features and Automotive02 is under-constrained, which means that it contains few constraints in contrast to the number of features. Besides that, we include the feature models axTLS, BerkeleyDB, E-Shop and GPL in our evaluation, which represent the smaller industrial feature models. Moreover, we want to compute results for a further small model Soletta, which has the peculiarity to have about twice as many constraints as features. We also perform our evaluation with two feature models, FinancialServices01 and BusyBox, which are intended to represent two medium-sized models. Besides that, we want to evaluate the Linux model in the version 2.6.33.3. A list of all evaluated feature models is given in Table 5.1. This table also shows exactly how many features and constraints the individual models have. Note, the number of features and constraints are computed by FeatureIDE [6] and ddueruem [48].

The majority of the models are listed by Kuiter et al. [50] and they are taken from a model repository by Kuiter. [50] The GPL and E-Shop models are the only two models we have included from the FeatureIDE example models. The version numbers of the feature models, to which a version number has been assigned, have been included in the table.

For each feature model, we try to isolate the feature interaction through combinatorial interaction analysis. We create new configurations for each model through the random and the split algorithm presented in Chapter 4. In Table 5.2, we show the number of variant features of the features models, because we do not consider core and dead features.
in our feature interactions, as discussed in Section 3.1. Note that the variant features are computed by excluding the core and dead features from all features. Besides that, Table 5.2 lists the number of all potential pair-wise feature interactions that can be the faulty interaction in this feature model. For example considering the Breakfast model, we compute all potential feature interactions based on a failing configuration, because each configuration includes one feature selection of each pair and there are ten pairs.

**Feature Interaction Simulation**

When simulating the interaction, we choose the feature interaction size \( t = 2 \) of the interaction. We simulate the feature interaction under the assumption \( A3 \) to have only one deterministic feature interaction in our system. First, we generate one random configuration and based on that we simulate a random feature interaction contained in this configuration by randomly choosing two feature assignments. Thus, the first generated configuration is the failing one \( c_{\text{fail}} \). After that, we generate another configuration \( c_{\text{corr}} \), which explicitly does not contain the feature interaction. Each feature is represented through a literal that can either be positive (i.e., the underlying feature is selected) or negative (i.e., the underlying feature is deselected). Thus, the simulated pair-wise interaction is represented through two literals that are only contained in the failing configuration and not in the correct configuration (i.e., \( fi = \{ l_1, l_2 \} : (fi \subseteq c_{\text{fail}}) \land (fi \notin c_{\text{corr}}) \)).

**Output Data**

After choosing the feature model, generating the two configurations, and simulating the feature interaction, we now present the possible output of our experiments. The output data divides into two possible outcomes. On the one hand, we could have the output that we have isolated our simulated feature interaction, and, on the other hand, we could only get a set of feature interactions which contains the feature interaction. In the case where the analysis results in a set of feature interactions for the split algorithm, this means that the considered feature interaction cannot be isolated further (i.e., that the remaining interactions cannot be chosen separately in a configuration and therefore
cannot be eliminated from the set of remaining potential interactions). For example, the feature interactions $A \land B$, $B \land C$, and $A \land C$ remain but the actual feature interaction is only $A \land B$, but through constraints of the underlying feature model we cannot create a new configuration that contains less interactions. Thus, we get a subset of potential interactions. Considering the random algorithm it is possible to have a set of feature interactions as output when reaching the maximum number of configurations, as described in Section 4.2.2.

**Technical Setup**

Here, we describe the technical setup that specifies the properties of the system used for the experiments. The experiments ran under Windows 11 on a machine with an AMD Ryzen 7 4700U (4.1 GHz) CPU and 16 GB RAM. Our implementation presented in Chapter 4 has been performed with the FeatureIDE version 3.9.0 in Eclipse version 2021-06 (4.20.0) under Java 1.8.0 232-b09.

**5.3 Results**

In the following, we present the results of our evaluation. We separate the results according to the research questions. The results of the three models GPL, BusyBox, and Automotive01 are discussed in more detail below to show several models exemplary for different feature-model sizes. The detailed results of the other models can be found in Chapter A.

**5.3.1 RQ1: How effective can we isolate the feature interaction that leads to the failing configuration?**

In the first research question, we are interested in how often we can actually isolate the interaction. Per model and per algorithm, we see in Figure 5.3 that for some models almost all interactions could be isolated independently of the algorithm, as for example BusyBox and axTLS. Furthermore, for each model, all 100 experiments were run through except for Automotive02, where none of the algorithms produced a result after a timeout of 12 hours. Moreover, the split algorithm timed out after 12 hours for Linux and therefore only 26 experiments were run.

Besides that, we can see that for FinancialServices01 we could only isolate 19% of the interactions with the random algorithm, where the split algorithm isolated 70% of all simulated interactions. For our small running-example model Breakfast we can see that both algorithms isolated 100% of the simulated interactions. We have also tested both algorithms for generated feature models that contain only independent optional features and have been able to find any interaction.

**Observation 1.1:** For feature models with only optional independent features, we find 100% of the pair-wise interactions for both algorithms.

For all considered models, we can find about 87% of the simulated interactions with our algorithms on average.
5.3. Results

Only 26 out of 100 experiments have been completed for the split algorithm.

Observation 1.2: With the random algorithm, we find 82.6\% of the pair-wise interactions. Also, with the split algorithm, we find 91.3\% of the pair-wise interactions.

In the following, we want to take a closer look at the three models GPL, BusyBox, and Automotive01 exemplarily. First, we want to show the reduction of interactions per configuration for each experiment. In Figure 5.4, where 36\% of the experiments cannot isolate the interaction for the random algorithm, we can see that some experiments cannot exclude further interactions after approximately the fourth configuration but compute more configurations until the maximum number of configurations is reached. In contrast, the split algorithm terminates earlier when the algorithm cannot exclude more interactions. In comparison, we can see that the split algorithm can exclude more interactions per new configuration than the random algorithm, i.e., about half of the remaining potential interactions. In Section A.1, we can make the same observation of interaction reduction by looking at the Breakfast model.

For BusyBox in Figure 5.5, the x-axis is plotted logarithmically to see the reduction more clearly. We see that the random algorithm can reduce the interactions with less configurations than the split algorithm. Besides that, we see that in the split algorithm there are many curves that are very flat for a very long time and hardly exclude any interaction, but individual curves in one step exclude almost all interactions.

The results from Automotive01 in Figure 5.6, we can see similar curves as with the previous model. In the lower plot, we can see at approximately 200 configurations the reduction that excludes about half of the interactions. When comparing the random algorithm to the split algorithm, we can see that the random algorithm needs less configurations to isolate the interactions.
Figure 5.4: Reduced Interactions per Configuration of \textbf{GPL}

Figure 5.5: Reduced Interactions per Configuration of \textbf{BusyBox}
Figure 5.6: Reduced Interactions per Configuration of Automotive01

**Observation 1.3**: For models with a higher number of features, the random algorithm is more effective in isolating the interaction.

### 5.3.2 RQ2: How many configurations are needed to isolate the feature interaction?

The number of configurations to be tested is an important parameter because generating and testing such configurations is often very time-consuming. With these results of needed configurations, one can estimate how much effort will go into the later generation and testing of the configurations. In this research question, we want to compare the needed configurations per algorithm.

In Figure 5.7, we see the distribution of the number of configurations needed in each experiment for the model GPL. We divide the results in isolated interactions and not isolated interactions. We see that the random algorithm needs on average 58 configurations till it determines. In contrast, the split algorithm needs on average 9 new configurations. Furthermore, the random algorithm needs on average 15 configurations till the interaction is isolated. In contrast, the split algorithm needs on average 9 new configurations. Besides that, we can see that the random algorithm tries to test all configurations until the maximum number is reached that is why the not isolated value is equal in each iteration.

**Observation 2.1**: The random algorithm tests new configurations until it reaches the maximum number although no further interactions are excluded per step.

In Figure 5.8, we can see the distribution of the number of configurations needed in each experiment for the BusyBox model. Here, we can see that the maximum number of
considered configurations is higher in the split algorithm. The number of configurations to isolate the interactions is also much higher as in the random algorithm even though both algorithms isolate 99% of all simulated interactions.

In Figure 5.9, we can see the distribution of the number of configurations needed for both algorithms for the Automotive01 model. Here, we see similarities to the Busybox, too. The split algorithm needs more configurations to determine whether it can isolate the interaction or not.

**Observation 2.2**: For models with a higher number of features, the random algorithm needs less configurations to isolate the interactions than the split algorithm.

Further data about the minimum and maximum needed configurations for each algorithm per feature model are found in Section A.2.

### 5.3.3 RQ3: How long does it take to isolate the feature interaction that leads to the failing configuration?

In the last research question, we want to compare the computation time of both algorithms for each feature model. We want to investigate how long it takes until one experiment terminates and whether the interaction is isolated or not.

In Figure 5.10, we can see the computation-time distribution for the experiments per algorithm for the GPL model. Here, we can see that the algorithms that the not isolated average time is higher for the random algorithm.

In Figure 5.11, we can see the needed time distribution for the BusyBox model. Here, we can see that the split algorithm needs more time to isolate the interaction on average, but we must take into account that the split algorithm needs more configurations, as
5.3. Results

Figure 5.8: Number of Needed Configurations for BusyBox

Figure 5.9: Number of Needed Configurations for Automotive01
Figure 5.10: Computation Time for GPL

mentioned in the results from the first research question. Moreover, we see that the time needed for one experiment with the split algorithm has a bigger variance of computation times than the random algorithm.

In Figure 5.12, we can see the computation time distribution to isolate the interaction for both algorithms for the Automotive01 model. Note that the axes are displayed in minutes. When comparing both algorithms, we can see that the split algorithm needs on average 25.6 seconds to determine whether the interaction is isolated or not. In contrast, the random algorithm needs 0.2 seconds the compute one experiment on average.

**Observation 3**: The split algorithm needs more time for one experiment on average in contrast to the random algorithm.

Further data about the minimum and maximum needed computation time for each algorithm per feature model are found in Section A.3. Besides that, we present the time needed for each configuration per algorithm.

### 5.4 Discussion

In this section, we discuss our results from the previous section for each model and algorithm. We address the three research questions defined beforehand.

**RQ1: How effective can we isolate the feature interaction that leads to the failing configuration?**

Considering the effectiveness of our interaction analysis for both strategies random and split, we want to discuss our observations made in Section 5.3. For feature models with only optional independent features, we can find 100% of the pair-wise interactions, because each interaction can be part of a configuration independent of other interactions.
Figure 5.11: Computation Time for **BusyBox**

Figure 5.12: Computation Time for **Automotive01**
More precisely, any feature can be selected or deselected independently of other features, and thus the two features of an interaction can be set without having to set other features automatically because of constraints. For example, the model Breakfast has only one cross-tree constraint including two features. Here, we see that every simulated interaction is isolated through performing our analysis with both algorithms.

For some models we cannot isolate every simulated interaction, because of the dependencies between the features included in the interactions. For example, for the simulated interaction $[-31, 14]$ in the GPL model we have two remaining potential interaction $[-31, 14], [-5, 14]$. Considering these two remaining interactions, we cannot generate a configuration that contains only one of the interactions because of the underlying constraints. In our current implementation it is not possible to identify these dependencies between features.

When we have features in an interaction that are included in cross-tree constraints, we can see that we cannot isolate the feature interaction but only a set of remaining interaction pairs. For example, we have $A \land B$, $B \land C$, and $A \land C$, and we have the cross-tree constraints $A \Leftrightarrow B$ and $B \Leftrightarrow C$, we could not create any configuration that does not contain these three remaining feature interactions independently. Hence, we have reduced the set of feature interactions as far as possible. Here, we could have simulated the feature interaction $B \land C$ and get as result that we have not isolated this interaction.

Looking at the FinancialServices01 model, we can only isolate 70% of all interactions by using the split algorithm, because of the already mentioned dependencies and the fact that we are unable to create configurations that separates the remaining potential interactions. In contrast when considering the random algorithm, we only have an interaction-isolation rate of 20%. Here, we can see that our chosen maximum number of configuration is disadvantageous, because if we did not have the maximum number of configurations, more interactions could be found. These lower percentage of isolating is due to the fact that FinancialServices01 is a very constraint-rich feature model.

**RQ2: How many configurations are needed to isolate the feature interaction?**

With regard to the second research question, we want to discuss our observations made in Section 5.3. We want to compare both algorithms in terms of number of needed configurations to isolate the interaction as well as to reduce the set of remaining interactions.

Considering the random configuration algorithm, if we generate configurations randomly we can possibly have many configurations that exclude no further feature interactions from the remaining set of all potential interactions. In this case, we would have to go through every further configuration until the maximum value is reached, as can be seen in the results for not isolated interactions. Thus, this brings us to the point where we do not gain any information about the potential interactions but continue to create and test further configurations. Therefore, the termination criterion in the random algorithm can possibly be optimized.

Looking at the split algorithm, we have the problem of the case when no configuration is creatable for the computed feature split. In that case, we include only one interaction and exclude all others, which leads to the fact that we have only information about one remaining interaction if this is the interaction we are looking for. Obviously, if we do
this for all remaining interactions, potentially many configurations are needed until the algorithm terminates.

**RQ3: How long does it take to isolate the feature interaction that leads to the failing configuration?**

Given the needed computation time when proceeding our interaction analysis for the two strategies random and split, let us discuss our observations from Section 5.3. Regarding the computation time for an experiment, we see that the more configurations are needed, the time increases. Furthermore, we can see that our algorithms both scales for ten out of eleven feature models. Considering the Linux model, we need approximately 1,378.01 seconds on average to perform our analysis. The random algorithm has the advantage that it can quickly create configurations, but the analysis with the randomly created configurations needs many configurations in case the interaction cannot be completely isolated.

Moreover, we see that the split algorithm needs more time to perform an experiment on average than the random algorithm. The disadvantage of the split algorithm is that it requires effort in the split process and in trying to create the configurations. Another disadvantage is that the more configurations we need throughout our analysis, the longer it takes to isolate the interaction. Here we see a strong dependency to the second research question.

The actual testing is represented by an oracle that has perfect fault information, which reduces the time needed for testing in our evaluation to a minimum, but does not apply in the real-world context. Therefore, it must be noted that the times are only a guideline and the number of configurations is more indicative of the computation time that may be required.

### 5.4.1 Threats to Validity

In this section, we provide the internal and external threads to validity. By providing the internal validity we want to show the quality of the underlying measurements and the impact of the results of our work. The external validity describes how the results obtained in one particular experimental setup can be applied to other setups beyond this thesis [20].

**Internal Validity**

In our work, we use pseudo-random numbers to create configurations and simulate interactions, which means that we may get good or bad results by coincidence. Due to the use of random numbers, measured values and their derived values like the average or the mean can be inaccurate. We address this problem by performing 100 iterations of experiments for each model and algorithm in order to make a more general prediction of the results of our experiments. Furthermore, we present the results graphically through box plots.

To measure the runtime of the experiments we use the java method `System.nanoTime()`, which means that any measured values may be inaccurate due to the underlying measurement method. Furthermore, the used integrated development environment can influence some time measurements, which must be taken into account.
External Validity

By discussing the external validity we get information about the extent to which the effect of our analysis persists under different conditions or outcomes [20]. Thus, we want to determine to what extent our concept and our results can be generalized.

First, we want to look at the assumptions defined in Section 3.2, which determine the limitations of generalization of our concept.

A1: The feature model has at least one valid configuration that contains the feature interaction.

A2: The feature model has at least one valid configuration that does not contain the feature interaction.

A3: The occurring fault is deterministic.

The consequences from A2 are that we need at least one configuration that does not contain the fault, which cannot be guaranteed when for example updating the underlying feature model and adding new features. Another major consequence from our assumptions, in particular A3, is that we assume that our system has only one uniquely identifiable fault that is reproducible, which cannot always be guaranteed in real-world setups. Furthermore, we cannot guarantee that only one feature interaction causes the fault or that one interaction causes only one fault. However, as we aim to demonstrate the feasibility of the overall approach, we would argue that the limitations implied by the assumptions constitutes a good trade-off.

Our experiments have been performed only on eleven feature models we have chosen, which therefore excludes the possibility of transferring the results and outcomes to any other feature model. However, when selecting the models to be studied, we tried to select as many models as possible that differ in the number of features and constraints. Most of the selected models are industrial models, which shows that our concept is probably transferable to more industrial models.

5.5 Summary

In this chapter, we have presented the evaluation of our approach combinatorial interaction analysis. The research questions we wanted to answer were how effectively we can isolate the interaction and how much configuration and time is needed to do so. The input for one experiment on which we want to perform our analysis is a feature model, a failing and a correct configuration as well as a simulated pair-wise feature interaction. In particular, we use feature models of different sizes, for example GPL and Busybox to investigate how our analysis works depend on the model. We proceed each experiment 100 iterations per feature model and per algorithm with pseudo-random configurations and simulated interactions.

Regarding our results we find 82.6% of the pair-wise interactions with the random algorithm and we find 91.3% of the pair-wise interactions with the split algorithm. The
number of needed configuration is less for the random algorithm for large feature models, because of the split procedure in the split algorithm. This split step offers optimization possibilities in that case when no half split can be performed. The computation time spent per algorithm obviously correlates with the number of configurations, but the split algorithm also needs more time to calculate the interaction split compared to the random algorithm that generates new configuration without less computations. Moreover, the actual testing is represented by an oracle that has perfect fault information, which reduces the actual testing time to a minimum, but does not apply in the real-world context.
6. Related Work

In the following, we give an overview of related work and discuss the differences to our work. Besides product sampling and combinatorial interaction testing, we discuss how feature interactions and detecting interaction bugs have been handled so far.

**Product Sampling**

Nowadays, there are many sampling algorithms that deal with generating a representative sample of configurations to be used for testing [5, 9, 28, 34]. Many existing sampling algorithms do not scale well to large product lines, because they need a substantial amount of time to generate the samples [33]. In particular, samples are often not available until a sampling algorithm completely terminates, which is not practicable in most cases. Thus, Al-Hajjaji et al. [33] present an incremental approach of product sampling for pair-wise interaction testing to offer configurations before finishing the algorithm. Furthermore, Krieter et al. [43] present a sampling algorithm that provides partial results.

One common known baseline for sampling algorithms is uniform sampling, where we generate randomly configurations and every valid configuration is equally likely to be selected [42]. These algorithms do typically not produce $t$-wise coverage for large systems and this is why distinct algorithms are required for $t$-wise sampling. In our approach, we use randomly generated configurations, too. Several greedy algorithms generate a set of products that guarantee a certain degree of coverage [40] but our concept of combinatorial interaction analysis needs no sample that fulfills several coverage criteria but can be performed with any sampling algorithm. None of the approaches deal with the actual identification process of the fault in that case when a configuration is faulty and so the developer is on his own. At this point, our approach becomes relevant by supporting the developer in finding faults that are caused by feature interactions.

**Combinatorial Interaction Testing**

Combinatorial interaction testing computes a small set of valid configurations by a sampling algorithm, in which each combination of $t$ features appears in at least one configuration [9, 24, 43]. One of the widely-used testing approaches is pair-wise testing,
which is based on the assumption that most faults are caused by one-wise or pair-wise interactions [18]. Nie et al. provide a literature study about several papers that have dealt with the concept of combinatorial interaction testing to detect faults triggered by interactions [14]. Several approaches have been proposed to reduce the number of configurations to be tested by sampling a subset of products achieving a certain degree of combinatorial interaction coverage [9, 33, 40]. As in our approach, the focus is on covering certain \( t \)-wise feature interactions. In contrast, we need no specific coverage criteria when generating configurations and therefore, we often have to generate fewer configurations.

Constrained combinatorial interaction testing adapts the technique by focusing on scenarios where some features cannot coexist in the configuration [13]. With this adaptation in the area of software testing, the problem of handling real-world constraints is considered [13, 32, 37]. Ahmed et al. provide a literature study about the categorization of current contributions in the field of constrained interaction testing [13]. Finding an efficient way to handle constraints is the key factor in wider applicability of combinatorial interaction testing techniques, and current techniques leave room for improvement [5, 31, 32, 37]. Regarding our approach, we have to deal with constraints when generating configurations and separating some feature interactions that are dependent through constraints. This is a problem when we want to exclude interactions from our set of remaining potential interactions. To tackle this problem, we perform, for example, an atomic set analysis to consider the constraints that form atomic sets.

**Sampling Including Solution Space or Expert Knowledge**

Most sampling techniques only consider the problem space to generate suitable configurations, but including further information from solution space, such as test artifacts or code coverage, has rarely been used and seems to be understudied [40]. In all the papers cited by Varshosaz et al. [40] only the problem space was evaluated but not the solution space. Our approach also includes only the problem space.

As scalability is the main issue in many sampling approaches, a further approach to improve sampling is to include expert knowledge, such as assigning priorities to features or knowledge about shared resources [17, 23, 40]. Oster et al. [8] were the first to compute a representative sample based on an input set of predefined configurations. Ensan et al. [11] provide an approach to enable experts to rule out certain configurations by defining a set of features that need to be considered [11]. In our concept, this approach could be helpful to reduce the set of potential feature interactions we have to consider. Moreover, by allowing experts to specify test costs for each feature we can better estimate testing effort for some specific configurations [22]. Because of the fact that the time budget for testing is limited or unknown, the order in which products are tested can play an important role in effective testing [41]. Regarding our analysis concept, we present an idea of including further knowledge to reduce the set of potential feature interactions in Section 3.4.

**Test-Cases for Product Lines**

Considering the selection and generation of test-cases, there are several works that tackle the problem of generating suitable configurations by sampling algorithms. However, they do not deal with the problem of generating suitable test-cases that fit for all
products. The problem of how to create the test-cases are tackled by several papers [9, 10, 15, 18]. Baller et al. [27] provide an optimization framework for test suites. One strategy is to generate test-cases through analyzing a feature model in order to generate a test suite that satisfies t wise coverage [9, 25, 26]. In contrast to these works, we focus on generating configurations to narrow down the fault and we do not take into account how to generate suitable test-suites when generating configurations.

**Feature Interaction Detection**

Detecting and identifying feature interactions is still a big challenge [21]. Many variability bugs involve multiple features and are hence feature-interaction bugs [38]. Calder et al. [4] detect feature interactions by pair-wise analysis. Furthermore, Kuhn et al. [1] state that higher order interactions are less likely to occur than pair-wise interactions.

Currently, we do not know how common interaction faults are in practice or whether current interaction testing techniques are effective at finding the faults [12]. Abal et al. [38] provide a variability bug database with real-world interaction bugs, where 41 faults are caused by single features and 57 faults by feature interactions with an interaction size $t \geq 2$, but researches still lack of concrete variability bugs in large systems [38]. Besides, variability bugs may also involve non-locally defined features (i.e., features defined in another subsystem) [38]. Of course, we can never know what interaction size is required to detect all faults in a system [1]. This problem also is present in our concept of combinatorial interaction analysis. Currently, we focus on pair-wise interactions.

**Machine Learning Approaches**

There are several machine learning approaches regarding configuration generation. Temple et al. [36] generate configurations and test them through an oracle to resolve failing configurations by inferring constraints. Siegmund et al. [19] present a tool to analyse several configuration results to get as output which features interact, with focus on non-functional requirements. In contrast to machine learning approaches where we cannot guarantee that the learning process leads to exact results, we have a concrete approach to be able to determine an exact set of potential interactions.
7. Conclusion

In this thesis, we provide the first approach to identify feature interactions through combinatorial interaction analysis. When features interact, there can be unforeseen faults and security vulnerabilities, which is one reason why it is important to find these feature interactions. While with current testing strategies one can identify failing configurations, it is challenging how to identify the feature interaction actually causing the fault. In particular, it is difficult to identify feature interactions due to the potentially large number of interactions for large feature models. Opposed to combinatorial interaction testing, we want to perform an interaction analysis based on a failing configuration, where we want to analyze the potential feature interactions that can cause the software fault until we have isolated the feature interaction.

In combinatorial interaction analysis we consider all potential $t$-wise interactions, that can be responsible for the software fault. We determine this set by looking at the failing configurations to see which interactions are covered. By excluding all feature interactions that are included in correct configurations, our analysis provides conclusions that can be drawn about the remaining potential interactions. We present two approaches to create new configurations, a random and a divide-and-conquer strategy, to create new configurations. By creating new configurations, new information about the features involved in the configurations is gathered, which is then used to continue with our analysis until we have isolated the feature interaction. Regarding the divide-and-conquer strategy we apply a concept, where we want to take the remaining interactions and recursively split them to be part of new configuration to see if the new configuration still fails or succeeds.

We performed 100 experiments for each feature model and each algorithm with pseudorandom configurations and simulated pair-wise interactions. With the random algorithm we have an approach that requires less time for the generation of configurations and still scales well even for large models. Our results show that we can find 82.6% of the pair-wise interaction for the considered feature models. In comparison the split algorithm isolates 91.3% of the simulated pair-wise interactions. Therefore, both algorithms scale for large models.
In summary, we provide a first concept and implementation to tackle the problem of identifying and isolating feature interactions through combinatorial interaction analysis. Our concept forms a good basis to be adapted in future work.
8. Future Work

In this chapter, we present an overview on possible future work and possibilities to optimize our approach. Our concept of combinatorial interaction analysis could be performed this way only through certain constraints and under certain assumptions, which we aim to resolve for more applicability in industry. With regard to the research questions of this thesis, we are interested in how we can isolate more or all of the interactions. Can we design an algorithm in such a way that we can isolate all interactions or at least get an even smaller set of remaining interactions? Can we find the interactions with less configurations or get similar values with less configurations? Can the algorithm terminate faster?

Types of Faults

In the current analysis, we disregard the specific kind of fault that can cause the interaction and restrict ourselves to the constraint that the fault is deterministic. Future work can take into account the different faults that feature interactions can cause and we can tackle the problem to look at the distinction between runtime error and compile error. Furthermore, we can look at how to handle sporadic faults and flaky tests, which means that the test intermittently fails for no apparent reason or works on one particular machine and fails with continuous integration. Currently, we provide our analysis under the assumption that we have only one fault in our system. As future work we can consider more than one feature interaction or multiple faults caused by one interaction. Additionally, we can imagine more than one feature interaction with different interaction sizes $t$. Here, we have to look at the failing tests and the amount of bugs that occur through feature interactions. Moreover, we can look at the concept of error masking, which defines the case in that one fault does never occur or not occur in some configurations because of other earlier occurring faults in the system.

Currently, we only consider feature interactions that consist of conjunctions of selected or deselected features. In the future, we could include complex interactions in the analysis, which are represented by disjunctions of selected or deselected features. In this case, it might be helpful to distinguish whether the features of a complex feature interaction cause the same fault or a different fault.
Instead of only simulating feature interactions and check whether our algorithms can isolate the interaction or not, we can also test real-world feature interaction bugs. The variability bug database\(^1\) collected by Abal et al. [38] can serve as a good basis. Further, it is conceivable that one tries to find the same interaction faults with machine learning and whether this finds the interaction faults as well as our algorithm.

**Including Further Information**

Besides the already presented analysis concepts in Chapter 3, a possible future approach can be to look at including further information for our analysis. Here, the current implementation can serve as a good basis to reduce the analysis effort. Moreover, we can perform the interaction analysis by considering some parts of the solution space as white box, such as source code and statement coverage as well as sampling algorithms. We can also think of applying the interaction analysis by including source code of the considered features or looking at some types of code coverage (e.g., statement coverage) to reduce the time of analysis [34].

**Reducing the Set of Potential Interactions**

One possibility to reduce the set of potential interactions is to only consider concrete features and omit abstract features. In our current algorithm, both types (i.e., concrete and abstract features) are considered, but in most product lines, abstract features are only there for structural reasons [16].

Furthermore, one can include more information as input data, e.g., expert knowledge or an entire initial sample instead of only two start configurations. A basic form of this optimization approach is already presented in Section 3.4.1. Moreover, we can consider the evolution of the underlying feature model to get information about what features are new or in which feature constellation the fault did not occur.

Currently, we are not able to isolate all interactions due to the dependencies between features. This is where we could improve the analysis in the future by looking at the dependencies between the features contained in multiple interactions separately. For now, we perform an atomic set analysis to consider these dependencies between features, but further dependencies that occur when containing multiple features in the same configuration are currently not considered. Here, further optimizations of the algorithms can take place that include further dependencies between features by generalizing atomic sets to \(t\)-wise interactions.

**\(T\)-Wise Interactions**

In the current implementation, a few preparations are already taken to vary the interaction size of the feature interactions. At this point, one can continue and perform further evaluations on higher interaction sizes. The combinatorial interaction analysis can then proceed in such a way that it starts with the interaction size \(t = 1\) and continues to search for the interaction with higher values of \(t\) as described in Chapter 3.

Besides that, one can also include \(t\)-wise sampling algorithm in the analysis and thus get more information about the currently considered interactions of size \(t\) and possibly draw conclusions about smaller interaction sizes.

\(^1\)http://vbdb.itu.dk/database.html
As an outlook, this can also be used to perform a fault verification based on a found interaction with the next higher interaction size to check whether the found interaction is actually the searched interaction. This verification can also be performed with found interaction bugs in real-world product lines.

**Formal Verification**

In this thesis, we demonstrated the general feasibility of our concept. To this end, we defined assumptions with the goal of trading practical relevance for easier demonstration of the concept. In future work, we aim to lift these assumptions to make combinatorial interaction analysis more applicable to real-world scenarios. Subsequently, we envision a more sophisticated verification and evaluation with respect to soundness (i.e., are the found interactions actual faulty interactions), completeness (i.e., can we find all interactions), optimality (i.e., is the found interaction minimal), and complexity (i.e., scales to real-world scenarios).
9. Acknowledgements

I would like to thank the members of the Department of Software Engineering and Programming Languages at the University of Ulm for the enriching conversations and advice, in particular, Prof. Dr. Thomas Thüm and Dr. Sebastian Krieter for the supervision and mentoring. Besides that, the source code was developed in collaboration with Sebastian Krieter and Jens Meinicke, to which we would like to express our thanks, too. Further thanks goes to Prof. Dr. Malte Lochau from the University of Siegen for supervising and proofreading the work. Another thanks goes to my friends and family, especially Karl Wiesmayer, Lukas Güthing, Rahel Arens, Kevin Jedelhauser, Johannes Herschel, Sven Patrick Meier, Heiko Rettich, Stefan Wintergerst, Tobias Lahmann, and Johannes Vedder. Thanks for all the exciting discussions and conversations about my master’s thesis. Moreover, I would like to give a special thanks to Tobias Heß, because of the weekly discussions about the product-line domain and for sharing an office with me. Without him I would not have written this thesis.
A. Appendix

In this chapter, we can see further detailed results of our evaluation from Chapter 5. In Section A.1, one can see the reduced interactions per configuration of each feature model not displayed in Chapter 5. Here, we take a look at a combination of the first research question "How effective can we isolate the feature interaction that leads to the failing configuration?" and the second research question "How many configurations are needed to isolate the feature interaction?". In Section A.2, more details about the needed configurations per model are given, such as the maximum and minimum needed number of configuration for each algorithm. In Section A.3, more detailed results about the third research question "How long does it take to isolate the feature interaction that leads to the failing configuration?" are given, such as the average of needed time for each algorithm.

A.1 Reduced Interactions per Configuration

In the following figures you can see the burn-down charts from the feature models not displayed in Chapter 5. For axTLS Figure A.1, E-Shop Figure A.4, Linux Figure A.6, and Soletta Figure A.6, we see that many curves show a similar trend. Furthermore, we see in most of the models for the split algorithm that most curves have a weak negative slope, because of the current implementation of the split algorithm.

In the burn-down chart of the Breakfast model Figure A.3, we see the clear improvement of the split algorithm as presented in the concept in Chapter 3. More precisely, the interactions are excluded by about half each time in contrast to the interactions in the random algorithm.

A.2 Needed Configurations to Isolate the Feature Interaction

In this section, one can see further details about the computation time for each model per algorithm. In Table A.1, we can see that on average more configurations were always needed for the split algorithm than for the random algorithm except for the models BerkeleyDB, Breakfast, and GPL.
Figure A.1: Reduced Interactions per Configuration of axTLS

Figure A.2: Reduced Interactions per Configuration of BerkeleyDB
A.2. Needed Configurations to Isolate the Feature Interaction

Figure A.3: Reduced Interactions per Configuration of Breakfast

Figure A.4: Reduced Interactions per Configuration of E-Shop
Only 26 out of 100 experiments have been completed for the split algorithm.

Figure A.5: Reduced Interactions per Configuration of FinancialServices01

Figure A.6: Reduced Interactions per Configuration of Linux
A.2. Needed Configurations to Isolate the Feature Interaction

![Reduced Interactions per Configuration of Soletta](image)

Table A.1: Number of Needed Configurations per Algorithm

<table>
<thead>
<tr>
<th>Model</th>
<th>Max</th>
<th>Min</th>
<th>all</th>
<th>isolated</th>
<th>not isolated</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Automotive01</strong> - random</td>
<td>164</td>
<td>16</td>
<td>70.6</td>
<td>47</td>
<td>50.1</td>
</tr>
<tr>
<td><strong>Automotive01</strong> - split</td>
<td>1,348</td>
<td>16</td>
<td>512.7</td>
<td>522.5</td>
<td>493.6</td>
</tr>
<tr>
<td><strong>Automotive02</strong> - random</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Automotive02</strong> - split</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>BerkeleyDB</strong> - random</td>
<td>134</td>
<td>7</td>
<td>44.6</td>
<td>22.5</td>
<td>30</td>
</tr>
<tr>
<td><strong>BerkeleyDB</strong> - split</td>
<td>51</td>
<td>4</td>
<td>23.5</td>
<td>21</td>
<td>23.3</td>
</tr>
<tr>
<td><strong>Breakfast</strong> - random</td>
<td>17</td>
<td>2</td>
<td>6.7</td>
<td>6</td>
<td>6.7</td>
</tr>
<tr>
<td><strong>Breakfast</strong> - split</td>
<td>5</td>
<td>1</td>
<td>3.1</td>
<td>3</td>
<td>3.1</td>
</tr>
<tr>
<td><strong>BusyBox</strong> - random</td>
<td>154</td>
<td>13</td>
<td>28.3</td>
<td>24</td>
<td>27</td>
</tr>
<tr>
<td><strong>BusyBox</strong> - split</td>
<td>536</td>
<td>11</td>
<td>155.3</td>
<td>126.5</td>
<td>151.5</td>
</tr>
<tr>
<td><strong>axTLS</strong> - random</td>
<td>140</td>
<td>8</td>
<td>22.3</td>
<td>17</td>
<td>21.1</td>
</tr>
<tr>
<td><strong>axTLS</strong> - split</td>
<td>69</td>
<td>8</td>
<td>31.9</td>
<td>31.5</td>
<td>31.9</td>
</tr>
<tr>
<td><strong>E-Shop</strong> - random</td>
<td>146</td>
<td>10</td>
<td>30.1</td>
<td>23.5</td>
<td>28.9</td>
</tr>
<tr>
<td><strong>E-Shop</strong> - split</td>
<td>149</td>
<td>9</td>
<td>65</td>
<td>61.5</td>
<td>64.5</td>
</tr>
<tr>
<td><strong>FinancialServices01</strong> - random</td>
<td>156</td>
<td>9</td>
<td>139.6</td>
<td>156</td>
<td>69.8</td>
</tr>
<tr>
<td><strong>FinancialServices01</strong> - split</td>
<td>326</td>
<td>14</td>
<td>142.5</td>
<td>141</td>
<td>127.8</td>
</tr>
<tr>
<td><strong>GPL</strong> - random</td>
<td>126</td>
<td>4</td>
<td>58.3</td>
<td>27</td>
<td>20.1</td>
</tr>
<tr>
<td><strong>GPL</strong> - split</td>
<td>13</td>
<td>2</td>
<td>8.5</td>
<td>8</td>
<td>8.6</td>
</tr>
<tr>
<td><strong>Linux</strong> - random</td>
<td>174</td>
<td>20</td>
<td>67.7</td>
<td>52</td>
<td>66.5</td>
</tr>
<tr>
<td><strong>Linux</strong> - split</td>
<td>3,472</td>
<td>24</td>
<td>1,699.9</td>
<td>1,555.5</td>
<td>1,699.9</td>
</tr>
<tr>
<td><strong>Soletta</strong> - random</td>
<td>138</td>
<td>6</td>
<td>27.7</td>
<td>19</td>
<td>21.9</td>
</tr>
<tr>
<td><strong>Soletta</strong> - split</td>
<td>73</td>
<td>7</td>
<td>33.2</td>
<td>32.5</td>
<td>32.4</td>
</tr>
</tbody>
</table>
A.3 Computation Time to Isolate the Feature Interaction

In this section, one can see further details about the computation time for each model per algorithm. In Table A.2, we can see that many of the measured times are in millisecond range, but the split algorithm for Automotive01, BusyBox, and FinancialServices01 assuming values up to 500 seconds. For Linux we have values between one and 13 seconds even for the random algorithm. With the split algorithm, maximum values of approximately 3070 seconds per experiment are computed. Furthermore, we can see time needed per configuration for each algorithm in Table A.3.
### Table A.2: Computation Time per Algorithm in Seconds

<table>
<thead>
<tr>
<th>Model</th>
<th>Max</th>
<th>Min</th>
<th>all</th>
<th>Visited</th>
<th>not isolated</th>
<th>not visited</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automotive01 - random</td>
<td>0.499</td>
<td>0.107</td>
<td>0.2043</td>
<td>0.181</td>
<td>0.1831</td>
<td>0.167</td>
</tr>
<tr>
<td>Automotive01 - split</td>
<td>482.31</td>
<td>0.329</td>
<td>25.5978</td>
<td>21.663</td>
<td>25.203</td>
<td>21.492</td>
</tr>
<tr>
<td>Automotive02 - random</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Automotive02 - split</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>axTLS - random</td>
<td>0.037</td>
<td>0.001</td>
<td>0.0038</td>
<td>0.003</td>
<td>0.0036</td>
<td>0.003</td>
</tr>
<tr>
<td>axTLS - split</td>
<td>0.125</td>
<td>0.003</td>
<td>0.0149</td>
<td>0.012</td>
<td>0.0149</td>
<td>0.012</td>
</tr>
<tr>
<td>BerkeleyDB - random</td>
<td>0.031</td>
<td>0.001</td>
<td>0.0052</td>
<td>0.003</td>
<td>0.0038</td>
<td>0.002</td>
</tr>
<tr>
<td>BerkeleyDB - split</td>
<td>0.056</td>
<td>0.001</td>
<td>0.0094</td>
<td>0.007</td>
<td>0.009</td>
<td>0.0065</td>
</tr>
<tr>
<td>Breakfast - random</td>
<td>0.016</td>
<td>0</td>
<td>0.0111</td>
<td>0.001</td>
<td>0.0111</td>
<td>0.001</td>
</tr>
<tr>
<td>Breakfast - split</td>
<td>0.022</td>
<td>0.001</td>
<td>0.0018</td>
<td>0.001</td>
<td>0.0018</td>
<td>0.001</td>
</tr>
<tr>
<td>BusyBox - random</td>
<td>0.137</td>
<td>0.011</td>
<td>0.0204</td>
<td>0.018</td>
<td>0.02</td>
<td>0.018</td>
</tr>
<tr>
<td>BusyBox - split</td>
<td>2.343</td>
<td>0.031</td>
<td>0.848</td>
<td>0.592</td>
<td>0.8441</td>
<td>0.584</td>
</tr>
<tr>
<td>E-Shop - random</td>
<td>0.074</td>
<td>0.003</td>
<td>0.0081</td>
<td>0.007</td>
<td>0.0079</td>
<td>0.007</td>
</tr>
<tr>
<td>E-Shop - split</td>
<td>0.202</td>
<td>0.006</td>
<td>0.0815</td>
<td>0.0815</td>
<td>0.0807</td>
<td>0.081</td>
</tr>
<tr>
<td>FinancialServices01 - random</td>
<td>0.353</td>
<td>0.015</td>
<td>0.0794</td>
<td>0.08</td>
<td>0.0572</td>
<td>0.041</td>
</tr>
<tr>
<td>FinancialServices01 - split</td>
<td>8.895</td>
<td>0.076</td>
<td>1.383</td>
<td>0.8685</td>
<td>0.7514</td>
<td>0.617</td>
</tr>
<tr>
<td>GPL - random</td>
<td>0.022</td>
<td>0</td>
<td>0.0062</td>
<td>0.003</td>
<td>0.0026</td>
<td>0.002</td>
</tr>
<tr>
<td>GPL - split</td>
<td>0.053</td>
<td>0.001</td>
<td>0.0046</td>
<td>0.004</td>
<td>0.0044</td>
<td>0.003</td>
</tr>
<tr>
<td>Linux - random</td>
<td>12.402</td>
<td>1.432</td>
<td>2.2355</td>
<td>1.933</td>
<td>2.418</td>
<td>1.878</td>
</tr>
<tr>
<td>Linux - split</td>
<td>3,069.227</td>
<td>5.956</td>
<td>1,378.0113</td>
<td>1,560.199</td>
<td>1,378.0113</td>
<td>1,560.199</td>
</tr>
<tr>
<td>Soleta - random</td>
<td>0.047</td>
<td>0.001</td>
<td>0.0048</td>
<td>0.003</td>
<td>0.0042</td>
<td>0.003</td>
</tr>
<tr>
<td>Soleta - split</td>
<td>0.101</td>
<td>0.003</td>
<td>0.0151</td>
<td>0.014</td>
<td>0.0149</td>
<td>0.0135</td>
</tr>
</tbody>
</table>

### Table A.3: Computation Time per Configuration

<table>
<thead>
<tr>
<th>Model</th>
<th>Avg. Time in [s]</th>
<th>Avg. Time in [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automotive01 - random</td>
<td>0.0054</td>
<td>5.3847</td>
</tr>
<tr>
<td>Automotive01 - split</td>
<td>0.0604</td>
<td>60.3685</td>
</tr>
<tr>
<td>Automotive02 - random</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Automotive02 - split</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>axTLS - random</td>
<td>0.0013</td>
<td>1.2957</td>
</tr>
<tr>
<td>axTLS - split</td>
<td>0.0073</td>
<td>7.3495</td>
</tr>
<tr>
<td>BerkeleyDB - random</td>
<td>0.0500</td>
<td>50.0262</td>
</tr>
<tr>
<td>BerkeleyDB - split</td>
<td>0.2824</td>
<td>282.3585</td>
</tr>
<tr>
<td>Breakfast - random</td>
<td>0.0011</td>
<td>1.1176</td>
</tr>
<tr>
<td>Breakfast - split</td>
<td>0.0069</td>
<td>6.8746</td>
</tr>
<tr>
<td>Busybox - random</td>
<td>0.0009</td>
<td>9.154</td>
</tr>
<tr>
<td>Busybox - split</td>
<td>0.0063</td>
<td>6.2801</td>
</tr>
<tr>
<td>E-Shop - random</td>
<td>0.0499</td>
<td>49.8712</td>
</tr>
<tr>
<td>E-Shop - split</td>
<td>0.2819</td>
<td>281.9102</td>
</tr>
<tr>
<td>FinancialServices01 - random</td>
<td>0.0496</td>
<td>49.5608</td>
</tr>
<tr>
<td>FinancialServices01 - split</td>
<td>0.2806</td>
<td>280.5688</td>
</tr>
<tr>
<td>GPL - random</td>
<td>0.0001</td>
<td>0.1355</td>
</tr>
<tr>
<td>GPL - split</td>
<td>0.0005</td>
<td>0.5419</td>
</tr>
<tr>
<td>Linux - random</td>
<td>0.0490</td>
<td>48.9635</td>
</tr>
<tr>
<td>Linux - split</td>
<td>1.0434</td>
<td>1,043.4467</td>
</tr>
<tr>
<td>Soleta - random</td>
<td>0.0503</td>
<td>50.2803</td>
</tr>
<tr>
<td>Soleta - split</td>
<td>0.2828</td>
<td>282.8222</td>
</tr>
</tbody>
</table>
Bibliography


[21] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Berlin, Heidelberg, Germany: Springer, 2013. ISBN: 978-3-642-37520-0. DOI: 10.1007/978-3-642-37521-7. URL: https://doi.org/10.1007/978-3-642-37521-7 (see Pages 1, 5, 6, 8, 9, 11, 12, 19, 32, 63).


Declaration of Authorship

I hereby declare that this thesis is my own unaided work. All direct or indirect sources used are acknowledged as references. This paper was not previously presented to another examination board and has not been published as of yet.

___________________________  __________________________
Place, Date of Submission         Signature